אוניברסיטת אריאל בשומרון

קורס : רשתות תקשורת

Final project : HTTP redirect

מגישים : יאן שישפורטיש - טל מלכה

ת.ז : 341157501 - 315734616

מרצה. : מר ברגר הראל

03.9066659
03.9755815
אריאל, 40700
אוניברסיטת אריאל בשומרון ע"ר

Summary

I- *Explication of the code :*

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

```
8   # Project Vars.
9   dhcp_ip = "10.20.30.40"
10  ip_range_start = "10.20.30.41"
11  ip_range_end = "10.20.30.100"
12  router_ip = "10.20.30.101"
13  subnet_mask = "255.255.255.0"
14  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.
1 usage
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.

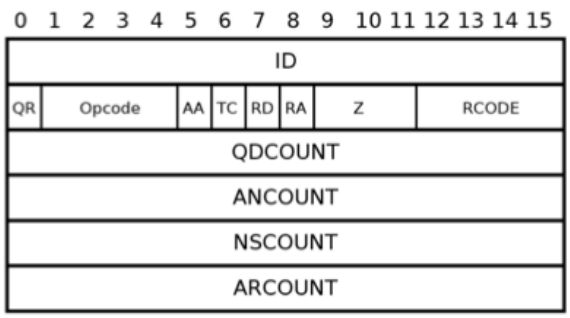
DNS Server

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.

Header Format



Code explanation:

The DNS server starts with:

```
3  # DNS Vars.
4  port = 53
5  ip = "127.0.0.1"
6  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:

```
8  # Socket init and binding.
9  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
10 sock.bind((ip, port))
11 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')
```

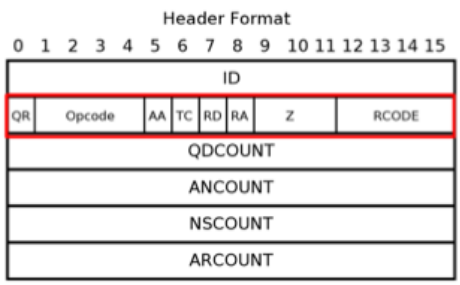
In this function we are about to

build the first byte.

(QR, Optcode, AA, TC, RD)

Second byte.

(RA, Z, RCODE)



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:

```
135 def log_request(data, addr):
136     domain, questiontype = getquestiondomain(data[12:])
137     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:

```
4   SizeOfPacket = 1024
5   clients = []
6   Server_IP = "127.0.0.104"
7   Scrap_IP = "127.0.0.34"
8   max_window = 64
9   last_received_id = {}
10  unacked_packets = {}
11  client_info = {}
```

The main and global variables of the Proxy.

Downloadable files list maker:

```python
# Get files trough TCP connection
1 usage
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:

```
40  def send_ack(server, addr, packet_id):
41      server.sendto("SIGNACK".encode(), addr)
42      client_info[addr]['congestion_window'] -= 1
43      if addr in unacked_packets and packet_id in unacked_packets[addr]:
44          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:

```
47  def send_lost_packet_signal(server, addr, expected_id):
48      print(f"Sending SIGNLOST:{expected_id} to {addr}")
49      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:

```
52  def send_signfull_signal(server, addr):
53      print(f"Sending SIGNFULL to {addr}")
54      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:

```
57  def server_statistics(addr):
58      info = client_info.get(addr)
59      if not info:
60          return "No client information available."
61
62      elapsed_time = time.time() - info['connection_time']
63      stats = f"Packets sent: {info['packets_sent']}\nPackets lost: {info['packets_lost']}\nConnected time: {elapsed_time:.2f} seconds"
64      return stats
```

Building the message when the signal “SIGNSTAT” is sent by the client.

Remove Client:

```python
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:

```python
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.

```
100                 if packet_id == last_received_id[addr] + 1:
101                     print(f"Received SIGNGET {packet_id} from {addr}: {payload}")
102                     last_received_id[addr] = packet_id
103                     message = get_files()
104                     time.sleep(0.5)
105                     if addr in unacked_packets and packet_id in unacked_packets[addr]:
106                         del unacked_packets[addr][packet_id]
107                     client_info[addr]['congestion_window'] -= 1
108                     server.sendto(f"ACKGET:{packet_id};{message}".encode(), addr)
109
110                 else:
111                     print(f"Received out of order packet {packet_id} from {addr}: {payload}")
112                     client_info[addr]['packets_lost'] += 1
113                     send_lost_packet_signal(server, addr, last_received_id[addr] + 1)
114
115             elif data.startswith("SIGNECHO:"):
116                 packet_id, timestamp = data.split(":", 1)[1].split(";", 1)
117                 packet_id = int(packet_id)
118                 last_received_id[addr] = packet_id
119                 client_info[addr]['packets_sent'] += 1
120                 print(f"Received SIGNECHO {packet_id} from {addr}. Sending back the timestamp.")
121                 if addr in unacked_packets and packet_id in unacked_packets[addr]:
122                     del unacked_packets[addr][packet_id]
123                 client_info[addr]['congestion_window'] -= 1
124                 server.sendto(f"ECHOREPLY:{packet_id};{timestamp}".encode(), addr)
125
```

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.

```
            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:

```
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:

```
8  Server_IP = "127.0.0.10"
9  Scrap_IP = "127.0.0.5"
10 url = "http://127.0.0.1/"
11 response = requests.get(url)
12 html_content = response.content
13 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: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:

```
119  def handle_lost_packet_signal(client, packet_id, sent_packets):
120      packet_data = sent_packets.get(packet_id)
121      print("-----------------------------------------------")
122      print(f"Resending packet {packet_id}: {packet_data}")
123      print("-----------------------------------------------")
124      if packet_data == "SIGNGET":
125          client.sendto(f"SIGNGET:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
126      if packet_data == "SIGNECHO":
127          client.sendto(f"SIGNECHO:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
128      if packet_data == "SIGNSTAT":
129          client.sendto(f"SIGNSTAT:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
130      else:
131          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:

```
133  def handle_ackget(data):
134      global connection_open
135      packet_id, file_list = data.split(":", 1)[1].split(";", 1)
136      my_list = file_list.split(";")[:-1]  # Remove the last empty element
137      list_queue.put(my_list)
138      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):

```
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:

```
182  def RUDP_Client(hostname):
183      global connection_open
184      client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
185      client.bind((client_ip, 49152))
186      client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
187
188      print(f"Gonna try to connect to {resolved_ip}")
189      print(" [RUDP] Sending a SIGNAL : SIGNEW to the RUDP Server")
190      client.sendto(f"SIGNEW:{hostname}".encode(), (resolved_ip, 49152))
191
192      sent_packets = {}
193      packet_id = hostname_to_numeric(hostname)
194      # Ajouter un thread pour écouter les messages entrants du serveur
195      receive_thread = threading.Thread(target=receive, args=(client, sent_packets))
196      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.

```
198      while True:
199          time.sleep(2)
200          if pause_client:
201              time.sleep(10)
202          if not list_queue.empty():
203              file_list = list_queue.get()
204              download_file(file_list)
205              break
206          elif not connection_open:
207              create_new_connection = input(
208                  "Connection closed. Do you want to send SIGNEW to create a new connection? (yes/no): ")
209              if create_new_connection.lower() == "yes":
210                  connection_open = True
211                  client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
212                  client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
213                  print(" [RUDP] Sending a SIGNAL : SIGNEW to the RUDP Server")
214                  client.sendto(f"SIGNEW:{hostname}".encode(), (resolved_ip, 49152))
215              else:
216                  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.

```
218         else:
219             # input message to let the user what signal is going to be sent
220             print("-------------------------------------------------------")
221             print("This is the list of signals you can send :")
222             print("SIGNGET : to get the redirection to the server HTTP")
223             print("SIGNEND : to end the connection")
224             print("SIGNECHO : to know the ping with the server RUDP")
225             print("SIGNSTAT : to know the stat of your connection with the server RUDP")
226             print("-------------------------------------------------------")
227 
228             message = input("SIGNAL :")
229             packet_id += 1
230 
231             if message == "SIGNGET":
232                 print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNGET to the RUDP Server")
233                 client.sendto(f"SIGNGET:{packet_id};{message}".encode(), (resolved_ip, 49152))
234                 sent_packets[packet_id] = message
235 
236             elif message == "SIGNEND":
237                 print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNEND to the RUDP Server")
238                 client.sendto(f"SIGNEND:{packet_id};{message}".encode(), (resolved_ip, 49152))
239                 sent_packets[packet_id] = message
240 
241             elif message == "SIGNECHO":
242                 timestamp = time.time()
243                 print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNECHO to the RUDP Server")
244                 client.sendto(f"SIGNECHO:{packet_id};{timestamp}".encode(), (resolved_ip, 49152))
245                 sent_packets[packet_id] = message
246 
247             elif message == "SIGNSTAT":
248                 print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNSTAT to the RUDP Server")
249                 client.sendto(f"SIGNSTAT:{packet_id}".encode(), (resolved_ip, 49152))
250                 sent_packets[packet_id] = message
251 
252             else:
253                 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.

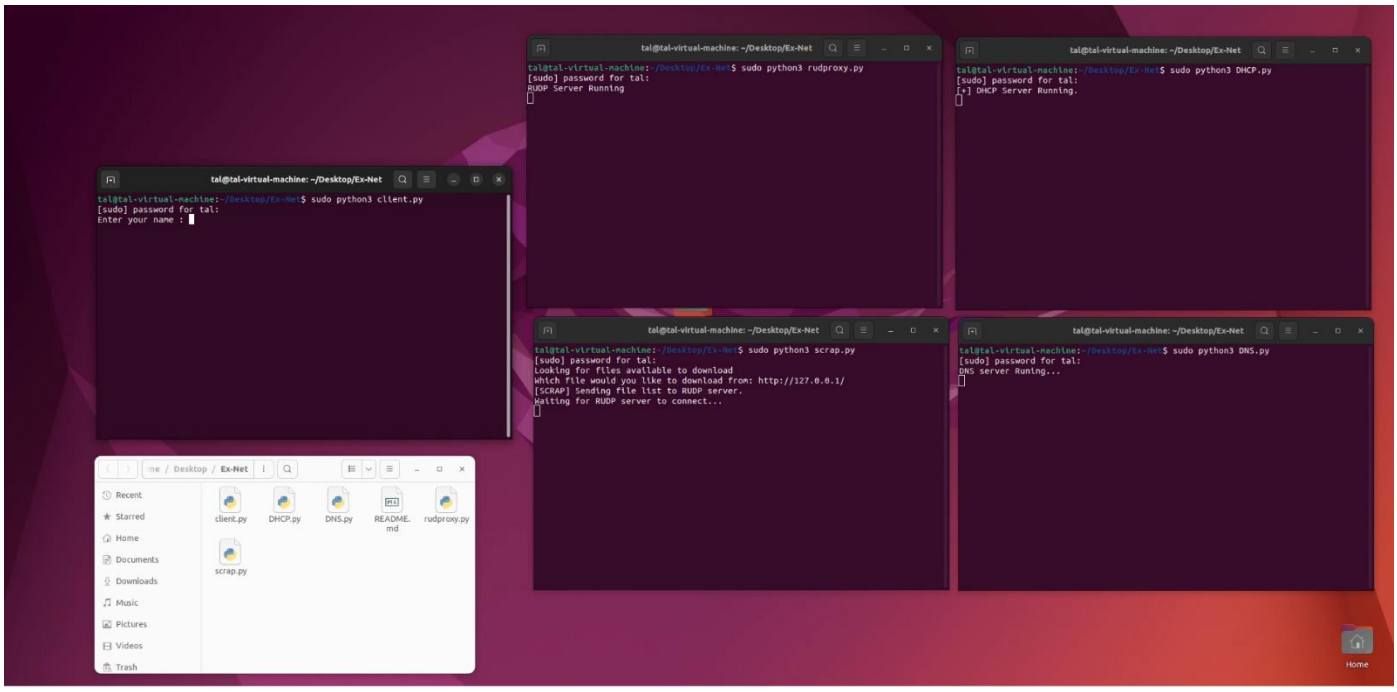
RUDP Block:

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

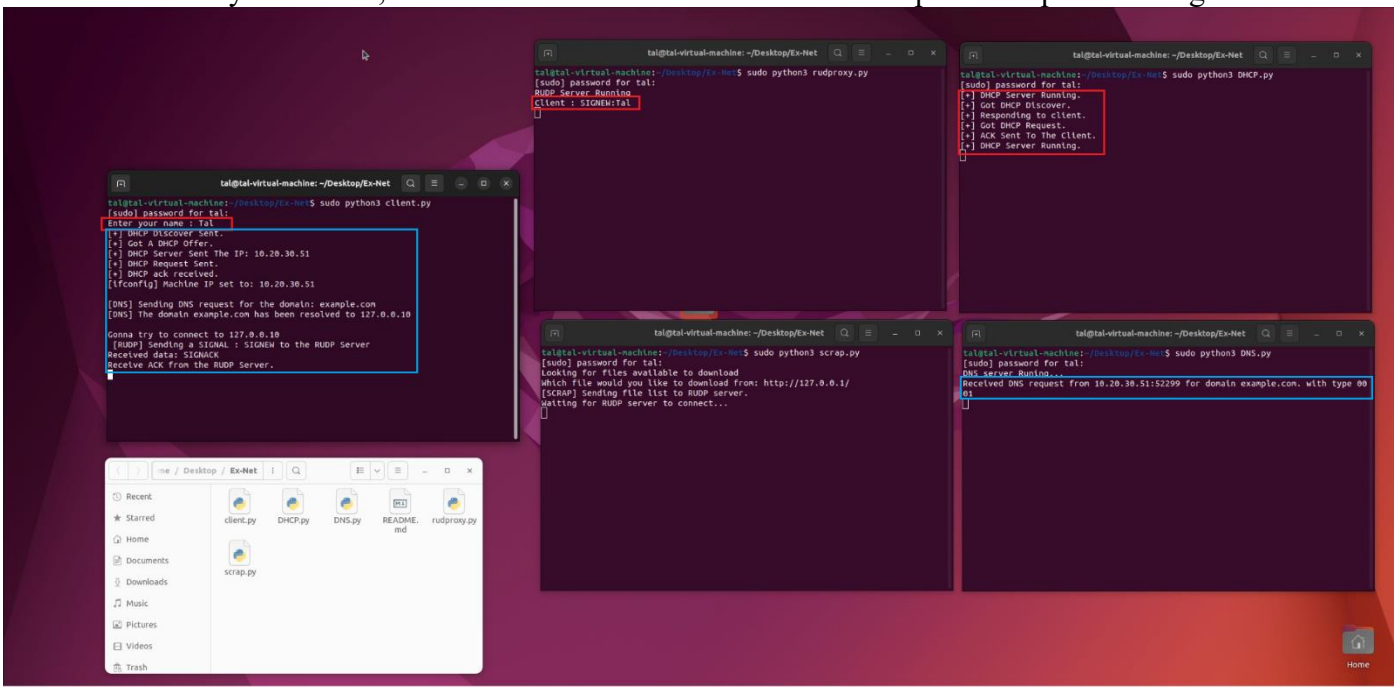
II- Running on the Terminal and explications :

