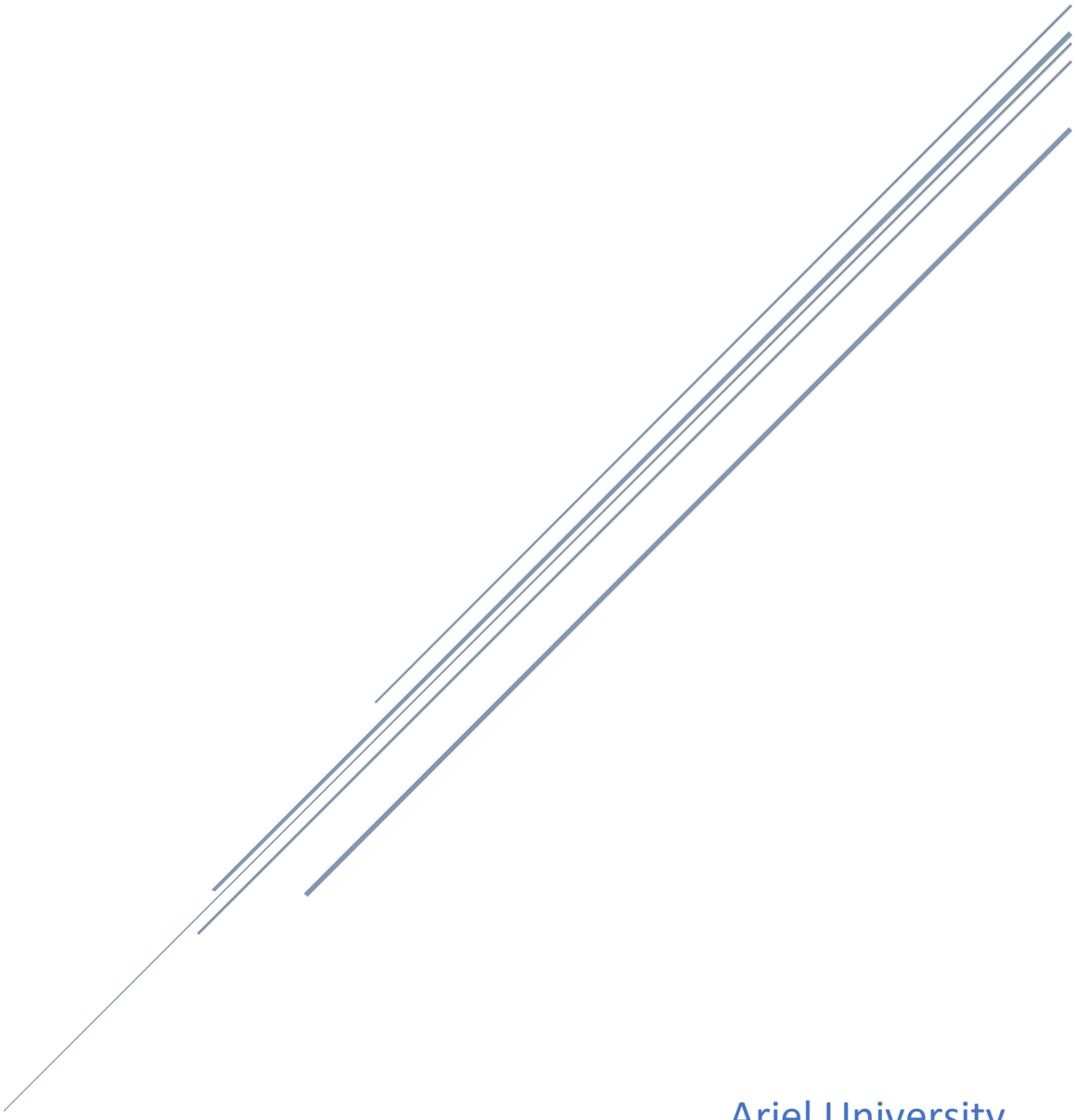
HTTP APPLICATION

CN_FINALL

Maor Saadon 318532421

Duvi Amiram 305677494



Ariel University
Connection networking

Contents

1.1 About the System

1.1.1 Project overview

My project focuses on three servers and a client, with the servers being DNS, DHCP, and HTTP application servers. The DNS server will be responsible for translating domain names into IP addresses, while the DHCP server will be in charge of assigning IP addresses dynamically to the client devices. The HTTP application server will host a web application that the client can access through a web browser.

Additionally, the HTTP application server will have the role of performing redirects to another server and downloading files from there. The client device will be configured to interact with all three servers, ensuring seamless communication between them. The project aims to showcase how these servers can work together to provide efficient network services to client devices.

We will expand on the RUDP protocol later, but to answer the questions:

1. how do you overcome lost packages?

2. how the system overcomes latency problems?

Well as requested our system based on the RUDP protocol. A small explanation is given about the RUDP protocol:

RUDP (Reliable User Datagram Protocol) is a transport layer protocol that provides reliability on top of the User Datagram Protocol (UDP) by implementing features such as error detection, retransmission of lost packets, and congestion control. However, even with these features, RUDP may still experience packet loss due to various reasons such as network congestion, faulty network equipment, or temporary network interruptions.

1. To overcome lost packages when using RUDP, the protocol implements a mechanism for retransmitting lost packets. When a packet is sent by the sender, it is assigned a sequence number that is used to track the packets at the receiver. If a packet is lost or not acknowledged by the receiver, the sender will retransmit the packet after a certain amount of time (known as the retransmission timeout).

The retransmission timeout is dynamically adjusted based on the network conditions, and it is typically set to a value that allows the packet to be retransmitted before the receiver's timeout period expires. This helps to ensure that the packet is retransmitted as quickly as possible without causing unnecessary delay or congestion on the network.

In addition to retransmitting lost packets, RUDP also uses checksums to detect errors in the packets. If an error is detected, the receiver will request that the sender retransmit the packet.

Overall, by using a combination of retransmission, error detection, and congestion control mechanisms, RUDP is able to provide reliable data transmission even in the face of packet loss.

2. To overcome latency problems when using RUDP, the protocol implements several techniques that help to minimize the impact of latency on data transmission. These techniques include:

a. Retransmission timeout (RTO) adjustment: RUDP dynamically adjusts the RTO based on the network conditions, such as the round-trip time (RTT) of the packets. This helps to ensure that retransmissions are sent quickly enough to minimize the impact of latency.

b. Selective retransmission: RUDP allows the sender to retransmit only the lost packets, rather than retransmitting all the packets in a stream. This reduces the amount of data that needs to be transmitted and can help to reduce latency.

c. Pipelining: RUDP allows the sender to transmit multiple packets without waiting for an acknowledgement for each packet. This helps to keep the data flowing and can help to reduce the impact of latency.

d. Congestion control: RUDP uses congestion control mechanisms, such as slow start and congestion avoidance, to prevent the network from becoming congested and causing increased latency.