Pictures & Process Explanation

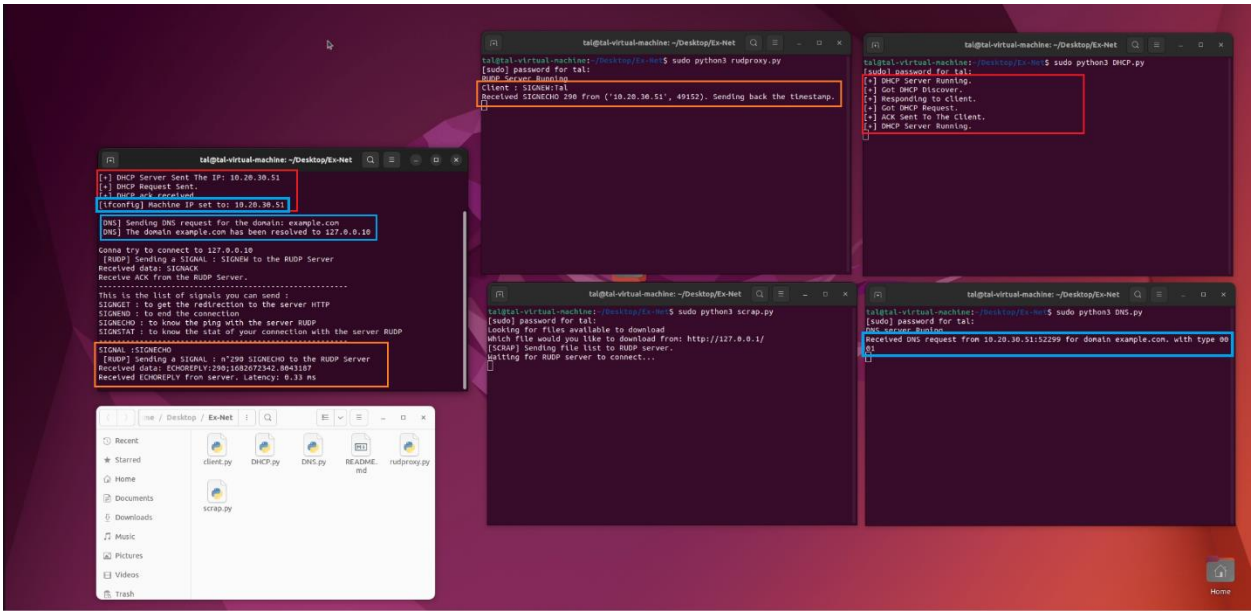
First, we would like to get all the terminals up and ready.



As you can see, The client terminal waits for the name input to keep on running.



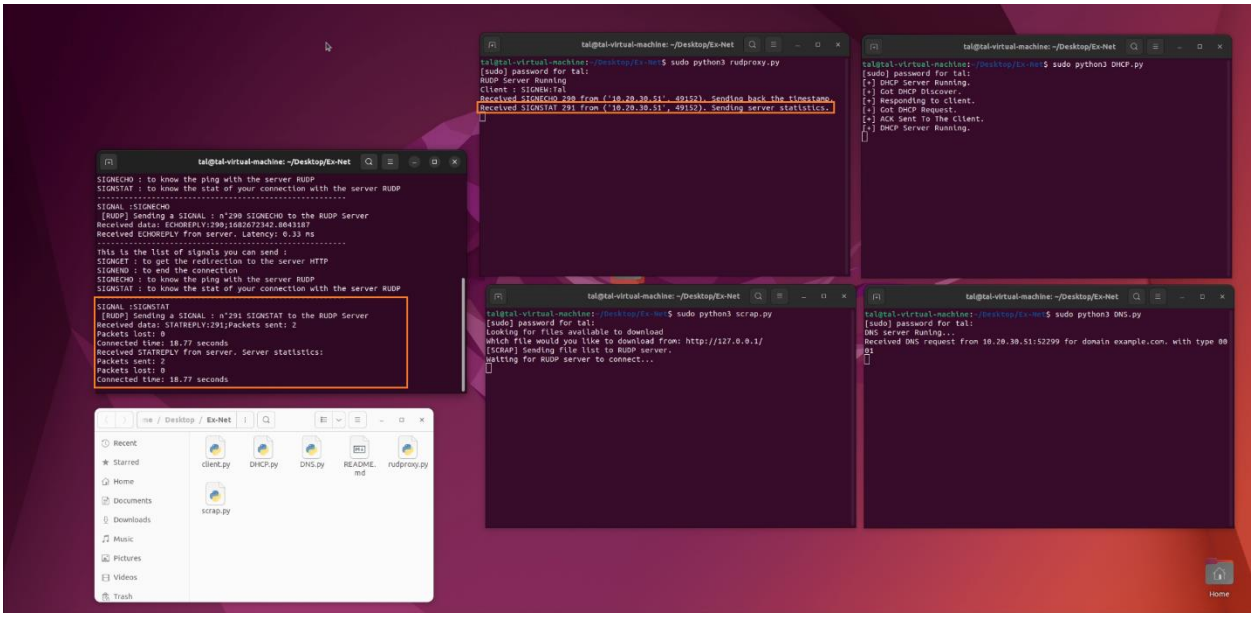
Right after the name input we can see a direct communication with the DHCP, RUDP and DNS Servers.



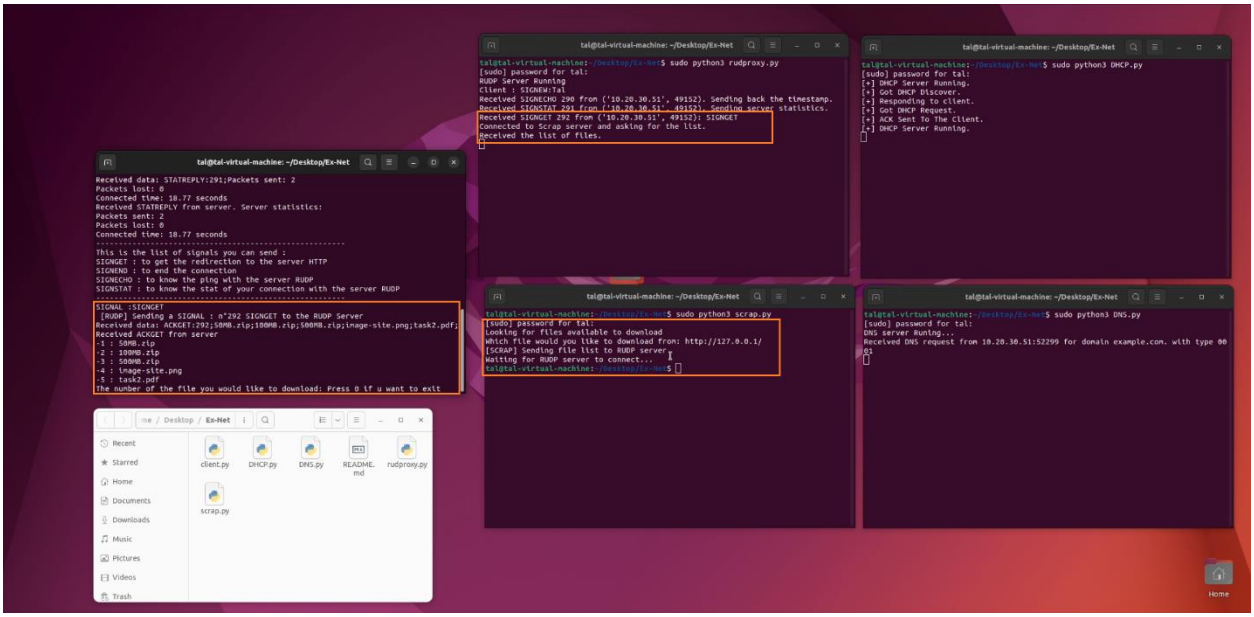
And that will be the full present the full picture a few moments after plus the information when the Client notifies the RUDP server to perform a latency test between them.

Plus as you can see the communication between the Client and the DNS under [ifconfig] when the Client is truly under the IP that the DHCP offered him.

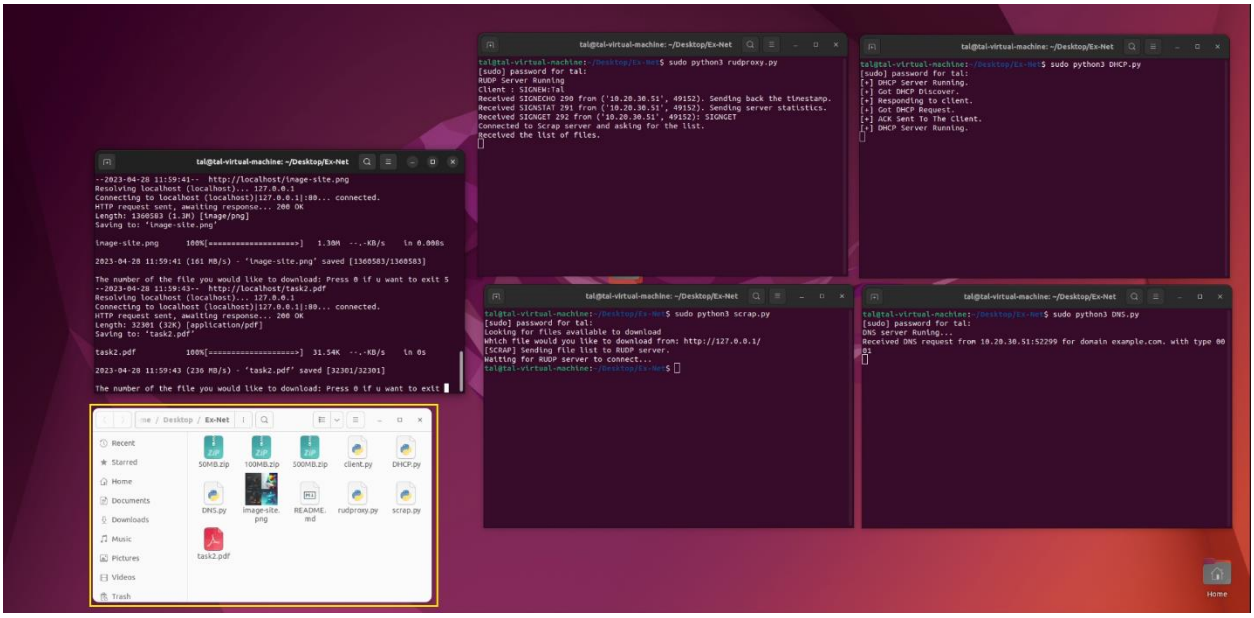
(DHCP, RUDP, DNS, ifconfig)



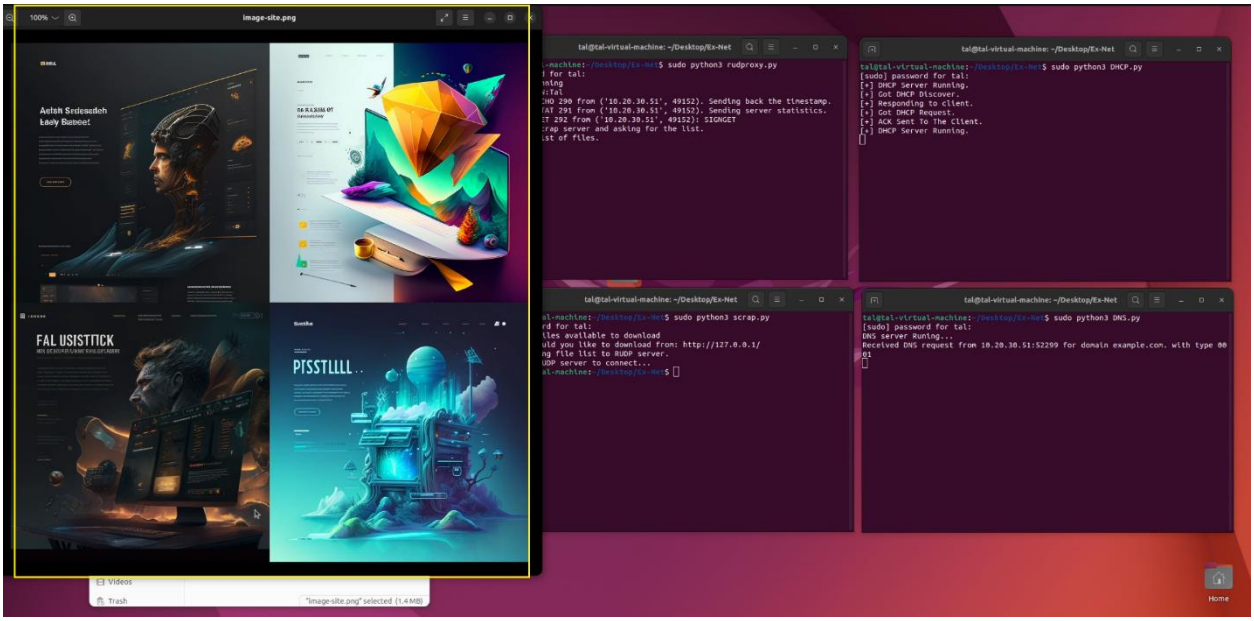
The Client notifies the RUDP server to perform a full connection test including packet loss information. (Orange)



In this case as we can see the Client notifies the RUDP Server to communicate with the Download server and to pull the list of downloadable files. (Orange)

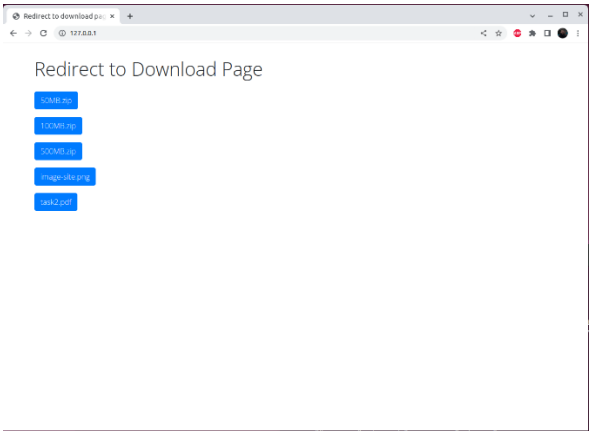


In this picture we are showing that the files have been downloaded successfully. (Yellow)



This picture will show you that the files are valid and not corrupted.

Thanks to Midjourney 😊.

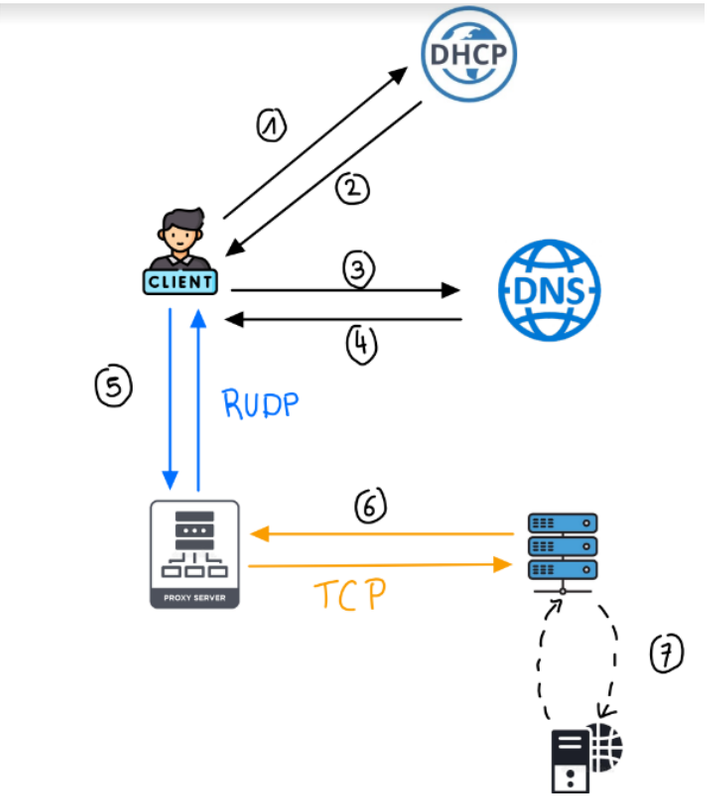


This is the website that holds the files and the actual page we used to scrap the data from.

We added a full YouTube video to showcase our project:

https://youtube.com/watch?v=is2kWKgyeng

Diagram



1) The client send a discover packet on the network and will send a request to the DHCP server
2) The DHCP server send an offer ip to the client and after the request from the client for this ip , the server will send him a ACK.
3) The client ask the DNS Server the ip from the domain that he need(queries).
4) The DNS send the ip to answer the querie of the client
5) The client start a new connection with the proxy Server with the protocol RUDP. He is able to send different signals like : SIGNGET , SIGNECHO , SIGNSTAT , SIGNEND. The proxy gonna respond to every request from the client.
6) The proxy and an other server (Scrap Server) will communicate with the protocol TCP. The Scrap Server going to send the list of different file on the Apachee Server when the client will send SIGNGET.
7) The Scrap Server going to write a list of the different downloadable file on the Apachee Server

Lost Packet in our Project :

When a packet is received, the server checks if the client's address is in the client_info dictionary. If the address is not present, it means that the client is new and the server sends an acknowledgment (ACK) with the "SIGNACK" signal. If the client already exists in client_info, the server checks the congestion window size for that client. When receiving a packet, the server checks if the packet is in the expected order by comparing its ID with that of last_received_id[addr]. If the packet is in order, the server processes the packet and sends an appropriate acknowledgment (ACK). If the packet is out of order, it means a packet has been lost. In this case, the server sends a "SIGNLOST" signal to the client with the expected packet ID using the send_lost_packet_signal function.

In our client code, if a packet is lost, the RUDP server will send a "SIGNLOST" signal to the client with the ID of the lost packet. The client handles this signal by calling the handle_lost_packet_signal function, which returns the lost packet to the RUDP server. The solution to handle packet loss in this case is to use this handle_lost_packet_signal function. This function retrieves the lost packet information from the sent_packets dictionary and sends the packet back to the RUDP server using the same signal that was originally sent. This allows the RUDP server to receive the lost packet and continue processing subsequent packets.

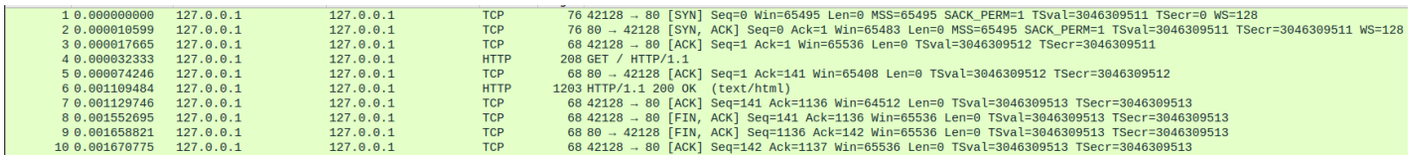
Congestion control in our Project :

In our proxy server code, it handles this through the congestion window (congestion_window) in the client_info dictionary for each client. When a new client connects, its congestion window is initialized to 1 ('congestion_window': 1). Each time a valid packet is received from a client, the congestion window is increased by 1 (client_info[addr]['congestion_window'] += 1). When an acknowledgment (ACK) is sent to the client, the congestion window is reduced by 1 (client_info[addr]['congestion_window'] -= 1). If the congestion window reaches the maximum size (max_window), a SIGNFULL signal is sent to the client (send_signfull_signal(server, addr)). This mechanism makes it possible to control congestion by adapting the congestion window according to the quantity of packets in transit. When the congestion window is small, it means that the server sends fewer packets, which reduces the load on the network and limits congestion. When the congestion window is large, it means that the server sends more packets, which can increase the load on the network and potentially cause congestion.

For the client: When it receives a "SIGNFULL" signal from the server, the client pauses sending packets for 10 seconds (time.sleep(10)) before resuming. This gives the server time to free up space in its receive window.

III- Explication of each packets from Wireshark :

Apachee Server :

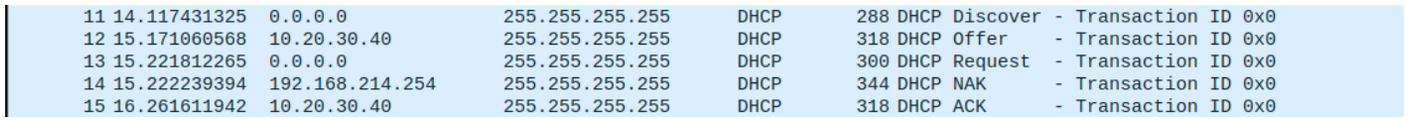


1	0.000000000	127.0.0.1	127.0.0.1	TCP	76	42128 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=3046309511 TSecr=0 WS=128
2	0.000010599	127.0.0.1	127.0.0.1	TCP	76	80 → 42128 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3046309511 TSecr=3046309511 WS=128
3	0.000017665	127.0.0.1	127.0.0.1	TCP	68	42128 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3046309512 TSecr=3046309511
4	0.000032333	127.0.0.1	127.0.0.1	HTTP	208	GET / HTTP/1.1
5	0.000074246	127.0.0.1	127.0.0.1	TCP	68	80 → 42128 [ACK] Seq=1 Ack=141 Win=65408 Len=0 TSval=3046309512 TSecr=3046309512
6	0.001109484	127.0.0.1	127.0.0.1	HTTP	1203	HTTP/1.1 200 OK (text/html)
7	0.001129746	127.0.0.1	127.0.0.1	TCP	68	42128 → 80 [ACK] Seq=141 Ack=1136 Win=64512 Len=0 TSval=3046309513 TSecr=3046309513
8	0.001552695	127.0.0.1	127.0.0.1	TCP	68	42128 → 80 [FIN, ACK] Seq=141 Ack=1136 Win=65536 Len=0 TSval=3046309513 TSecr=3046309513
9	0.001658821	127.0.0.1	127.0.0.1	TCP	68	80 → 42128 [FIN, ACK] Seq=1136 Ack=142 Win=65536 Len=0 TSval=3046309513 TSecr=3046309513
10	0.001670775	127.0.0.1	127.0.0.1	TCP	68	42128 → 80 [ACK] Seq=142 Ack=1137 Win=65536 Len=0 TSval=3046309513 TSecr=3046309513

In our code scrap.py , we accessed to the http server 127.0.0.1 to analyse the website and get the list of files.

- These packets 1 to 3 , it's a TCP 3-Way Handshake Process.
- The packet 4 : GET we are asking for the html code and all that makes up the site
- The packet 5 : ACK to respond that the server get our demand GET
- The packet 6 : This packet is the server response to the previous HTTP request. "HTTP/1.1" indicates the protocol version used, "200" is the response status code: means that the request was processed successfully. The content of the requested resource (for example, the HTML code for the home page) is also included in this package.
- The packet 7 : ACK to respond that we get the reponse from the server
- These packet 8 to 10 : To close the TCP connection

DHCP Server :

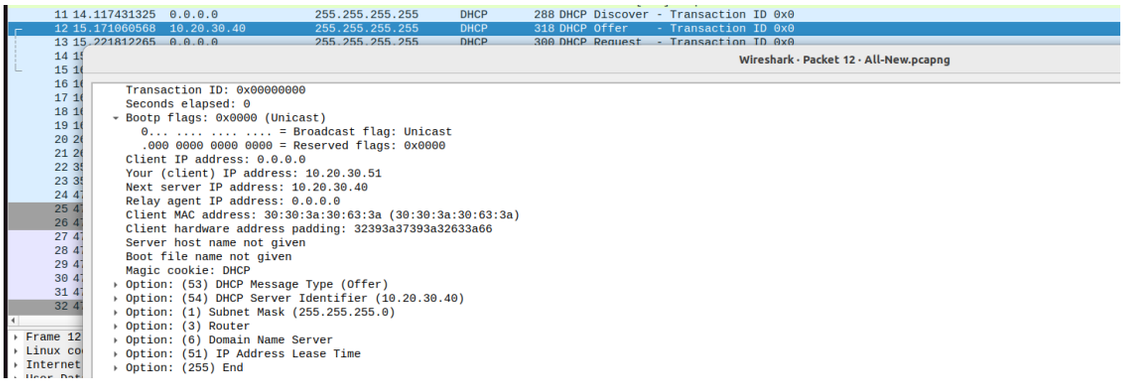


11	14.117431325	0.0.0.0	255.255.255.255	DHCP	288	DHCP Discover - Transaction ID 0x0
12	15.171060568	10.20.30.40	255.255.255.255	DHCP	318	DHCP Offer - Transaction ID 0x0
13	15.221812265	0.0.0.0	255.255.255.255	DHCP	300	DHCP Request - Transaction ID 0x0
14	15.222239394	192.168.214.254	255.255.255.255	DHCP	344	DHCP NAK - Transaction ID 0x0
15	16.261611942	10.20.30.40	255.255.255.255	DHCP	318	DHCP ACK - Transaction ID 0x0

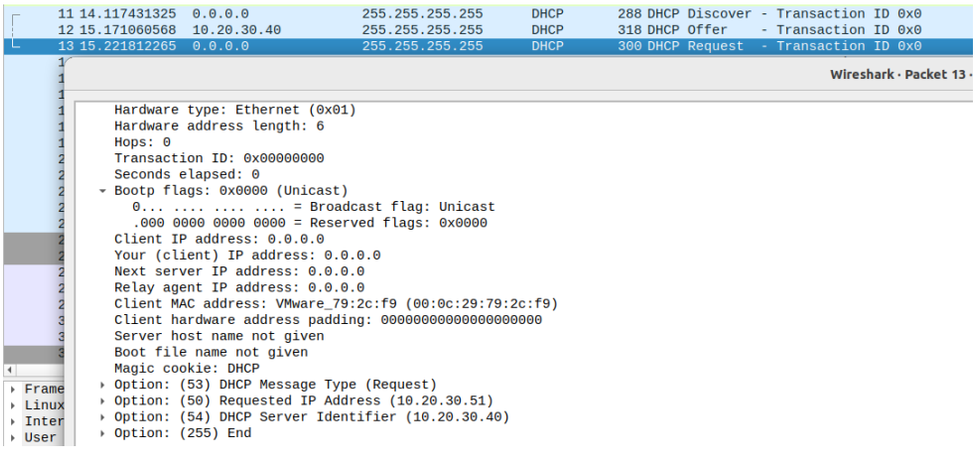
These packets represent the communication beetween the client and the DHCP server.

- The packet 11 : DHCP Discover: This packet is sent by the DHCP client to discover the DHCP servers available on the local network. The client essentially requests that any DHCP server respond with an IP address offer and other configuration information.

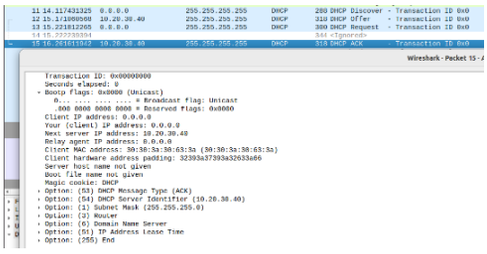
- The packet 12 : DHCP Offer: This packet is sent by a DHCP server in response to a DHCP Discover packet. It contains an IP address offer for the client, along with other configuration information, such as subnet mask, default gateway, and DNS servers.



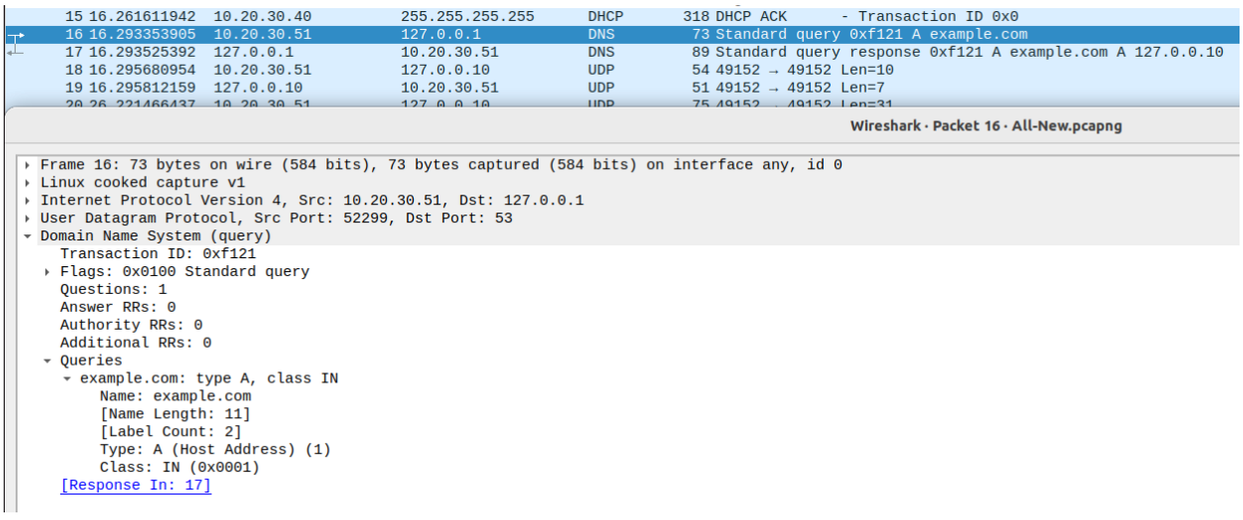
- This is the packet 12 : DHCP Offer , as we can see the DHCP Server (10.20.30.40) send the IP : 10.20.30.51 and also different option like Subnet Mask.
- The packet 13 : DHCP Request: This packet is sent by the DHCP client to accept the IP address offer and configuration information received from the DHCP server. It also informs other DHCP servers on the network that the client has accepted a particular server's offer, and that other offers (if any) are no longer needed. As we can see the request from the client , ask for the IP 10.20.30.51 to the Server 10.20.30.40.