Overall, by using these techniques, RUDP is able to minimize the impact of latency on data transmission and provide reliable and efficient data transfer over the network.

1.1.2 DHCP

What is that DHCP?

A Dynamic Host Configuration Protocol (DHCP) server is a network server that automatically assigns IP addresses and other network configuration parameters to devices on a network. DHCP simplifies network administration by allowing devices to obtain network settings automatically instead of requiring manual configuration.

When a device, such as a computer or a smartphone, connects to a network, it sends a request to the DHCP server for an IP address. The DHCP server responds by assigning a unique IP address to the device, along with other network configuration information such as subnet mask, default gateway, and DNS server addresses.

DHCP servers are commonly used in local area networks (LANs) and are particularly useful in large organizations where manual IP address assignment would be time-consuming and error-prone. DHCP servers can also be configured to assign specific IP addresses to specific devices, which is useful for devices that require a static IP address, such as servers or network printers.

1.1.3 DNS

The Domain Name System (DNS) is a hierarchical distributed naming system for computers, services, or other resources connected to the Internet or a private network. DNS translates human-readable domain names, such as "google.com", into IP addresses that computers use to identify each other on the network.

When a user types a domain name into a web browser, the browser sends a request to a DNS resolver, which is typically provided by the user's Internet service provider (ISP). The resolver then queries one or more DNS servers to obtain the IP address associated with the domain name. Once the IP address is obtained, the browser can then connect to the web server associated with that IP address to retrieve the requested web page.

DNS is critical to the functioning of the Internet as it enables users to access websites and other online services using memorable domain names instead of having to remember the numerical IP addresses associated with those services. DNS is also used for other purposes such as email routing, network address translation (NAT), and other network services.

1.1.4 HTTP APP

HTTP stands for Hypertext Transfer Protocol, which is the primary protocol used for transferring data over the World Wide Web. An HTTP application, also known as a web application, is a software application that uses HTTP as its primary communication protocol to interact with users and other software applications over the internet.

HTTP applications are typically accessed using a web browser, which sends HTTP requests to the web server hosting the application. The server then responds to the requests by sending back HTTP responses, which may include HTML, CSS, JavaScript, images, or other types of data.

Web applications can perform a wide range of tasks, from simple tasks such as displaying web pages or handling user authentication to complex tasks such as online shopping, social networking, and collaborative document editing. Popular examples of web applications include online email services, social media platforms, online banking, and e-commerce websites.

HTTP applications can be built using a variety of programming languages, frameworks, and tools, depending on the specific requirements of the application. Some popular web development frameworks include Ruby on Rails, Django, and Node.js, while popular web development tools include HTML, CSS, JavaScript, and jQuery.

In our case, when a client, such as a web browser, sends a request to a server using the HTTP protocol, the server can respond with an HTTP redirect response code (such as 301 or 302) to indicate that the requested resource has been moved permanently or temporarily to a different location.

In the case of a file download, this could mean that the file is no longer hosted on the original server, but has been moved to a different server or location. The redirect response will include a new URL for the client to request the resource from the new location.

Once the client receives the redirect response, it will typically send a new request to the new URL, and the server at that location will respond with the requested file for download.

This process of redirecting to a new server or location can be useful in a variety of scenarios, such as when a website is moved to a new domain or when large files are hosted on a separate file server for better performance.

1.1.5 RUDP

RUDP (Reliable User Datagram Protocol) is a transport layer protocol designed to provide a reliable, connection-oriented service over unreliable, connectionless transport protocols such as UDP (User Datagram Protocol).

The main purpose of RUDP is to provide a reliable data transfer service to applications that require guaranteed delivery of data. It does this by adding reliability and flow control mechanisms to UDP.

RUDP provides reliable delivery of data by implementing mechanisms such as retransmission of lost or damaged packets, packet sequencing, and acknowledgment of received packets. RUDP also provides flow control to prevent the receiver from being overwhelmed by the sender's data.

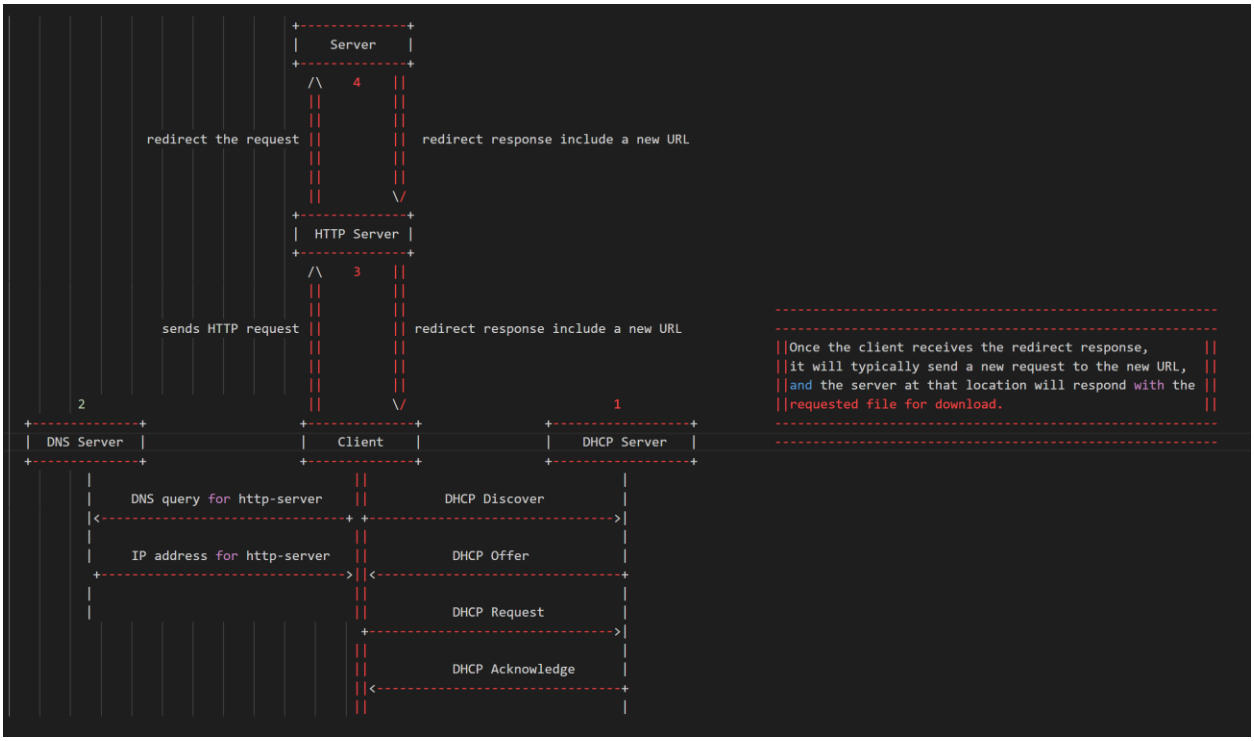
In addition to reliability and flow control, RUDP also supports congestion control. Congestion control mechanisms are used to avoid network congestion by controlling the rate at which data is transmitted. This helps to prevent network congestion, which can result in dropped packets and degraded performance.