- The packet 15 : DHCP ACK: This packet is sent by the DHCP server to confirm that the client has received the IP address offer and configuration information. This completes the IP address assignment process and the client can begin using the assigned IP address and configuration information to communicate on the network.



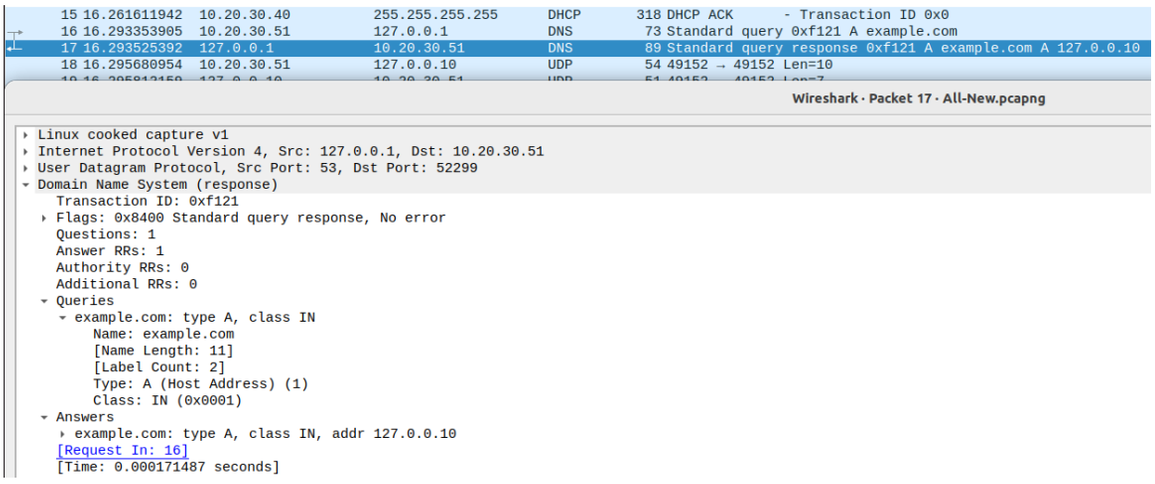
DNS Server :



The packet 16 and 17 is the communication between the client and the server DNS.

- Packet 16 : The client send a Querie to our DNS Server with the domain name "example.com" when the Type is : A. As we can see the reponse of the Querie is in Packet 17.

- Packet 17 : The DNS Server sent to the client the IP of the "example.com" Type A when the adress of this domain is : 127.0.0.10

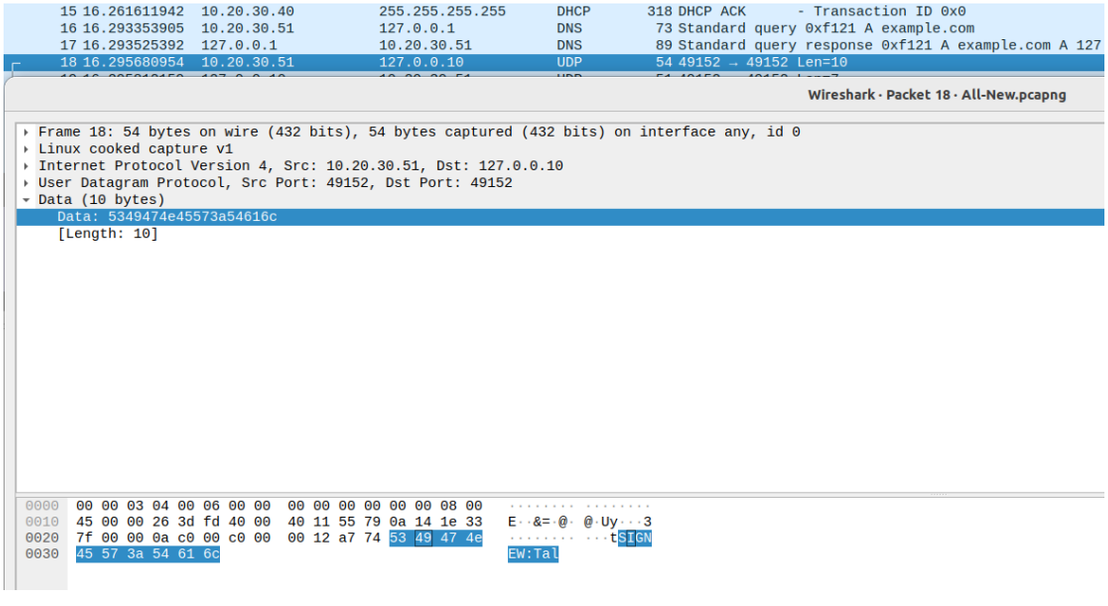


RUDP Connection : Client ⟷ Rudproxy

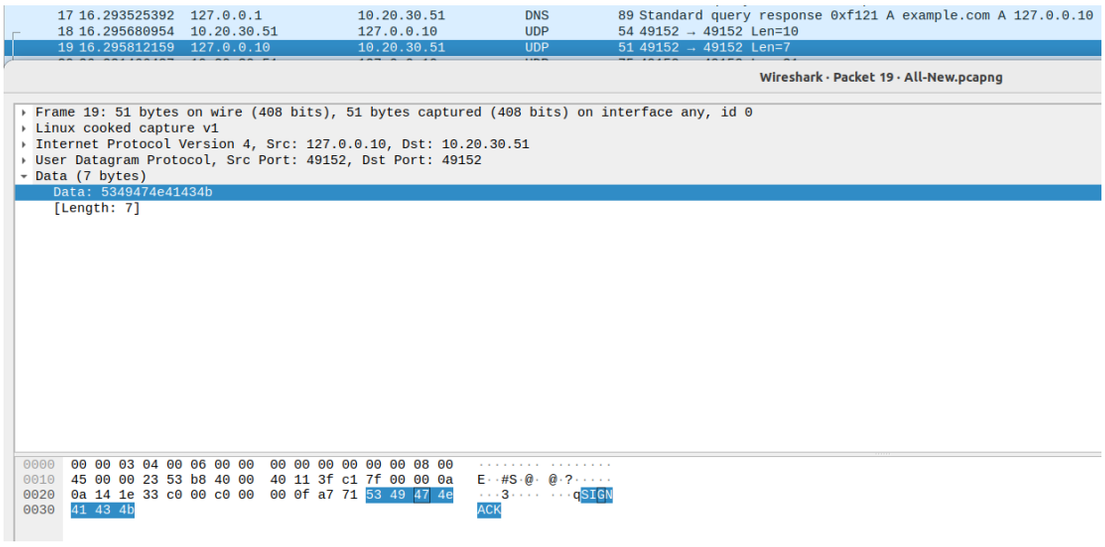
Now we are going to analyse the differents packets between the Client and the rudproxy. The communication going to be in RUDP.

Client : 10.20.30.51

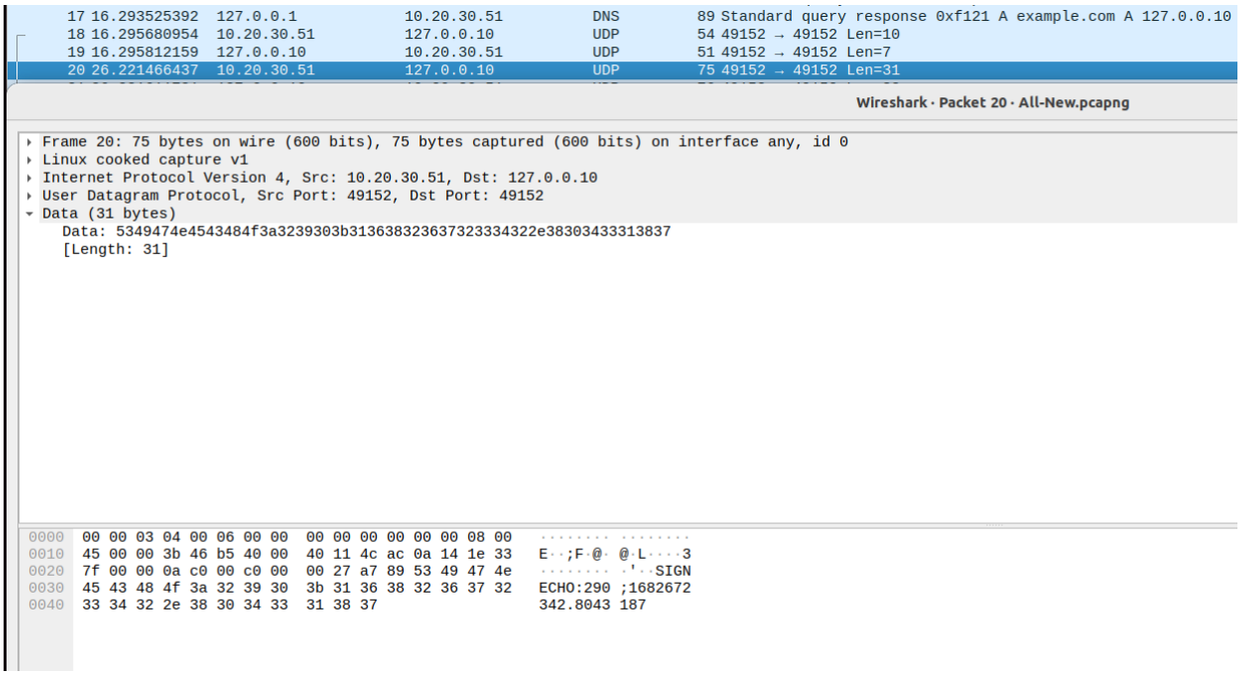
RUDP Server : 127.0.0.10



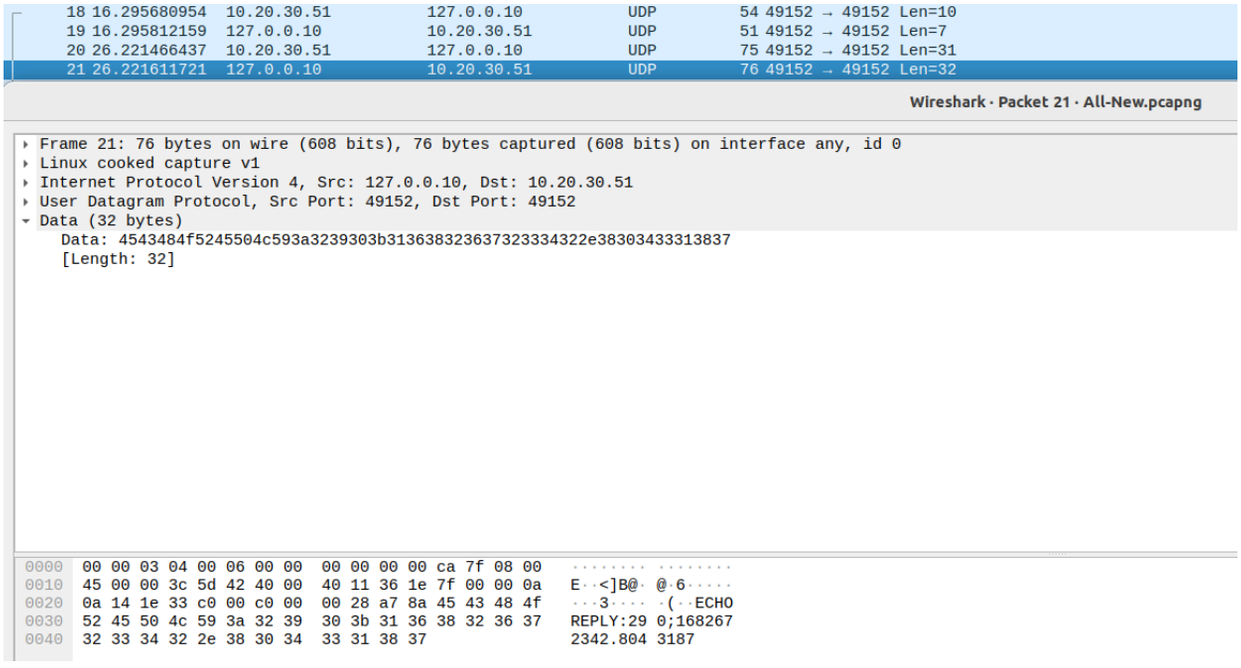
- The packet 18 : The client start a new connection with the Proxy Server. As we can see in the Data the client sent a SIGNEW + name of user to start the discusion.



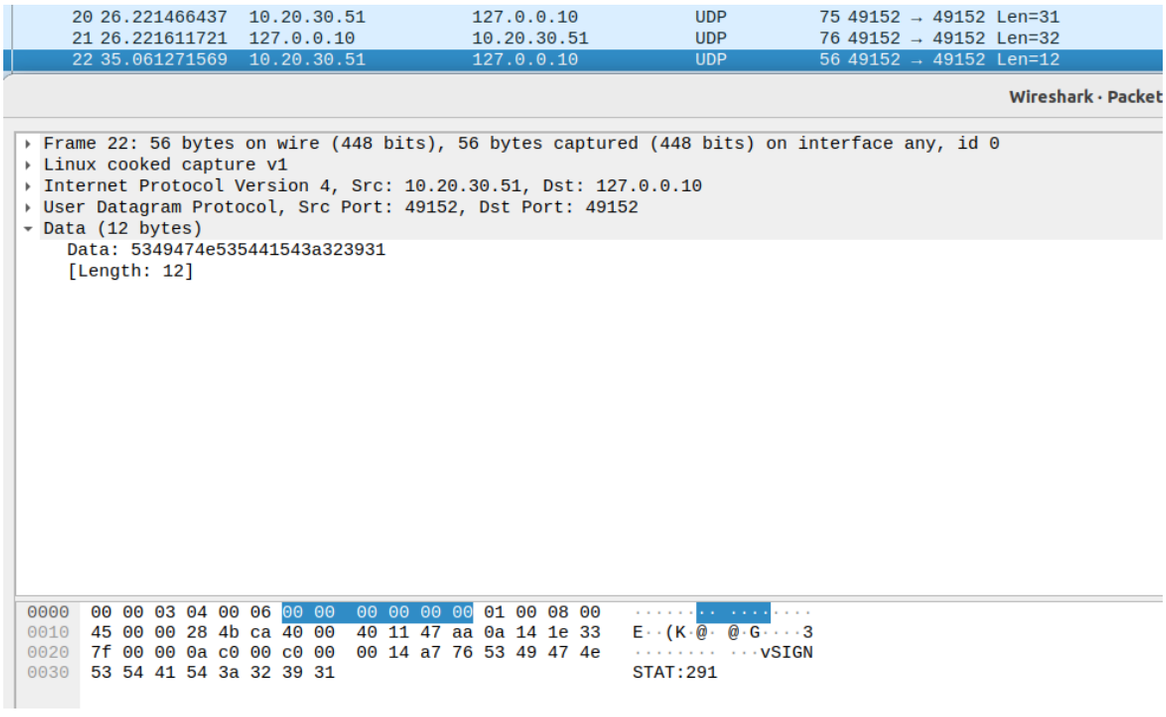
- The packet 19 : This is the response of the Server with the SIGNACK that he accept the connection with the Client.



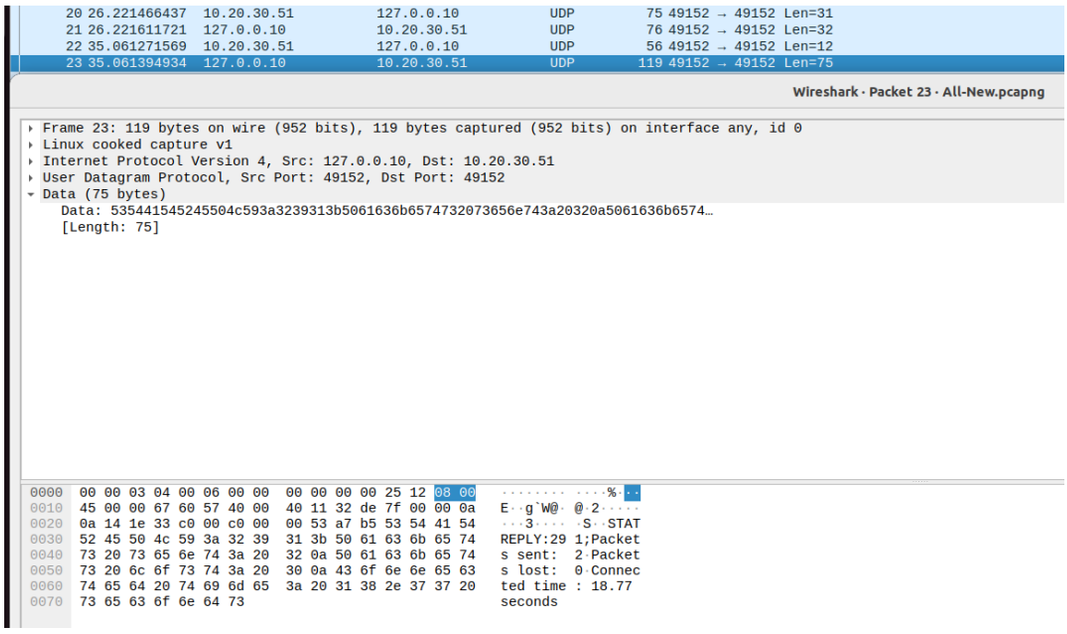
- Packet 20 : The client sent to the Server a new SIGNAL : SIGNECHO to know the ping between him and the Server. Packet ID : 290



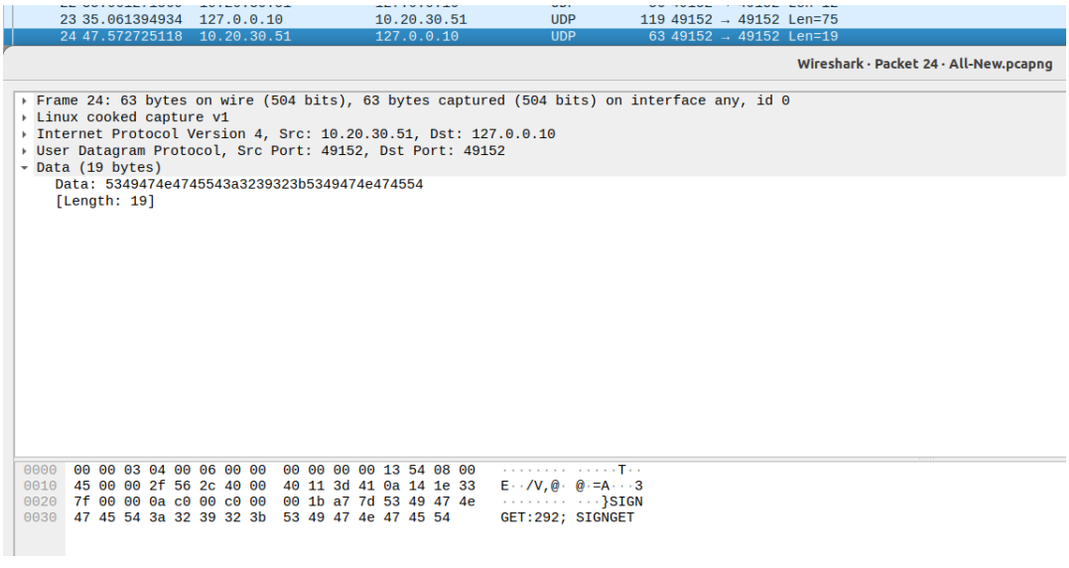
- The packet 21 : This is the response of the SIGNECHO , the server sent the time and also the SIGN : ECHOREPLY to the client.



- The packet 22 : The client sent to the Proxy Server a new SIGNAL : SIGNSTAT to know some information about the connection (packet sent , packet lost , time …). Packet ID : 291.

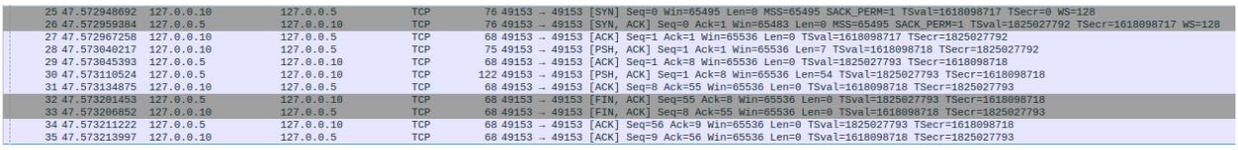


- The packet 23 : The Server sent a reponse after the SIGNSTAT with a the flag : STATREPLY , as we can see in the Data the server sent him different information.



- The packet 24 : The client sent to the server a new SIGNAL : SIGNGET to get the list of files that he can download. Packet ID : 292

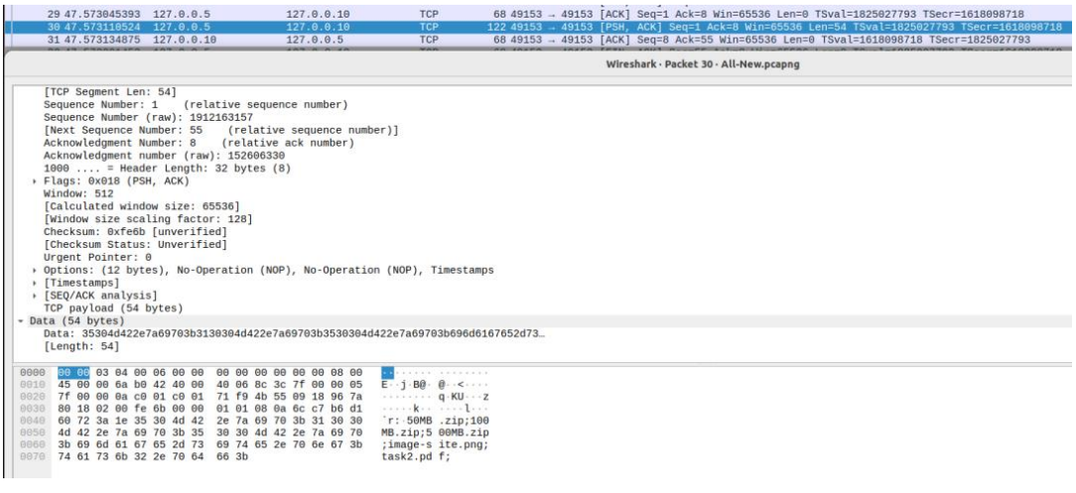
Connection TCP : Proxy Server ⟷ Scrap Server / Redirect



25	47.572948692	127.0.0.10	127.0.0.5	TCP	76 49153 → 49153 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1618098717 TSecr=0 WS=128
26	47.572959384	127.0.0.5	127.0.0.10	TCP	76 49153 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=1825027792 TSecr=1618098717 WS=128
27	47.572967258	127.0.0.10	127.0.0.5	TCP	68 49153 → 49153 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1618098717 TSecr=1825027792
28	47.573040217	127.0.0.10	127.0.0.5	TCP	75 49153 → 49153 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=7 TSval=1618098718 TSecr=1825027792
29	47.573045393	127.0.0.5	127.0.0.10	TCP	68 49153 → 49153 [ACK] Seq=1 Ack=8 Win=65536 Len=0 TSval=1825027793 TSecr=1618098718
30	47.573110524	127.0.0.5	127.0.0.10	TCP	122 49153 → 49153 [PSH, ACK] Seq=1 Ack=8 Win=65536 Len=54 TSval=1825027793 TSecr=1618098718
31	47.573134875	127.0.0.10	127.0.0.5	TCP	68 49153 → 49153 [ACK] Seq=8 Ack=55 Win=65536 Len=0 TSval=1618098718 TSecr=1825027793
32	47.573201453	127.0.0.5	127.0.0.10	TCP	68 49153 → 49153 [FIN, ACK] Seq=55 Ack=8 Win=65536 Len=0 TSval=1825027793 TSecr=1618098718
33	47.573206852	127.0.0.10	127.0.0.5	TCP	68 49153 → 49153 [FIN, ACK] Seq=8 Ack=55 Win=65536 Len=0 TSval=1618098718 TSecr=1825027793
34	47.573211222	127.0.0.5	127.0.0.10	TCP	68 49153 → 49153 [ACK] Seq=56 Ack=9 Win=65536 Len=0 TSval=1825027793 TSecr=1618098718
35	47.573213997	127.0.0.10	127.0.0.5	TCP	68 49153 → 49153 [ACK] Seq=9 Ack=56 Win=65536 Len=0 TSval=1618098718 TSecr=1825027793

The client ask the list of differents files that he can download.The proxy Server perform a redirect to the Scrap Server with a connection TCP.

- The packets 25 to 27 : it's a TCP 3-Way Handshake Process. As we can see the connection it's between 127.0.0.10 and 127.0.0.5 (Proxy Server and Scrap Server)
- The packet 28 : [PSH,ACK] 127.0.0.10 → 127.0.0.5 The Proxy Server sent the request to the Scrap Server to get the list of files.
- The packet 29 : The scrap Server sent to the Proxy an ACK to assure to him that he get his request. We can see the list in Data (54 bytes)
- The packet 30 : [PSH,ACK] 127.0.0.5 → 127.0.0.10 The Scrap Server sent the list of files to the Proxy Server.

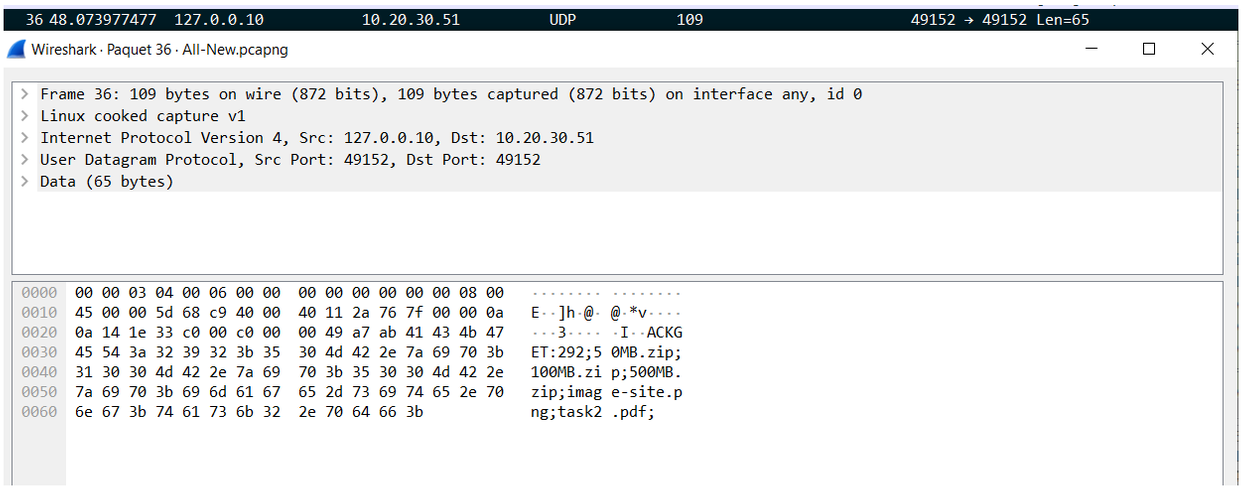


```
29 47.573045393  127.0.0.5          127.0.0.10          TCP      68 49153 → 49153 [ACK] Seq=1 Ack=8 Win=65536 Len=0 TSval=1825027793 TSecr=1618098718
30 47.573110524  127.0.0.5          127.0.0.10          TCP     122 49153 → 49153 [PSH, ACK] Seq=1 Ack=8 Win=65536 Len=54 TSval=1825027793 TSecr=1618098718
31 47.573134875  127.0.0.10         127.0.0.5           TCP      68 49153 → 49153 [ACK] Seq=8 Ack=55 Win=65536 Len=0 TSval=1618098718 TSecr=1825027793

                                  Wireshark · Packet 30 · All-New.pcapng

    [TCP Segment Len: 54]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 1912163157
    [Next Sequence Number: 55    (relative sequence number)]
    Acknowledgment Number: 8    (relative ack number)
    Acknowledgment number (raw): 152606330
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 512
    [Calculated window size: 65536]
    [Window size scaling factor: 128]
    Checksum: 0xfe6b [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (54 bytes)
▾ Data (54 bytes)
    Data: 35304d422e7a69703b3130304d422e7a69703b3530304d422e7a69703b696d6167652d73…
    [Length: 54]

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ··· ····· ·····
0010  45 00 00 6a b0 42 40 00  40 06 8c 3c 7f 00 00 05   E··j·B@· @·<····
0020  7f 00 00 0a c0 01 c0 01  71 f9 4b 55 09 18 96 7a   ·····q·KU···z
0030  80 18 02 00 fe 6b 00 00  01 01 08 0a 6c c7 b6 d1   ·····k·· ····l···
0040  60 72 3a 1e 35 30 4d 42  2e 7a 69 70 3b 31 30 30   `r:·50MB .zip;100
0050  4d 42 2e 7a 69 70 3b 35  30 30 4d 42 2e 7a 69 70   MB.zip;5 00MB.zip
0060  3b 69 6d 61 67 65 2d 73  69 74 65 2e 70 6e 67 3b   ;image-s ite.png;
0070  74 61 73 6b 32 2e 70 64  66 3b                     task2.pd f;
```

- The packet 31 : The proxy Server sent an ACK to the Scrap Server to assure that he got the list of files from him.

- The packets 32 to 35 : To close the connection between the Proxy Server and Scrap Server.

Now the Proxy Server going to send the list to the Client.