RUDP has some advantages over other reliable transport protocols such as TCP (Transmission Control Protocol). One advantage is that RUDP is less complex than TCP, which makes it more suitable for use in embedded systems and other environments with limited resources. RUDP also has lower latency and overhead than TCP, which can be beneficial in applications that require real-time or low-latency data transfer.

However, RUDP also has some disadvantages compared to TCP. For example, RUDP does not provide congestion control as effectively as TCP, which can lead to network congestion and performance issues. Additionally, RUDP is not widely used and may not be supported by all network devices and applications.

Overall, RUDP is a reliable, connection-oriented protocol that can be a good alternative to TCP in certain situations. It provides mechanisms for reliable data transfer, flow control, and congestion control, while also being less complex and having lower latency than TCP. However, its effectiveness in preventing network congestion and its limited support in some environments should be taken into a count when considering its use.

1.1.6 Diagram of our system



The flow of the system is as follows:

1. The client sends a DHCP request to the DHCP server to obtain an IP address.
2. The DHCP server assigns an IP address to the client.
3. The client sends a DNS query to the DNS server to resolve the hostname of the HTTP server.
4. The DNS server responds to the client with the IP address of the HTTP server.
5. The client sends an HTTP request to the HTTP server to download a file.
6. The HTTP server responds to the client with the file.
7. The client receives the file.

1.1.7 How to run the program

download all the files.

open the directory containing the files, in 5 teminals in a lonux environment.

write the following commands, one in each teminal:

1. $ sudo python3 DNS.py

2. $ sudo python3 DHCP.py

3. $ sudo python3 TCPDownloadManager.py

4. $ sudo python3 RUDPDownloadManager.py.py

5. $ sudo python3 Client.py

press ctrl + click on the link that appears

choose your desired host and txt or html file name, and the desired protocol for transmission and click on "Submit Request".

enjoy, while the app downloads your desired file to the directory!

1.2 DHCP

1.2.1 Code Description

Before we will jump into the code, we want to explain about D.O.R.A.

DORA is an acronym for the four-step process used by DHCP to assign IP addresses to devices on a network.
Here are the steps involved:

1. Discover: The client device sends out a broadcast message on the network, requesting an IP address. The broadcast message includes the client's MAC address and other information about the client.

2. Offer: The DHCP server receives the broadcast message and responds with an IP address offer. The server sends a broadcast message back to the client, including an available IP address and other configuration information, such as the subnet mask, DNS servers, and default gateway.

3. Request: The client receives the IP address offer from the DHCP server and decides whether to accept it. If the client accepts the offer, it sends a broadcast message to the server, indicating that it wants to use the offered IP address.

4. Acknowledge: The DHCP server receives the broadcast message from the client and sends a final message to the client, acknowledging that the IP address has been assigned to the client. This message includes the lease time for the IP address, which indicates how long the client can use the address before it needs to renew the lease.

Overall, the DORA process ensures that IP addresses are assigned in a coordinated and efficient manner, minimizing the risk of IP address conflicts, and making it easier for network administrators to manage network resources.

DHCP server:

1. Import necessary libraries:

The first step of the code is to import the necessary libraries, namely **scapy.all**, **scapy.layers.dhcp**, **scapy.layers.inet**, and **scapy.layers.l2**. These libraries are used throughout the code to handle and manipulate network packets.

```
from scapy.all import *

from scapy.layers.dhcp import DHCP, BOOTP
from scapy.layers.inet import IP, UDP
from scapy.layers.l2 import Ether
```

2. Instantiate a DhcpHandler object & Use scapy to sniff network traffic:

The next step is to create an instance of the DhcpHandler class. And then we will use scapy's **sniff** function to capture and process network traffic that matches a specified filter. The **sniff** function takes two arguments: a filter expression and a callback function (handler.handle in this case). The filter expression is used to specify the types of packets that should be captured and processed by the handle method. In this case, the filter expression specifies that UDP packets with a source or destination port of 67 or 68 should be captured and processed.

```
handler = DhcpHandler()
#capture and process network traffic that matches a specified filter.
sniff(filter='udp and (port 67 or port 68)', prn=handler.handle)
```

3. Initialize instance variables:

The **DhcpHandler** class has an **__init__** method that initializes two instance variables. The **ip_pool** variable is a list of IP addresses that can be assigned to clients. The **ip_assignments** variable is a dictionary that maps assigned IP addresses to the MAC addresses of the clients that have been assigned those addresses.

```
def __init__(self):
    self.ip_pool = ['192.168.0.%d' % i for i in range(100, 200)]
    self.ip_assignments = {}
```

4. Define a method to get the next available IP address:

The **DhcpHandler** class has a **get_next_available_ip** method that returns the next available IP address from the **ip_pool** list. If there are no more available IP addresses, the method returns None.

```python
def get_next_available_ip(self):
    for ip_address in self.ip_pool:
        if str(ip_address) not in self.ip_assignments:
            return str(ip_address)
    return None
```

5. Define a method to handle DHCP packets:

The **DhcpHandler** class has a **handle** method that is responsible for handling DHCP packets. The method takes a packet object as an argument and checks whether the packet is a DHCP Discover or a DHCP Request.

If the packet is a DHCP Discover, the method extracts the client's MAC address from the packet, assigns the next available IP address to the client, constructs a DHCP offer packet, and sends the offer packet to the client.

```python
if packet[DHCP] and packet[DHCP].options[0][1] == 1: # DHCP Discover
    print("DHCP Discover received")
    # Extract the client's MAC address from the request
    mac_address = packet[Ether].src.replace(':', '')

    # Assign the next available IP address to the client
    ip_address = self.get_next_available_ip()
    # print(ip_address)
    if not ip_address:
        return  # No more IP addresses available

    # Add the IP address assignment to the dictionary
    self.ip_assignments[ip_address] = mac_address

    # Construct the DHCP offer
    offer =Ether(src=get_if_hwaddr(conf.iface), dst=packet[Ether].src) / \
            IP(src=get_if_addr(conf.iface), dst='255.255.255.255') / \
            UDP(sport=67, dport=68) / \
            BOOTP(op=2, yiaddr=ip_address, siaddr=get_if_addr(conf.iface), chaddr=packet[Ether].src) / \
            DHCP(options=[('message-type', 'offer'),
                          ('server_id', get_if_addr(conf.iface)),
                          ('lease_time', 1200),
                          ('subnet_mask', '255.255.255.0'),
                          ('router', get_if_addr(conf.iface)),
                          'end', 'pad'])
bu, 4 days ago • Initial commit …
    # Send the DHCP offer to the client
    time.sleep(2)
    print("sending DHCP offer")
    sendp(offer, iface=conf.iface)
```