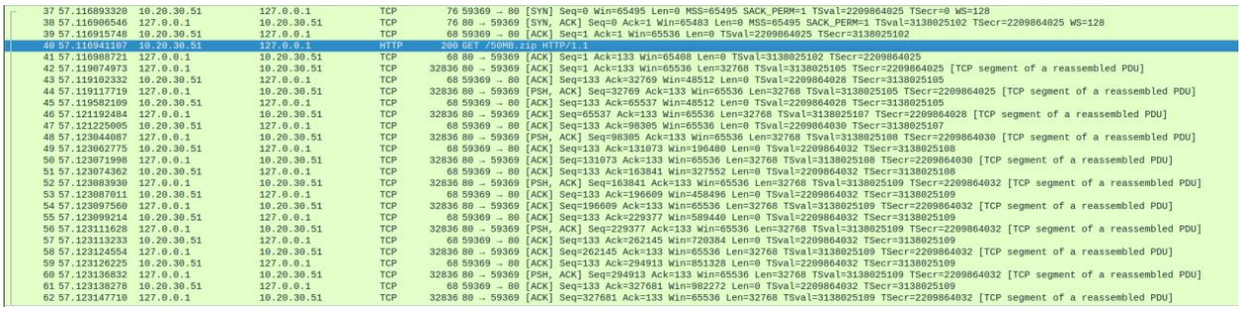
The packet 36 : 127.0.0.10 → 10.20.30.51 , the Proxy Server sent now the list of files to the Client. He used the SIGNAL : ACKGET and as we can see in the Data (65 bytes) the differents file to download.

The list is :

1. 50MB.zip
2. 100MB.zip
3. 500MB.zip
4. Image-site.png
5. Task2.pdf

For the explication the client going to download every files.

The first file : 50MB.zip



```
37 57.116893320  10.20.30.51   127.0.0.1    TCP      76 59369 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2209864025 TSecr=0 WS=128
38 57.116906546  127.0.0.1    10.20.30.51  TCP      76 80 → 59369 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3138025102 TSecr=2209864025 WS=128
39 57.116915748  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2209864025 TSecr=3138025102
40 57.116941197  10.20.30.51   127.0.0.1    HTTP     200 GET /50MB.zip HTTP/1.1
41 57.116988721  127.0.0.1    10.20.30.51  TCP      68 80 → 59369 [ACK] Seq=1 Ack=133 Win=65408 Len=0 TSval=3138025102 TSecr=2209864025
42 57.119074973  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=1 Ack=133 Win=65536 Len=32768 TSval=3138025105 TSecr=2209864025 [TCP segment of a reassembled PDU]
43 57.119102332  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=32769 Win=48512 Len=0 TSval=2209864028 TSecr=3138025105
44 57.119117719  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [PSH, ACK] Seq=32769 Ack=133 Win=65536 Len=32768 TSval=3138025105 TSecr=2209864025 [TCP segment of a reassembled PDU]
45 57.119582189  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=65537 Win=48512 Len=0 TSval=2209864028 TSecr=3138025105
46 57.121192484  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=65537 Ack=133 Win=65536 Len=32768 TSval=3138025107 TSecr=2209864028 [TCP segment of a reassembled PDU]
47 57.121225005  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=98305 Win=65536 Len=0 TSval=2209864030 TSecr=3138025107
48 57.123044087  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [PSH, ACK] Seq=98305 Ack=133 Win=65536 Len=32768 TSval=3138025108 TSecr=2209864030 [TCP segment of a reassembled PDU]
49 57.123062775  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=131073 Win=196480 Len=0 TSval=2209864032 TSecr=3138025108
50 57.123071998  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=131073 Ack=133 Win=65536 Len=32768 TSval=3138025108 TSecr=2209864030 [TCP segment of a reassembled PDU]
51 57.123074362  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=163841 Win=327552 Len=0 TSval=2209864032 TSecr=3138025108
52 57.123083930  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [PSH, ACK] Seq=163841 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
53 57.123087011  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=196609 Win=458496 Len=0 TSval=2209864032 TSecr=3138025109
54 57.123097560  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=196609 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
55 57.123099214  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=229377 Win=589440 Len=0 TSval=2209864032 TSecr=3138025109
56 57.123111628  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [PSH, ACK] Seq=229377 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
57 57.123113233  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=262145 Win=720384 Len=0 TSval=2209864032 TSecr=3138025109
58 57.123124554  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=262145 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
59 57.123126225  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=294913 Win=851328 Len=0 TSval=2209864032 TSecr=3138025109
60 57.123136832  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [PSH, ACK] Seq=294913 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
61 57.123138278  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [ACK] Seq=133 Ack=327681 Win=982272 Len=0 TSval=2209864032 TSecr=3138025109
62 57.123147710  127.0.0.1    10.20.30.51  TCP    32836 80 → 59369 [ACK] Seq=327681 Ack=133 Win=65536 Len=32768 TSval=3138025109 TSecr=2209864032 [TCP segment of a reassembled PDU]
```

- The packet 37 to 39 : it's a TCP 3-Way Handshake Process.

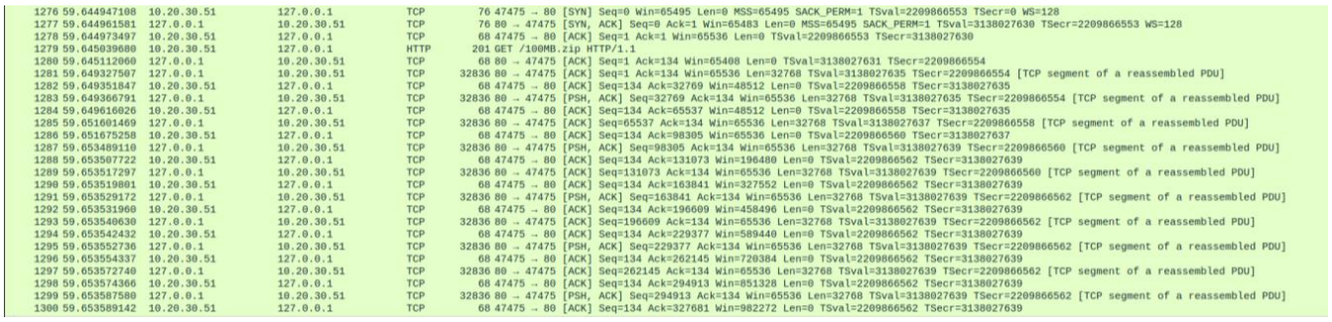
- The packet 40 : [200 GET /50MB.zip HTTP/1.1] The client asking the file : 50MB.zip .

- The next packets going to be the download of this file , as we can see many packets of [ACK] and [PSH,ACK].

```
1273 57.290518245  10.20.30.51   127.0.0.1    TCP      68 59369 → 80 [FIN, ACK] Seq=133 Ack=52429101 Win=3112448 Len=0 TSval=2209864199 TSecr=3138025275
1274 57.290733825  127.0.0.1    10.20.30.51  TCP      68 80 → 59369 [FIN, ACK] Seq=52429101 Ack=134 Win=65536 Len=0 TSval=3138025276 TSecr=2209864199
```

- The packet 1273 and 1274 : The client got the file 50MB.zip , so now he can close the connection with the flag [FIN, ACK].

The second file : 100MB.zip

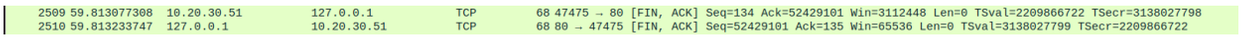


```
1276 59.644947108  10.20.30.51   127.0.0.1     TCP      76 47475 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2209866553 TSecr=0 WS=128
1277 59.644961581  127.0.0.1     10.20.30.51   TCP      76 80 → 47475 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3138027630 TSecr=2209866553 WS=128
1278 59.644973497  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2209866553 TSecr=3138027630
1279 59.645039680  10.20.30.51   127.0.0.1     HTTP    201 GET /100MB.zip HTTP/1.1
1280 59.645112060  127.0.0.1     10.20.30.51   TCP      68 80 → 47475 [ACK] Seq=1 Ack=134 Win=65408 Len=0 TSval=3138027631 TSecr=2209866554
1281 59.649327507  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [ACK] Seq=1 Ack=134 Win=65536 Len=32768 TSval=3138027635 TSecr=2209866554 [TCP segment of a reassembled PDU]
1282 59.649351847  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=32769 Win=48512 Len=0 TSval=2209866558 TSecr=3138027635
1283 59.649366791  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [PSH, ACK] Seq=32769 Ack=134 Win=65536 Len=32768 TSval=3138027635 TSecr=2209866554 [TCP segment of a reassembled PDU]
1284 59.649616026  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=65537 Win=48512 Len=0 TSval=2209866558 TSecr=3138027635
1285 59.651601469  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [ACK] Seq=65537 Ack=134 Win=65536 Len=32768 TSval=3138027637 TSecr=2209866558 [TCP segment of a reassembled PDU]
1286 59.651675258  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=98305 Win=65536 Len=0 TSval=2209866560 TSecr=3138027637
1287 59.653489110  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [PSH, ACK] Seq=98305 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866560 [TCP segment of a reassembled PDU]
1288 59.653507722  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=131073 Win=196480 Len=0 TSval=2209866562 TSecr=3138027639
1289 59.653517297  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [ACK] Seq=131073 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866560 [TCP segment of a reassembled PDU]
1290 59.653519801  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=163841 Win=327552 Len=0 TSval=2209866562 TSecr=3138027639
1291 59.653529172  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [PSH, ACK] Seq=163841 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866562 [TCP segment of a reassembled PDU]
1292 59.653531960  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=196609 Win=458496 Len=0 TSval=2209866562 TSecr=3138027639
1293 59.653540630  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [ACK] Seq=196609 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866562 [TCP segment of a reassembled PDU]
1294 59.653542432  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=229377 Win=589440 Len=0 TSval=2209866562 TSecr=3138027639
1295 59.653552736  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [PSH, ACK] Seq=229377 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866562 [TCP segment of a reassembled PDU]
1296 59.653554337  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=262145 Win=720384 Len=0 TSval=2209866562 TSecr=3138027639
1297 59.653572740  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [ACK] Seq=262145 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866562 [TCP segment of a reassembled PDU]
1298 59.653574366  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=294913 Win=851328 Len=0 TSval=2209866562 TSecr=3138027639
1299 59.653587580  127.0.0.1     10.20.30.51   TCP   32836 80 → 47475 [PSH, ACK] Seq=294913 Ack=134 Win=65536 Len=32768 TSval=3138027639 TSecr=2209866562 [TCP segment of a reassembled PDU]
1300 59.653589142  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [ACK] Seq=134 Ack=327681 Win=982272 Len=0 TSval=2209866562 TSecr=3138027639
```

- The packets 1276 to 1278 : it's a TCP 3-Way Handshake Process.

- The packet 1279 : [200 GET /100MB.zip HTTP/1.1] The client asking the file : 100MB.zip .

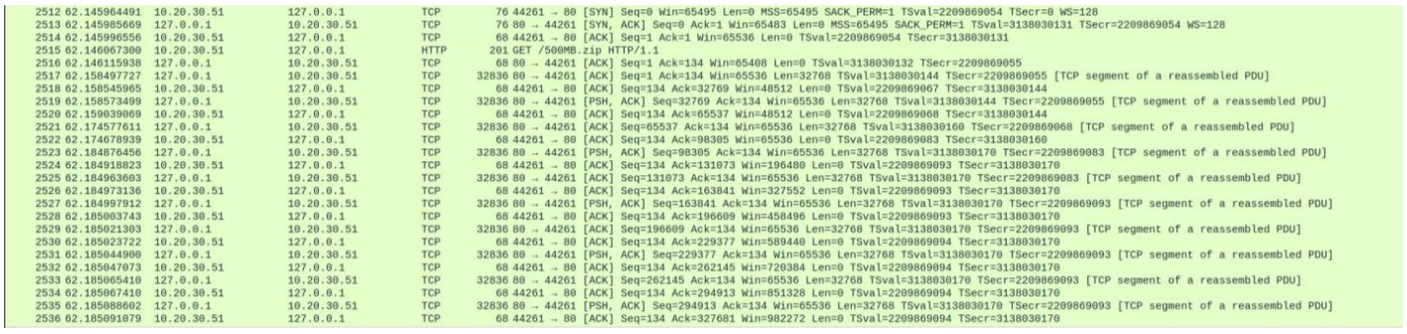
- The next packets going to be the download of this file , as we can see many packets of [ACK] and [PSH,ACK].



```
2509 59.813077308  10.20.30.51   127.0.0.1     TCP      68 47475 → 80 [FIN, ACK] Seq=134 Ack=52429101 Win=3112448 Len=0 TSval=2209866722 TSecr=3138027798
2510 59.813233747  127.0.0.1     10.20.30.51   TCP      68 80 → 47475 [FIN, ACK] Seq=52429101 Ack=135 Win=65536 Len=0 TSval=3138027799 TSecr=2209866722
```

- The packet 2509 and 2510 : The client got the file 100MB.zip , so now he can close the connection with the flag [FIN, ACK].

The third file : 500MB.zip

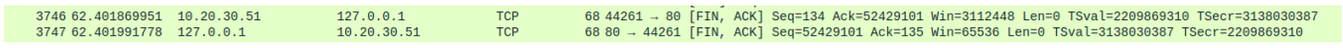


```
2512 62.145964491  10.20.30.51   127.0.0.1     TCP      76 44261 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2209869054 TSecr=0 WS=128
2513 62.145985669  127.0.0.1     10.20.30.51   TCP      76 80 → 44261 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3138030131 TSecr=2209869054 WS=128
2514 62.145996556  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2209869054 TSecr=3138030131
2515 62.146067300  10.20.30.51   127.0.0.1     HTTP    201 GET /500MB.zip HTTP/1.1
2516 62.146115938  127.0.0.1     10.20.30.51   TCP      68 80 → 44261 [ACK] Seq=1 Ack=134 Win=65408 Len=0 TSval=3138030132 TSecr=2209869055
2517 62.158497727  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [ACK] Seq=1 Ack=134 Win=65536 Len=32768 TSval=3138030144 TSecr=2209869055 [TCP segment of a reassembled PDU]
2518 62.158545965  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=32769 Win=48512 Len=0 TSval=2209869067 TSecr=3138030144
2519 62.158573499  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [PSH, ACK] Seq=32769 Ack=134 Win=65536 Len=32768 TSval=3138030144 TSecr=2209869055 [TCP segment of a reassembled PDU]
2520 62.159039069  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=65537 Win=48512 Len=0 TSval=2209869068 TSecr=3138030144
2521 62.174577611  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [ACK] Seq=65537 Ack=134 Win=65536 Len=32768 TSval=3138030160 TSecr=2209869068 [TCP segment of a reassembled PDU]
2522 62.174678939  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=98305 Win=65536 Len=0 TSval=2209869083 TSecr=3138030160
2523 62.184876456  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [PSH, ACK] Seq=98305 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869083 [TCP segment of a reassembled PDU]
2524 62.184918823  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=131073 Win=196480 Len=0 TSval=2209869093 TSecr=3138030170
2525 62.184963603  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [ACK] Seq=131073 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869083 [TCP segment of a reassembled PDU]
2526 62.184973136  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=163841 Win=327552 Len=0 TSval=2209869093 TSecr=3138030170
2527 62.184997912  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [PSH, ACK] Seq=163841 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869093 [TCP segment of a reassembled PDU]
2528 62.185003743  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=196609 Win=458496 Len=0 TSval=2209869093 TSecr=3138030170
2529 62.185021303  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [ACK] Seq=196609 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869093 [TCP segment of a reassembled PDU]
2530 62.185023722  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=229377 Win=589440 Len=0 TSval=2209869094 TSecr=3138030170
2531 62.185044900  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [PSH, ACK] Seq=229377 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869093 [TCP segment of a reassembled PDU]
2532 62.185047073  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=262145 Win=720384 Len=0 TSval=2209869094 TSecr=3138030170
2533 62.185065410  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [ACK] Seq=262145 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869093 [TCP segment of a reassembled PDU]
2534 62.185067410  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=294913 Win=851328 Len=0 TSval=2209869094 TSecr=3138030170
2535 62.185088602  127.0.0.1     10.20.30.51   TCP   32836 80 → 44261 [PSH, ACK] Seq=294913 Ack=134 Win=65536 Len=32768 TSval=3138030170 TSecr=2209869093 [TCP segment of a reassembled PDU]
2536 62.185091079  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [ACK] Seq=134 Ack=327681 Win=982272 Len=0 TSval=2209869094 TSecr=3138030170
```

- The packets 2512 to 2514 : it's a TCP 3-Way Handshake Process.