If the packet is a DHCP Request, the method extracts the client's MAC address and requested IP address from the packet, assigns the requested IP address to the client, constructs a DHCP ACK packet, and sends the ACK packet to the client.

```
elif packet[DHCP] and packet[DHCP].options[0][1] == 3: # DHCP Request       You, 22 minutes ago • Uncommi
    print("DHCP Request received")
    # Extract the client's MAC address and requested IP address from the request
    mac_address = packet[Ether].src.replace(':', '')
    requested_ip = packet[BOOTP].yiaddr

    # Assign the requested IP address to the client
    ip_address = requested_ip

    # Construct the DHCP ACK
    ack =Ether(src=get_if_hwaddr(conf.iface), dst=packet[Ether].src) / \
          IP(src=get_if_addr(conf.iface), dst='255.255.255.255') / \
          UDP(sport=67, dport=68) / \
          BOOTP(op=5, yiaddr=ip_address, siaddr=get_if_addr(conf.iface), chaddr=packet[Ether].src) / \
          DHCP(options=[('message-type', 'ack'),
                        ('server_id', get_if_addr(conf.iface)),
                        ('lease_time', 1200),
                        ('subnet_mask', '255.255.255.0'),
                        ('router', get_if_addr(conf.iface)),
                        ('dnsservers', '127.0.0.1'),
                        'end', 'pad'])

    # Send the DHCP ACK to the client
    time.sleep(2)
    print("sending DHCP ack")
    sendp(ack, iface=conf.iface)
```

1.2.2 Flowchart

Here's a simple flowchart that outlines the basic steps involved in a DHCP lookup process between a DHCP client and server:

1. Get MAC address of client interface

2. Send DHCP Discover packet to broadcast address

3. Wait for DHCP Offer packet from server

4. If Offer received:

a. Send DHCP Request packet with offered IP address and server ID

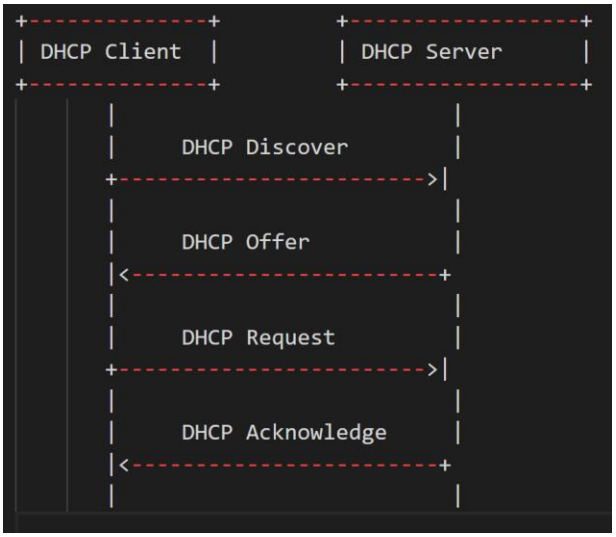
b. Wait for DHCP Acknowledgement packet from server

5. If Acknowledgement received:

a. Configure network interface with assigned IP address and lease time

b. End

Here's a simple diagram that illustrate the communication between a DHCP server and a DHCP client:



In this diagram, the DHCP client sends a DHCP Discover message to the broadcast address. The DHCP server receives the message and responds with a DHCP Offer message. The client then sends a DHCP Request message requesting the offered IP address, and the server sends a DHCP Acknowledge message to confirm that the IP address has been assigned to the client.

1.3 DNS

1.3.1 Code Description

The DNS server receives DNS requests from clients, looks up the requested domain name in its DNS zones, and returns the corresponding IP address to the client. The code is divided into several functions, each of which performs a specific task.

Here's a step-by-step explanation of the code:

1. Import the socket module. This module provides low-level network communication functionality, including the ability to create and interact with sockets.

```
import socket
```

2. Define a class named DNSserver. This class has two instance variables: 'addresses' is a dictionary that maps domain names to IP addresses, and 'ip_domain' is a variable that stores the resolved IP address for a given domain name.

```
class DNSserver:
    def __init__(self):
        self.addresses = {}
        self.ip_domain = None
```

3. Define a method named **'receive_dns_query'** that listens for incoming DNS queries and handles them. The method does the following:

a. Creates a UDP socket using the socket.socket() method with parameters 'socket.AF_INET' for IPv4 and 'socket.SOCK_DGRAM' for UDP.

b. Binds the socket to a local address and port using the bind() method with the parameter tuple ('127.0.0.1', 53), which means the socket will listen for incoming connections on all network interfaces and port 53 (the standard port for DNS).

c. Starts an infinite loop that waits for incoming packets using the recvfrom() method of the socket. The method blocks until a packet is received. When a packet is received, the method calls the 'handle_query_response' method to construct a DNS response packet, and sends the response packet back to the client using the sendto() method of the socket.

```
def recieve_dns_query(self):
    # create a UDP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # bind the socket to a local address and port
    server_socket.bind(('127.0.0.1' , 53))

    while True:
        # receive a DNS query packet
        query_packet, client_address = server_socket.recvfrom(4096)

        # construct the DNS response packet
        response_packet = self.handle_query_response(query_packet)

        # send the response packet to the client
        server_socket.sendto(response_packet, client_address)
```

4. Define a method named 'handle_query_response' that constructs a DNS response packet for a given DNS query packet. The method does the following:

a. Calls the 'extract_domain' method to extract the domain name from the query packet.

b. Checks if the domain name is in the 'addresses' dictionary. If it is, the method retrieves the IP address from the dictionary. If it's not, the method resolves the IP address using the gethostbyname() method of the socket module.

c. Constructs the DNS response packet by concatenating byte strings that represent the different fields of the packet. The response packet includes the ID, QR flag, Opcode, AA flag, TC flag, RD flag, RA flag, Z flag, RCODE, QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT, the question section of the query packet, and the answer section that contains the IP address of the domain name.