- The packet 2515 : [200 GET /500MB.zip HTTP/1.1] The client asking the file : 500MB.zip .

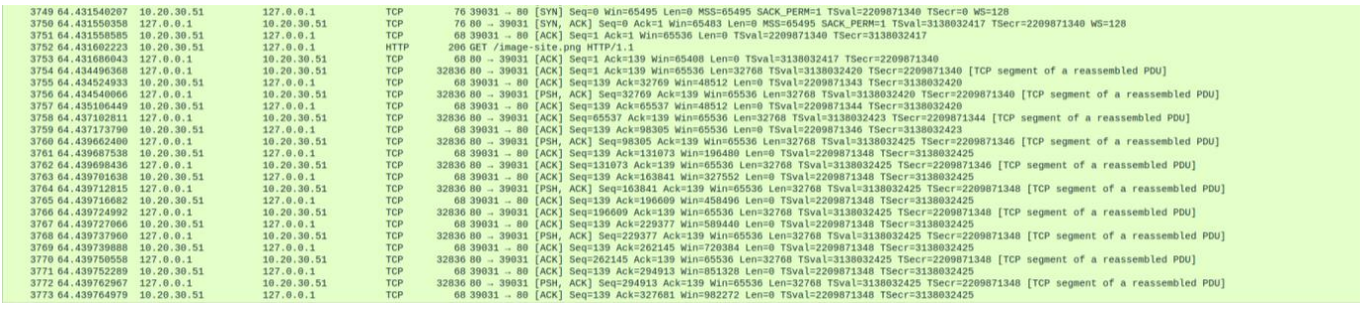
- The next packets going to be the download of this file , as we can see many packets of [ACK] and [PSH,ACK].



```
3746 62.401869951  10.20.30.51   127.0.0.1     TCP      68 44261 → 80 [FIN, ACK] Seq=134 Ack=52429101 Win=3112448 Len=0 TSval=2209869310 TSecr=3138030387
3747 62.401991778  127.0.0.1     10.20.30.51   TCP      68 80 → 44261 [FIN, ACK] Seq=52429101 Ack=135 Win=65536 Len=0 TSval=3138030387 TSecr=2209869310
```

- The packet 3746 and 3747 : The client got the file 500MB.zip , so now he can close the connection with the flag [FIN, ACK].

The fourth file : image-site.png

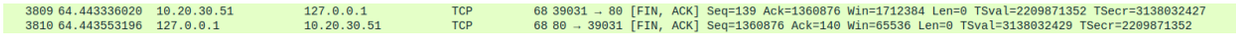


```
3749 64.431540207  10.20.30.51    127.0.0.1     TCP      76 39031 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2209871340 TSecr=0 WS=128
3750 64.431550358  127.0.0.1     10.20.30.51   TCP      76 80 → 39031 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3138032417 TSecr=2209871340 WS=128
3751 64.431558585  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2209871340 TSecr=3138032417
3752 64.431602223  10.20.30.51   127.0.0.1     HTTP    206 GET /image-site.png HTTP/1.1
3753 64.431606043  127.0.0.1     10.20.30.51   TCP      68 80 → 39031 [ACK] Seq=1 Ack=139 Win=65408 Len=0 TSval=3138032417 TSecr=2209871340
3754 64.434496368  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [ACK] Seq=1 Ack=139 Win=65536 Len=32768 TSval=3138032420 TSecr=2209871340 [TCP segment of a reassembled PDU]
3755 64.434524933  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=32769 Win=48512 Len=0 TSval=2209871343 TSecr=3138032420
3756 64.434540066  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [PSH, ACK] Seq=32769 Ack=139 Win=65536 Len=32768 TSval=3138032420 TSecr=2209871340 [TCP segment of a reassembled PDU]
3757 64.435106449  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=65537 Win=48512 Len=0 TSval=2209871344 TSecr=3138032420
3758 64.437102811  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [ACK] Seq=65537 Ack=139 Win=65536 Len=32768 TSval=3138032423 TSecr=2209871344 [TCP segment of a reassembled PDU]
3759 64.437173790  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=98305 Win=65536 Len=0 TSval=2209871346 TSecr=3138032423
3760 64.439662400  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [PSH, ACK] Seq=98305 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871346 [TCP segment of a reassembled PDU]
3761 64.439687538  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=131073 Win=196480 Len=0 TSval=2209871348 TSecr=3138032425
3762 64.439698436  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [ACK] Seq=131073 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871346 [TCP segment of a reassembled PDU]
3763 64.439701638  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=163841 Win=327552 Len=0 TSval=2209871348 TSecr=3138032425
3764 64.439712815  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [PSH, ACK] Seq=163841 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871348 [TCP segment of a reassembled PDU]
3765 64.439716682  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=196609 Win=458496 Len=0 TSval=2209871348 TSecr=3138032425
3766 64.439724992  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [ACK] Seq=196609 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871348 [TCP segment of a reassembled PDU]
3767 64.439727066  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=229377 Win=589440 Len=0 TSval=2209871348 TSecr=3138032425
3768 64.439737960  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [PSH, ACK] Seq=229377 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871348 [TCP segment of a reassembled PDU]
3769 64.439739888  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=262145 Win=720384 Len=0 TSval=2209871348 TSecr=3138032425
3770 64.439750558  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [ACK] Seq=262145 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871348 [TCP segment of a reassembled PDU]
3771 64.439752289  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=294913 Win=851328 Len=0 TSval=2209871348 TSecr=3138032425
3772 64.439762967  127.0.0.1     10.20.30.51   TCP   32836 80 → 39031 [PSH, ACK] Seq=294913 Ack=139 Win=65536 Len=32768 TSval=3138032425 TSecr=2209871348 [TCP segment of a reassembled PDU]
3773 64.439764979  10.20.30.51   127.0.0.1     TCP      68 39031 → 80 [ACK] Seq=139 Ack=327681 Win=982272 Len=0 TSval=2209871348 TSecr=3138032425
```

- The packets 3749 to 3751 : it's a TCP 3-Way Handshake Process.

- The packet 3752 : [200 GET /image-site.png HTTP/1.1] The client asking the file : image-site.png.

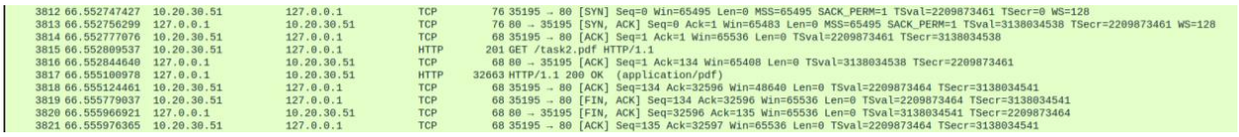
- The next packets going to be the download of this file , as we can see many packets of [ACK] and [PSH,ACK].



```
3809 64.443336020  10.20.30.51    127.0.0.1     TCP      68 39031 → 80 [FIN, ACK] Seq=139 Ack=1360876 Win=1712384 Len=0 TSval=2209871352 TSecr=3138032427
3810 64.443553196  127.0.0.1     10.20.30.51   TCP      68 80 → 39031 [FIN, ACK] Seq=1360876 Ack=140 Win=65536 Len=0 TSval=3138032429 TSecr=2209871352
```

- The packet 3809 and 3810 : The client got the file 500MB.zip , so now he can close the connection with the flag [FIN, ACK].

The last file : task2.pdf



```
3812 66.552747427  10.20.30.51         127.0.0.1          TCP        76 35195 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2209873461 TSecr=0 WS=128
3813 66.552756299  127.0.0.1          10.20.30.51        TCP        76 80 → 35195 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3138034538 TSecr=2209873461 WS=128
3814 66.552777076  10.20.30.51         127.0.0.1          TCP        68 35195 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2209873461 TSecr=3138034538
3815 66.552809537  10.20.30.51         127.0.0.1          HTTP      201 GET /task2.pdf HTTP/1.1
3816 66.552844640  127.0.0.1          10.20.30.51        TCP        68 80 → 35195 [ACK] Seq=1 Ack=134 Win=65408 Len=0 TSval=3138034538 TSecr=2209873461
3817 66.555100978  127.0.0.1          10.20.30.51        HTTP    32663 HTTP/1.1 200 OK  (application/pdf)
3818 66.555124461  10.20.30.51         127.0.0.1          TCP        68 35195 → 80 [ACK] Seq=134 Ack=32596 Win=48640 Len=0 TSval=2209873464 TSecr=3138034541
3819 66.555779037  10.20.30.51         127.0.0.1          TCP        68 35195 → 80 [FIN, ACK] Seq=134 Ack=32596 Win=65536 Len=0 TSval=2209873464 TSecr=3138034541
3820 66.555966921  127.0.0.1          10.20.30.51        TCP        68 80 → 35195 [FIN, ACK] Seq=32596 Ack=135 Win=65536 Len=0 TSval=3138034541 TSecr=2209873464
3821 66.555976365  10.20.30.51         127.0.0.1          TCP        68 35195 → 80 [ACK] Seq=135 Ack=32597 Win=65536 Len=0 TSval=2209873464 TSecr=3138034541
```

- The packets 3812 to 3814 : it's a TCP 3-Way Handshake Process.

- The packet 3815 : [200 GET /task2.pdf HTTP/1.1] The client asking the file : task2.pdf.

- The packet 3816 : The http Server send an ACK to the Client that he got his request.

- The packet 3817 : [HTTP/1.1 200 OK …] This packet is the server response to the previous HTTP request. "HTTP/1.1" indicates the protocol version used, "200" is the response status code: means that the request was processed successfully. The content of the requested resource (task2.pdf) is included in this package

- The packet 3818 : The Client send an ACK to the HTTP , because he got with success the answer of his request.

- The packets 3819 to 3821 : To close the TCP connection

IV- DNS

Introduction :

The DNS server, or Domain Name System, is a system that allows domain names (for example, www.example.com) to be translated into IP addresses (like 192.0.2.1), making it easier to browse the Internet. This is a protocol that usually works on port 53, using both UDP and TCP protocols. DNS servers are essential to ensure efficient communication between clients (computers, smartphones, etc.) and servers hosting websites. The importance of a DNS server lies in its ability to facilitate access to websites, by allowing users to connect to the servers hosting these sites simply by using their domain name. Without DNS servers, users would have to remember and enter IP addresses to access sites, which would be very inconvenient. In this text, we will discuss the technical aspects of how a DNS server works, the different queries and types of DNS servers, as well as the practical implementation of a DNS server within the framework of our project.

How a DNS server work ?

A DNS (Domain Name System) server is a domain name system that acts as an intermediary between clients and servers hosting websites. Its primary role is to translate domain names into IP addresses, allowing users to browse the Internet using easy-to-remember names rather than numeric IP addresses.

The domain name resolution process begins when the user enters a domain name in their browser. The client then sends a request to the local DNS server (usually provided by the Internet access provider) to obtain the IP address associated with the domain name. If the local DNS server does not cache this information, it queries DNS servers higher in the hierarchy until it gets the answer. The DNS hierarchy begins with the root servers, which are at the top of the structure and contain information about top-level domain (TLD) name servers, such as .com, .org, or .net. TLD servers contain information about second-level domain name servers, which are usually operated by individual companies or organizations.

There are two types of DNS queries: recursive and iterative. When a client sends a recursive query to a DNS server, it asks the server to provide the IP address associated with the domain name or forward the query to other servers until the

answer is found. The DNS server then responds with the IP address or an error, if the domain name is not resolved. In an iterative query, the client asks the DNS server to provide the IP address associated with the domain name, without requiring the server to forward the query to other servers. If the DNS server does not know the IP address, it returns information about the higher DNS server in the hierarchy. The client can then send a new iterative request to this superior server.

There are several types of DNS servers, each with a specific role in the DNS hierarchy:

- Root Server: This is the first server consulted in the domain name resolution process. It contains information about TLD servers and directs requests to them.
- TLD Server: This server stores information about second-level domain name servers and is responsible for managing top-level domains (such as .com, .org, etc.).
- Local DNS server: This is the DNS server to which clients send their queries directly. It can be managed by the Internet service provider or a company.

In our project :

As part of this project, a DNS server was set up to respond to DNS queries and resolve domain names to IP addresses. The DNS server code was written in Python and uses the socket module to handle network communications. The socket module was used to create a UDP server and listen for DNS queries on port 53. The DNS server IP address was set to "127.0.0.9" and the fixed IP address to return for queries was set to "127.0.0.104" (the different addresses can change).

Socket initialization: The socket was initialized using socket.AF_INET (for IPv4) and socket.SOCK_DGRAM (for UDP). The socket has been bound to the IP address and port defined earlier.

Implementation of DNS resolution functions: Several functions have been implemented to handle the DNS resolution process, such as getflags(), getquestiondomain(), buildquestion(), rectobytes(), and buildresponse.()

-Main loop: The DNS server operates continuously, waiting for incoming queries, processing those queries, and returning appropriate responses.

-DNS server test: The DNS server can be tested by using tools like nslookup or dig to send DNS queries and check if the responses are correct and consistent with expectations.

Conclusion :

In this text, we have presented the concept and operation of DNS servers, which are essential for facilitating access to websites and ensuring smooth communication between clients and servers. We covered the technical aspects of how DNS servers work, including the domain name resolution process, types of DNS queries (recursive and iterative), and the different types of DNS servers (root servers, TLD servers, and local/local DNS servers). authoritative).

Additionally, we looked at a practical case of creating a DNS server using Python and the socket module. This project made it possible to implement the knowledge acquired on DNS servers and to better understand the DNS system.

DNS servers play a crucial role in the stable and fast functioning of the Internet, by allowing users to easily connect to servers hosting websites using domain names instead of numerical IP addresses. Without DNS servers, internet browsing would be much less convenient and potentially slower, as users would have to remember and type in IP addresses to access sites.

V- DHCP

Introduction :

The DHCP server (Dynamic Host Configuration Protocol) is an essential element in the management of local area networks (LAN). It automatically assigns IP addresses and other network configuration settings to devices connected to a local network. By simplifying and automating the management of IP addresses, DHCP servers make it easier to set up and maintain networks and reduce the risk of human error.DHCP servers play a crucial role in local area networks by enabling efficient and centralized management of IP addresses and network configurations. Using DHCP servers, network administrators can manage IP address pools and automatically assign addresses to devices connected to the network. This avoids having to manually configure each device and allows better management of available IP resources. In this text, we will explore how DHCP servers work, as well as a practical case of creating a DHCP server.

How a DHCP server work :

DHCP Server is a service that automatically assigns IP addresses, subnet masks, default gateways, and other network configuration settings to devices connected to a local network. DHCP servers simplify and speed up the management of IP addresses by avoiding the need to manually configure each device.

The process of allocating IP addresses with DHCP follows a sequence called DORA (Discover, Offer, Request, Acknowledge). First, a device connected to the local network sends a Discover message to request an IP address from the DHCP server. The DHCP server responds with an Offer message which contains an available IP address for the device. Then the device sends a Request to confirm its choice of IP address. The DHCP server responds with an Acknowledge message that confirms the allocation of the IP address to the device. At this point, the device can begin to use the allocated IP address to communicate with the network.

In addition to IP address allocation, DHCP servers can also provide other network configuration parameters such as DNS server addresses . DHCP can be vulnerable to certain attacks, such as denial of service (DoS) attacks and IP

spoofing attacks. To counter these risks, networks can implement security mechanisms such as DHCP Snooping, which monitors and filters out malicious DHCP messages, and access controls to limit access to DHCP servers.

In our project :

As part of our project, we have created a simple DHCP server using the Python language and the Scapy library. We started by setting some variables for the network, such as the DHCP server IP address, the range of available IP addresses, the gateway address, the subnet mask, and the number of allocations. IP addresses.

We then created a "generate_ip" function to generate IP addresses from the available IP range. This function takes into account the number of allocations already made and returns a unique IP address from the range of available IP addresses.

We then created two functions "discover_to_offer" and "request_to_ack" to handle the DHCP Discover and Request messages respectively. The first function takes as input a Scapy packet containing a DHCP Discover message and the IP address offered by our DHCP server. It then constructs a DHCP Offer response using this IP address and sends the response to the DHCP client.

The second function takes as input a Scapy packet containing a DHCP Request message and sends a DHCP Acknowledge response to the DHCP client using the previously allocated IP address.

Finally, we used the main "main" function to run our DHCP server continuously, listening and responding to incoming DHCP Discover and Request packets. With these steps, we managed to create a simple DHCP server for our project.

Conclusion :

In conclusion, creating a DHCP server is a key part of managing a network, making it possible to centrally provide IP addresses to each connected device. In this project, we used Python and the Scapy library to create a simple yet effective DHCP server. By using the functions and variables we defined, we were able to offer a solution for more efficient IP address management.

This project allowed us to better understand the operation of the DHCP protocol and its role in the management of IP addresses in a network. We also learned how to use Scapy, a very useful network packet manipulation library.

Creating an efficient DHCP server can help maintain an organized network, avoiding address conflicts and ensuring stable connectivity for each device.

VI- RUDP (Reliable User Datagram Protocol)

Introduction :

The UDP protocol is widely used for transmitting data over networks because it is simple and efficient. However, it does not provide a transmission reliability mechanism, which can lead to data loss or corruption. To overcome this problem, the RUDP protocol (Reliable User Datagram Protocol) was developed. RUDP is a transport layer protocol that ensures the reliability of data transmission over networks using techniques such as flow control, retransmission of lost packets and acknowledgment of received packets. Although less commonly used than UDP, RUDP is an attractive choice for applications that require reliable data transmission over networks. In this text, we will examine how RUDP works, the differences between RUDP and UDP, the advantages and disadvantages of RUDP, and we will also describe a practical case of using RUDP for communication between a client and a server. Proxy.

How the protocol RUDP work :

RUDP is a transport protocol that provides guarantees of data transmission reliability by using techniques of flow control, retransmission of lost packets and acknowledgment of received packets. The protocol was developed to overcome the limitations of UDP, which does not provide a transmission reliability mechanism. Unlike TCP, which uses a transmission control protocol to ensure reliable data transfers, RUDP is datagram-based and does not involve a prior connection between hosts.

The differences between RUDP and UDP are mainly related to the reliability of data transmission. While UDP is an unreliable transport protocol, RUDP ensures reliable data transmission by using techniques of retransmitting lost packets and acknowledging received packets. Unlike TCP, which uses a flow control system to avoid network congestion, RUDP uses a congestion control mechanism based on a maximum allowed throughput for each connection. This ensures fast and reliable data transmission without overloading the network.

The process of sending and receiving RUDP packets is similar to that of UDP, with some differences related to transmission reliability mechanisms. When a host sends a RUDP packet, it expects an acknowledgment (ACK) from the

destination host. If the ACK is not received within a specified time, the packet is retransmitted. The destination host verifies the integrity of the received packet and sends an ACK back to the originating host to indicate correct receipt of the packet. If a packet is lost during transmission, the destination host can request its retransmission by sending a retransmission request message. The advantages of RUDP lie primarily in its ability to provide reliable data transmission without the time and resource costs of establishing a connection first, as is the case with TCP.

However, RUDP is not suitable for all situations. It can cause excessive use of network resources in high-congestion environments, and its use may require application adaptations that were not designed to support transmission reliability. Furthermore, RUDP does not guarantee delivery

In our project :

To guarantee the reliability of data transmission, we have chosen to implement a reliable transport protocol. We opted for a RUDP protocol (Reliable User Datagram Protocol), a simplified variant of the UDP protocol (User Datagram Protocol), which ensures the delivery of data without guaranteeing their order. In our implementation of RUDP, the server manages a flow control system that limits the number of packets sent by the client at the same time, in order to avoid network congestion. The server also keeps track of the packets sent, to ensure that each packet is received correctly and to avoid data loss. The client, on the other hand, sends numbered packets with a unique identifier for each packet, so that the server can verify that all packets are received correctly and in the correct order. If a packet is not received, the client retransmits the missing packet.

We have also put several flags or SIGNAL in place to allow interaction between the client and the server:

- SIGNEW: This signal is used by a new client to register with the server. The client sends this signal containing his username to the server, which records the client's information and sends back an acknowledgment (ACK).
- SIGET: This signal is used by a client to request a list of available files from the server. The server sends a response containing the list of requested files.
- SIGNECHO: This signal is used by a client to send an echo request to the server, which returns a response containing the response time between itself and the server.

- SIGNSTAT: This signal is used by a client to request statistics on server performance. The server returns a response containing the requested statistics.
- SIGNEND: This signal is used by a client to unsubscribe from the server and close the connection. The server returns an acknowledgment (ACK) and closes the connection with the client.
- SIGNACK: this signal is used by the server to confirm receipt of a packet sent by a client. The server returns this acknowledgment (ACK) to indicate to the client that its packet has been received.
- SIGNLOST: This signal is used by the server to inform a client that a packet sent by the client has been lost. The customer can then resend the missing package.

Conclusion :

In conclusion, we have set up a communication protocol based on the use of the UDP protocol with the management of the reliability of the data exchanged. We have implemented a flow and congestion control mechanism to optimize data transmission between the server and the clients. We have also defined different signals to enable smooth communication between server and clients, such as SIGNNEW for creating a new client, SIGNGET for data retrieval, SIGNLOST for signaling packet loss, SIGNFULL for informing a client that its congestion window is full, and SIGNEND for the end of the communication. All of these functionalities make it possible to guarantee the security and reliability of data exchanges between the server and the clients, while optimizing the transmission of data in an unstable and/or congested network environment.

VII- Question about Protocols and the project

1) מנה לפחות ארבע הבדלים עיקריים בין פרוטוקול **TCP** ל-**QUIC**

<u>הבדל ראשון:</u> הגדרת חיבור

כאשר נשווה את הגדרת החיבורים בין שני הפרוטוקולים נוכל ראות הבדלים משמעותיים בהגדרת החיבור הראשונית.
בשונה מTCP המבצע את לחיצת הידיים להקמת חיבור נוכל לראות כי QUIC מתבסס על Zero Round Trip משמע שהוא חוסך את "ההלוך חזור" המתבצע בTCP ובמקרה שלו ישירות מגדיר את החיבור.

<u>הבדל שני:</u> שידורים מרובים

פרוטוקול QUIC מאפשר לבצע שידורים מרובים תחת חיבור יחיד בשונה מTCP המחייב בשידור מרובה חיבורים לצור אל מול כל שידור חיבור נוסף.

<u>הבדל שלישי:</u> הצפנה

פרוטוקול QUIC מצפין את המידע המשודר תחת החיבורים כולל השידור המקיים את החיבור הראשוני, בשונה מTCP אשר אכן מקיים הצפנת מידע אך ורק תחת חיבור מאובטח.

<u>הבדל רביעי:</u> החלמה מאיבוד פאקטות

נוכל לראות הבדלים משמעותיים כאשר נשווה את האלגוריתמים של "ההחלמה" מול שני הפרוטוקולים. באופן גורף כאשר מדובר ב QUIC נוכל לראות כי הפעולה מתבצעת מאופן מהיר יותר תודות לשימוש באיתור באופן פרטני ולא גורף של אותם ACKים ושליחתם מחדש.

<u>הבדל חמישי:</u> דחיסת הHEADER

QUIC משתמש בדחיסת הנתונים המופיעים בHEADER בצורה יעילה יותר ובכך מונע את הגדלת הפאקטות הנשלחות באופן חוזר ונשנה למטרת החיבור מה שמקל באופן ישיר על השידור ומאפשר האצה במהירות השידור.

2) מנה לפחות שני הבדלים עיקריים בין **Cubic** ל-**Vegas**

<u>הבדל ראשון:</u> אלגוריתם ה – CC

ההבדל המשמעותי ביותר ביניהם הוא אלגוריתם ה CC אשר שניהם מבוססים עליהם.

Cubic משתמש בפונקציה מעוקבת כדי לקבוע את הקצב שבו ניתן לשלוח נתונים.

Vegas משתמשת בשילוב של מדידת אובדן המנות ועיכובן לשליטה בקצב השידור.

<u>הבדל שני:</u> מדידת ה "הלוך ושוב"

Cubic מבצע מדידה של זמן השידור הלוך ושוב (RTT) של השידור,

במילים אחרות - הזמן שלוקח לחבילה לעבור מהשולח למקבל ובחזרה, לעומת Vegas המודד רק את ההמתנה המתקיימת בצד אחד.

<u>הבדל שלישי:</u> גודל החלון

Cubic משנה את גודל החלון שלו על סמך החלון של אלגוריתם הCC שלו ומדידת הזמן של השידור הלוך ושוב.

Vegas משתמש בחלון של אלגוריתם הCC שלו ובמדידת ההמתנות כדי שלנות את גודל החלון שלו.

3) הסבר מהו פרוטוקול **BGP**, במה הוא שונה מ-**OSPF** והאם הוא עובד על פי מסלולים קצרים

<u>הסבר קצר על BGP:</u>

BGP הוא פרוטוקול המשמש לשליחת מידע על מצב התקשורת בין רשתות שונות באינטרנט, בנוסף הוא בוחר את הנתיב הטוב ביותר להעברת הנתונים כאשר נלקח החשבון, ראשית הנתיב הקצר ביותר והעלות הנמוכה ביותר.

שימושו העיקרי הוא מצד ספקי שירותי האינטרנט לצורך חיבור הרשתות שברשותם והחלפת מסלולים עם ספקי שירותי אחרים.

<u>שוני אל מול OSPF:</u>

OSFP הוא פרוטוקול פנימי המשמש כמערכת אוטומטית לשינוע מידע הנוגע לרשת בין נתבים, בעוד BGP הוא פרוטוקול חיצוני המשמש להחלפת מידע על ניתוב בין הנתבים, בנוסף OSPF מחשב את הנתיב הקצר ביותר על סמך מסד הנתונים של מצב קישור, בעוד BGP בוחר את הנתיב הטוב ביותר על סמך כללי מדיניות ותכונות.

4) בהינתן הקוד שפיתחתם בפרויקט זה, אנא הוסיפו את הנתונים לטבלה הזו על בסיס תהליך ההודעות של הפרויקט שלכם. הסבירו איך ההודעות ישתנו אם יהיה **NAT** בין המשתמש לשרתים והאם תשתמשו בפרוטוקול **QUIC**

on	Port Src	Port Des	IP Src	IP Des	MAC Src	MAC Des
0.1	49152	49152	127.0.0.10	10.20.30.51	c:29:79:2c:f900:0	c:29:79:2c:f900:0

כאשר ננסה להריץ את הפרויקט תחת NAT נוכל לראות כי כתובות הIP לא משתנות מכיוון שNAT מנתב את תקשורת המערכות הווירטואליות דרך כתובת הIP של המארח ובכך חלק מפונקציונליות שהשגנו על ידי עבודה עם DHCP שאינו מנוון אינן מתקיימות ולכן עבודה תחת רשת פנימית ובאמצעות Bridge מאפשרת לDHCP לחלק כתובות ובכך נוכל לבצע את המבוקש.

עבודה תחת QUIC לא עשינו מכיוון שהחיבור בין הקליינט לבין שרת הProxy מתבצע תחת תקשורת RUDP ושרת הProxy מפנה ישירות

אל שרת ההורדה לפי בקשת הRe-Direct, ובכך הקליינט מבצע את הורדת הקבצים ישירות אל מולApplicartion Server .

5) הסבירו את ההבדלים בין פרוטוקול **ARP** ל-**DNS**

DNS ו-ARP הם פרוטוקולים המספקים מידע על רשתות מחשבים המחוברות אליהן.

DNS משמש לתרגום שמות דומיין לכתובות IP, בעוד ARP משמש למיפוי כתובות IP לכתובות MAC,

DNS הוא מאגר נתונים המושטת באופן היררכי המכיל מידע על המכשירים המחוברים אל הרשת בעוד ARP משמש לתקשורת בין מכשירים באותם חלקי פנימיים של הרשת.

בנוסף, DNS פועל בשכבת האפליקציה ו- ARP פועל בשכבת הלינק.