d. Returns the response packet as a byte string.

```
def handle_query_response(self, query_packet):
    """
    Given a DNS query packet, construct a DNS response packet and return it.
    """
    # extract the domain name from the query packet
    domain = self.extract_domain(query_packet)

    # resolve the IP address for the domain name
    if domain in self.addresses:
        self.domain_ip = self.addresses[domain]
        print(f"Resolved {domain} to {self.domain_ip}, using our Database module. ")
    else:
        self.ip_domain = socket.gethostbyname(domain)
        print(f"Resolved {domain} -> {self.ip_domain}, using the socket module.")

    # construct the DNS response packet
    packet = b''
    packet += query_packet[:2]  # copy the ID from the query packet
    packet += b'\x81\x80'  # QR=1, Opcode=0, AA=1, TC=0, RD=1, RA=1, Z=0, RCODE=0
    packet += query_packet[4:6]  # copy the QDCOUNT from the query packet
    packet += b'\x00\x01'  # ANCOUNT=1
    packet += b'\x00\x00'  # NSCOUNT=0
    packet += b'\x00\x00'  # ARCOUNT=0
    # construct the question section
    packet += query_packet[12:]
    # construct the answer section
    packet += b'\xc0\x0c'  # pointer to the domain name in the question section
    packet += b'\x00\x01'  # TYPE=A
    packet += b'\x00\x01'  # CLASS=IN
    packet += b'\x00\x00\x01\x2c'  # TTL=300 seconds
    packet += b'\x00\x04'  # RDLENGTH=4 bytes
    packet += socket.inet_aton(self.ip_domain)

    return packet
```

5. Define a method named 'extract_domain' that extracts the domain name from the question section of a DNS query packet. The method does the following:

a. Initializes an empty string named 'domain' to store the domain name.

b. Sets the variable 'pos' to the starting position of the question section.

c. Iterates over the question section byte by byte, starting at position 'pos'. The first byte is the length of the first label (i.e., the first part of the domain name delimited by dots). The next 'length' bytes represent the label, which is decoded from ASCII to a string and appended to 'domain'. The process is repeated until the byte with value 0 is encountered, which marks the end of the domain name.

d. The trailing dot is removed from the domain name, and the method returns the resulting string.

```
def extract_domain(self,query_packet):
    """
    Given a DNS query packet, extract the domain name from the question section
    and return it.
    """
    domain = ''
    pos = 12
    while query_packet[pos] != 0:
        length = query_packet[pos]
        domain += query_packet[pos+1:pos+1+length].decode('ascii') + '.'
        pos += 1 + length
    domain = domain[:-1]  # remove the trailing dot
    return domain
```

6. In the main block of the code, the following steps are executed:

a. An instance of the DNSserver class named 'dns_server' is created.

b. The 'receive_dns_query' method of the 'dns_server' instance is called to start listening for incoming DNS queries and handling them.

```
if __name__ == '__main__':
    # create a DNS server instance
    dns_server = DNSserver()

    # listen for incoming DNS queries and handle them
    dns_server.recieve_dns_query()
```

1.3.2 Flowchart

Here's a simple flowchart that outlines the basic steps involved in a DNS lookup process between a DNS client and server:

1. DNS client sends a query for a domain name to the local DNS resolver.

2. If the local resolver has the IP address for the domain name in its cache, it returns the IP address to the client.

3. If the local resolver does not have the IP address in its cache, it sends the query to the root DNS server.

4. The root server responds with the IP address of the top-level domain server (such as .com or .org).

5. The local resolver sends the query to the top-level domain server.

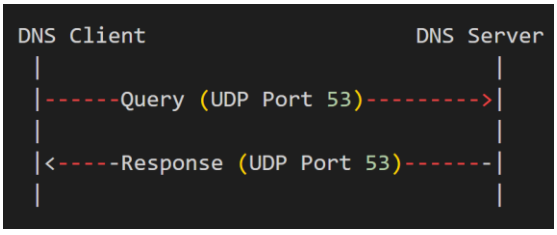
6. The top-level domain server responds with the IP address of the authoritative DNS server for the domain name (such as ns1.example.com).

7. The local resolver sends the query to the authoritative DNS server.

8. The authoritative DNS server responds with the IP address for the domain name.

9. The local resolver caches the IP address and returns it to the client.

Here's a simple diagram that illustrate the communication between a DNS server and a DNS client:



In this diagram, the DNS client sends a query to the DNS server using User Datagram Protocol (UDP) on port 53. The DNS server receives the query and processes it, then sends a response back to the client on UDP port 53. The client receives the response and uses the information provided to complete its request.

Note that in some cases, a DNS server may also communicate with other DNS servers to resolve a query. This communication typically occurs over Transmission Control Protocol (TCP) rather than UDP and may involve multiple rounds of querying and response exchanges. However, for the purposes of this diagram, we have simplified the communication to focus on the basic interaction between a DNS client and a DNS server.

1.4 HTTP APP

1.4.1 Code Description

HTTP – RUDP:

This is an implementation of a Reliable UDP (RUDP) HTTP server. The RUDP protocol is built on top of the User Datagram Protocol (UDP), but provides reliability features such as packet acknowledgement and retransmission. Our server uses the Cubic Congestion Control algorithm to manage the flow of packets and avoid network congestion. The server waits for a client to initiate a connection by sending a SYN packet. Once a connection is established, the server can send data packets to the client, and the client can send acknowledgement packets back to the server. The server uses a sliding window mechanism to manage the flow of packets and avoid congestion. The server can also receive file size information from the client, and can send requests to the client to ask for missing packets.

There is some explanation about our code:

1. deconstruct_packet(packet): This is a helper function that takes a packet as an argument and returns a dictionary with the packet's type, sequence number, source address, and data. It first unpacks the packet using struct.unpack() with the format string FORMAT, which is defined as !II and represents two unsigned integers. The first unsigned integer corresponds to the sequence number, and the second corresponds to the packet type. The function then returns a dictionary with the packet's information.

```
def deconstruct_packet(packet):
    """
    Helper function to deconstruct a packet into its constituent parts.
    :param packet: The packet to deconstruct
    :return: A dictionary with the packet's type, sequence number, source address, and data
    """
    # Unpacks the sequence number and packet type from the packet's header
    seq, packet_type = struct.unpack(FORMAT, packet[0][:HEADER_SIZE])
    # Returns a dictionary containing the packet's type, sequence number, source address, and data
    return {'type': packet_type, 'seq': seq, 'src_address': packet[1], 'data': packet[0][HEADER_SIZE:]}
```

2. class RUDPServer:

a. __init__(self, ip, port): This is the constructor method for the RUDPServer class. It creates a UDP socket using the socket module and sets several properties, including the socket's blocking mode, timeout value, and initial congestion window size. It also sets other properties, such as the outgoing sequence number, slow start threshold, and maximum window size.

```
def __init__(self, ip, port):
    # Creates a UDP socket
    self.sock = s.socket(s.AF_INET, s.SOCK_DGRAM)
    # Sets socket to non-blocking mode
    self.sock.setblocking(False)
    # Sets socket timeout to the default value
    self.sock.settimeout(TIMEOUT)
    # Dictionary for holding packets to be sent
    self.packets_to_send = {}
    # Dictionary for holding packets that have been sent but not yet acknowledged
    self.sent_items = {}
    # Sequence number for outgoing packets
    self.outgoing_seq = random.randint(0, (2 ** 16))
    # Initial congestion window size
    self.cwnd = 1
    # Maximum window size
    self.w_max = self.cwnd
    # Initial slow start threshold
    self.slow_start_threshold = 16
    # Flag for indicating whether we are currently in congestion avoidance phase
    self.congestion_avoidance = False
    # Timestamp for the last time we reduced the congestion window
    self.last_window_reduction = 0
    # Size of the file to be sent
    self.file_size = 0
    # Estimated round-trip time in seconds
    self.rtt = 0
    # Server IP address and port number
    self.server_address = ip, port
```

```
# Client IP address and port number
self.target_address = None
# Flag for indicating whether the server is currently connected to a client
self.connected = False
# Dictionary for holding requests
self.requests = {}
# Flag for indicating whether the file size information has been sent
self.file_info_sent = False
self.increment_seq = lambda: setattr(self, "outgoing_seq", self.outgoing_seq + 1)
```

b. bind(self): This method binds the socket to the specified address and port.

```
def bind(self):
    """
    Bind the socket to the specified address and port.
    """
    # Bind the socket to the server address specified in self.server_address
    self.sock.bind(self.server_address)
```

c. accept_connection(self): This method waits for a SYN packet from a client and responds with a SYN ACK packet to establish a connection. It uses the deconstruct_packet() helper function to extract the packet's information, and it sets the connected and target_address properties to indicate that the server is now connected to a client.

```
def accept_connection(self):
    # Loop until a connection is established with a client
    while not self.connected:
        try:
            # Receive a packet and extract its fields
            type, seq, address, data = deconstruct_packet(self.sock.recvfrom(CHUNK)).values()
            self.target_address = address
            # If the packet is a SYN packet, send a SYN-ACK packet back to the client
            if type == SYN:
                syn_ack_packet = struct.pack(FORMAT, self.outgoing_seq, SYN_ACK)
                bytes = self.sock.sendto(syn_ack_packet, self.target_address)
                self.increment_seq()
                if self.confirm_sent(bytes, syn_ack_packet):
                    self.connected = True
                    print(f"Connection established with client at IP address: {address[0]}")
                else:
                    print(f"Error sending message: {errno}")

                # Print a message to indicate that the connection was established
        except socket.timeout:
            # If a timeout occurs while waiting for a packet, continue the loop and try again
            continue
```

d. receive_packet(self): This method waits for and receives a packet from the client. It uses the deconstruct_packet() helper function to extract the packet's information and returns it.

```
def receive_packet(self):
    # Receive a packet and extract its fields, then return them
    type, seq, address, data = deconstruct_packet(self.sock.recvfrom(CHUNK)).values()
    return type, seq, address, data
```

e. **cubic_algo(self, T):** This method implements the Cubic Congestion Control algorithm to adjust the congestion window size. It takes a time T as an argument and returns the new congestion window size. The algorithm uses the maximum window size (w_max), a scaling factor (C), and a parameter (beta) to calculate the new window size.

```
def cubic_algo(self, T):
    """
    Implementation of the Cubic Congestion Control algorithm.
    :param T: Time elapsed since last window reduction
    :return: New congestion window size
    """
    # Calculate the scaling factor K based on the current window size and elapsed time since the last window reduction
    K = ((self.w_max * (1 - beta)) / C) ** (1 / 3)
    # Calculate the new congestion window size using the cubic function
    cwnd = C * ((T - K) ** 3) + self.w_max
    return cwnd
```

f. **send_data(self, address):** This method sends data packets to the client using the congestion control algorithm. It first checks if the server is in the slow start phase or the congestion avoidance phase and adjusts the congestion window size accordingly. If the new window size is smaller than the old window size, the method updates the slow start threshold and sets the last window reduction time. Finally, it sends the packet using the sendto() method and removes the packet from the packets_to_send dictionary.

```python
def send_data(self):
    all_packets_sent = False
    while self.packets_to_send and not all_packets_sent:
        # Congestion Control algorithm:

        # slow start
        if not self.congestion_avoidance:
            if self.cwnd >= self.slow_start_threshold:
                self.congestion_avoidance = True
            else:
                self.cwnd *= 2

        # congestion avoidance
        else:
            now = time.time()
            time_elapsed = now - self.last_window_reduction
            # calculate new congestion window size using cubic algorithm
            new_cwnd = self.cubic_algo(time_elapsed)

            if new_cwnd >= self.cwnd:
                self.cwnd = new_cwnd
            else:  # the new window is smaller than the old window
                self.w_max = self.cwnd
                self.last_window_reduction = time.time()
                self.slow_start_threshold = max(self.cwnd / 2, 1)
                self.cwnd = self.slow_start_threshold
                self.congestion_avoidance = False
```

```python
# send payload
# If file info has not been sent, send it first
if not self.file_info_sent:
    self.send_packet_count()

packets = []
first_seq_sent = min(self.packets_to_send)
for seq, data in self.packets_to_send.items():
    # don't send more packets than what the congestion window allows:
    if seq - first_seq_sent <= self.cwnd:
        packets.append(data)

time_of_sending = time.time()
# Send all the packets at once using the socket
if len(b''.join(packets)) > 0:
    sent = self.sock.sendto(b''.join(packets), self.target_address)
    if sent == len(b''.join(packets)):
        print(f"Packet sent successfully.\n")
    else:
        print(f"Error sending message: {errno}")
```

```
    # Receive acks
    try:
        while self.packets_to_send:

            packet_type, _, _, data = deconstruct_packet(self.sock.recvfrom(CHUNK)).values()

            if packet_type == FIN:
                all_packets_sent = True
                print("All packets sent successfully to client.\n")
                break

            if packet_type == ACK:

                time_of_ack = time.time()

                acked_seq = int(data.decode()[5:])
                if acked_seq in self.packets_to_send:  # if this is an acknowledgement packet for a
                    if acked_seq == first_seq_sent:
                        # calculate RTT
                        self.rtt = time_of_sending - time_of_ack
                        self.sock.settimeout(max(10, int(self.rtt // 2)))

                    # remove packet from packets_to_send and save in sent_items
                    self.sent_items[acked_seq] = self.packets_to_send.pop(acked_seq)

    except socket.timeout:
        continue

# close the connection
self.close_connection()
```

g. construct_payload method: This method takes the file data as input and constructs the data packets to be sent to the client. It iterates over the chunks of data and packs them into data packets along with sequence numbers.

```
def construct_payload(self, data):
    # Calculate the number of chunks needed to send the entire file
    seq = self.outgoing_seq
    for i in range((self.file_size + CHUNK) // CHUNK):
        # Pack the sequence number and packet type into a data packet
        data_packet = struct.pack(FORMAT, seq, DATA_PACKET)
        # Add a chunk of data to the packet
        data_packet += data[(i * CHUNK):((i + 1) * CHUNK)]
        # Store the packet in a dictionary, indexed by its sequence number
        self.packets_to_send[seq] = data_packet
        seq += 1
    # Update the outgoing sequence number to the next available number
    self.outgoing_seq = seq
```

h. send_packet_count method: This method sends a file size info packet to the client containing the number of packets to be sent. It then waits for an ACK packet from the client acknowledging the file size info packet.

```
def send_packet_count(self):
    seq = self.outgoing_seq
    # Increment the outgoing sequence number to use for the file size info packet
    self.increment_seq()
    # Pack the number of packets to send into a file size info packet
    packet_count = len(self.packets_to_send)
    file_size_info_packet = struct.pack(FORMAT, seq, FILE_SIZE_INFO)
    file_size_info_packet += f"Number of Packets: {packet_count}".encode()
    # Send the file size info packet to the target address and wait for a confirmation
    bytes = self.sock.sendto(file_size_info_packet, self.target_address)
    if self.confirm_sent(bytes, file_size_info_packet):
        print("File info sent successfully.\n")
        # Increment the sequence number for the outgoing packets and set the connection status to connected
    else:
        print(f"Error sending message: {errno}")

    # Wait for an acknowledgement that the file info packet was received
    attempts = 10
    while not self.file_info_sent and attempts > 0:
        try:
            type, _, _, data = self.receive_packet()
            acked_seq = int(data.decode()[5:])
            if type == ACK and acked_seq == seq:
                self.file_info_sent = True
                return
        except socket.timeout:
            attempts -= 1
            continue
    # If the file info packet was not acknowledged, close the connection with an error message
    self.close_connection(force=True, comment="Could not send file info")
```

i. close_connection method: This method sends a CLOSE_CONNECTION packet to the client and waits for an ACK packet acknowledging the CLOSE_CONNECTION packet. It then closes the server socket.

```
def close_connection(self, force=False, comment=""):
    # If the connection is not being forced closed...
    if not force:
        # Construct a FIN-ACK packet and send it
        seq = self.outgoing_seq
        packet = struct.pack(FORMAT, seq, FIN_ACK)
        self.sock.sendto(packet, self.target_address)
        # Increment the outgoing sequence number
        self.increment_seq()

        # Wait for an acknowledgement of the FIN-ACK packet
        attempts = 10
        while attempts > 0:
            try:
                packet_type, _, _, data = self.receive_packet()
                if packet_type == ACK:
                    acked_seq = int(data.decode()[5:])
                    if acked_seq == seq:
                        print("Closing the socket...\n")
                        break
            except socket.timeout:
                attempts -= 1

        force = not attempts

    # If the connection is being forced closed or the FIN-ACK packet was not acknowledged...
    if force:
        # Print an error message with a comment about the reason for the forced close
        print(f"Something went wrong. {comment}. Forcing disconnection with client...\n")

    # Close the socket and print a message
    self.sock.close()
    print("Socket closed.\n")
```

3. downloadmanager function: This is the main function that creates a RUDPServer object, binds it to a specific IP address and port, and waits for connection requests from clients. It listens for HTTP GET requests, extracts the file name and host name from the request, constructs a URL from these values, and sends an HTTP GET request to retrieve the file data. It then constructs the data packets to be sent to the client and sends them using the send_data method of the RUDPServer object. Finally, it waits for more HTTP GET requests from clients and repeats the process until no requests are received for a specified amount of time. When no requests are received for the specified amount of time, it closes the connection with the client using the close_connection method.

```python
def download_manager():

    rudp_s = RUDPServer(IP, PORT)
    rudp_s.bind()
    print("Ready to serve...\n")

    rudp_s.accept_connection()

    # Create a RUDPServer object with the defined IP and port number
    rudp_s = RUDPServer(IP, PORT)
    rudp_s.bind()
    print("Ready to serve...\n")

    # Accept incoming connection requests
    rudp_s.accept_connection()
```

```python
while True:
    try:
        # Receive a packet from the client
        t, seq, address, data = rudp_s.receive_packet()
        print("Request packet received...\n")

        # Extract the file name and host name from the request data
        print("Extracting URL...")
        request_string = data.decode()
        request_lines = request_string.split("\r\n")
        file_name = request_lines[0][5: -9]
        host_name = request_lines[1][6:]

        # Construct the URL for the HTTP GET request
        url = f"http://{host_name}/{file_name}"
        print(f"URL for http GET request: {url}.\n")

        # Send an HTTP GET request to the server
        print("Sending HTTP GET request...\n")
        response = requests.get(url)

        # Retry up to 10 times if the GET request fails
        request_received = False
        for attempt in range(10):
            if response.status_code >= 200 and response.status_code < 300:
                print("GET request successful.\n")
                request_received = True
                break
```

```
            # If the GET request still fails, print an error message and return
            if not request_received:
                print(f"GET request failed with status code {response.status_code}.\n")
                return

            # Retrieve the file data from the response
            print(f"Getting file: {file_name} from: {host_name}...\n")
            print("Retreiving file data...\n")
            data = response.content

            # Set the file size in the RUDPServer object and construct the payload
            rudp_s.file_size = len(data)
            print("Preparing file for download...\n")
            rudp_s.construct_payload(data)

            # Send the data packets to the client using RUDP
            print("Downloading file...\n")
            rudp_s.send_data()

            # Print a success message and return
            print("Download completed successfully!\n")
            return

        # Ignore socket.timeout exceptions and continue
        except socket.timeout:
            pass
```

HTTP – TCP:

Class TCPServer: a simple implementation of the main functions needed for running TCP based server operations, encapsulated in one class.

```
class TCPServer:
    """
    A simple TCP server implementation
    """
    def __init__(self, ip, port):
        # Creates a TCP socket
        self.sock = s.socket(s.AF_INET, s.SOCK_STREAM)
        self.address = ip, port

    def bind(self):
        self.sock.setsockopt(s.SOL_SOCKET, s.SO_REUSEADDR, 1)
        self.sock.bind(self.address)

    def socket(self):
        return self.sock
```

The download_manager() function: the main function for running the script for all binding the server port and server ip to the server's socket, accepting the connection from the client to interact with the client over the TCP transmission.

```
def download_manager():

    tcp_s = TCPServer(IP, PORT)

    tcp_s.bind()

    tcp_s.socket().listen()

    print("Listening...")

    connection, address = tcp_s.socket().accept()

    print("Connection with client established...\n")

    request = connection.recv(CHUNK)

    print(request)
```

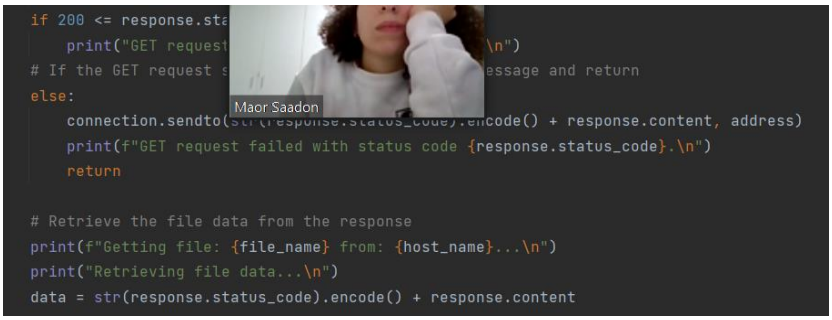
The function proceeds to receive the HTTP GET request from the client side parse the URL of the file from it.

```
request = connection.recv(CHUNK)

print(request)

print("Got HTTP GET request from client")

# Extract the file name and host name from the request data
print("Extracting URL...")
request_string = request.decode()
request_lines = request_string.split("\r\n")
```

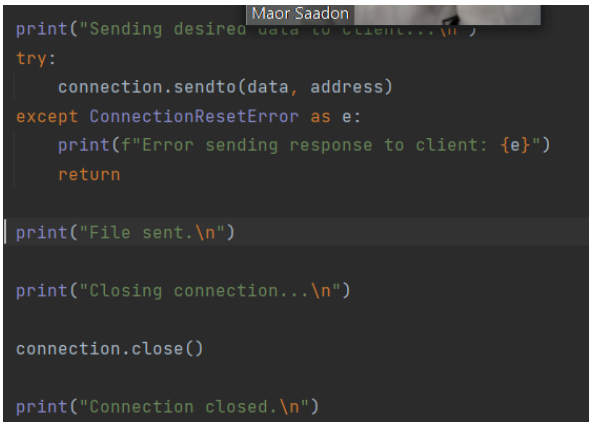
Constructing URL and sending HTTP get request using python's "requests" library:

```
# Construct the URL for the HTTP GET request
url = f"http://{host_name}/{file_name}"
print(f"URL for http GET request: {url}.\n")

# Redirect HTTP GET request to the server
print("Redirecting HTTP GET request...\n")
try:
    response = requests.get(url)
```

Checking status code of the response and retrieving the data:



Sending the file to the client and closing the connection:



1.4.2 Flowchart

RUDP Protocol:

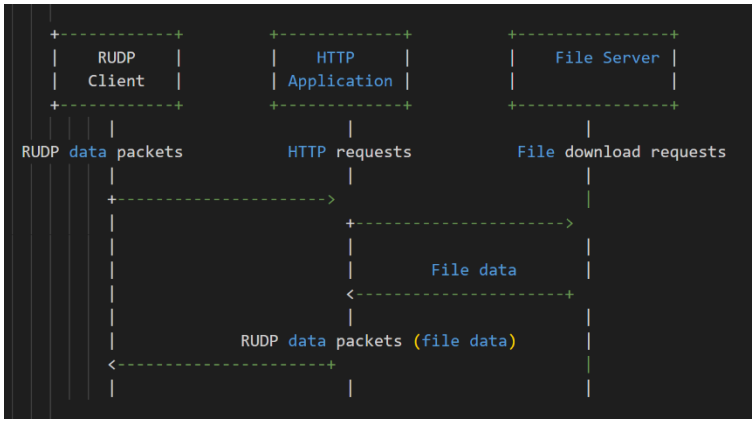
1. The client sends a data packet using the RUDP protocol to the HTTP application, which contains the request for an HTTP resource.

2. The HTTP application receives the RUDP packet from the client and processes the request.

3. The HTTP application sends a request to the file server to download the requested file.

4. The file server sends the requested file as a series of RUDP data packets to the HTTP application.

5. The HTTP application receives the RUDP data packets from the file server and sends them to the client as RUDP packets.



TCP Protocol:

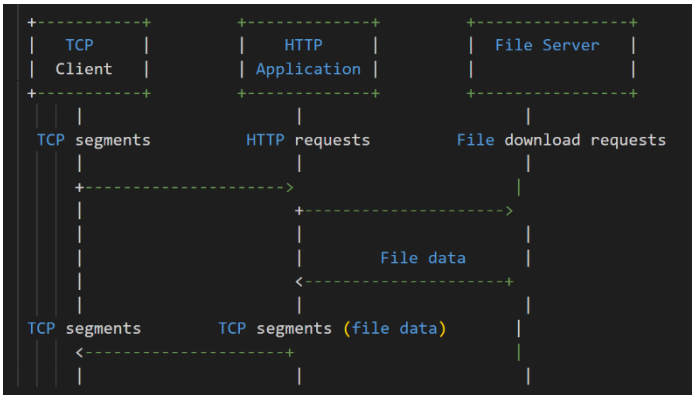
1. The client sends a TCP segment to the HTTP application containing the request for an HTTP resource.

2. The HTTP application receives the TCP segment from the client and processes the request.

3. The HTTP application sends a request to the file server to download the requested file.

4. The file server sends the requested file as a series of TCP segments to the HTTP application.

5. The HTTP application receives the TCP segments from the file server and sends them to the client as TCP segments.



In both cases, the client initiates the request and sends it to the HTTP application. The HTTP application processes the request and communicates with the file server to download the requested file. The file server sends the file back to the HTTP application, which then sends it back to the client. The main difference between RUDP and TCP protocols is in the way the data packets are transmitted, with RUDP providing a simpler, faster transmission method at the cost of reliability, while TCP provides a more reliable transmission method at the cost of increased complexity and potentially slower transmission speeds.

1.5 Client

1.5.1 Code Description

This code is distributed to several different clients:

1. Class DHCPClient: This class represents a DHCP client that sends Discover and Request packets to a DHCP server to obtain an IP address.

a. The init function initializes the client's MAC address and sets the IP address and DNS server IP address to None.

```
def __init__(self):
    # Get the MAC address of the interface being used
    self.mac_address = get_if_hwaddr(conf.iface)
    # Initialize a variable to hold the IP address assigned by DHCP (not yet known)
    self.ip_address = None
    self.DNSserver_ip = None
```

b. The handle_offer_packet function is called when the client receives an offer packet from a DHCP server. It sends a request packet in response to the offer.

```
def handle_offer_packet(self, packet):
    # Print a message indicating that an offer packet was received
    print("DHCP offer received")
    # Send a DHCP request packet in response to the offer
    self.send_request_packet(packet)
```

c. The handle_ack_packet function is called when the client receives an ACK packet from a DHCP server. It extracts the DNS server IP address from the packet and sets it as a class variable.

```
def handle_ack_packet(self, ack):
    print("DHCP ack received")
    self.DNSserver_ip = ack[DHCP].options[5][1]
```

d. The send_discover_packet function constructs and sends a DHCP Discover packet over the network, and then waits for a DHCP offer packet to arrive. When a packet is received, it calls handle_offer_packet to send a DHCP request packet.

```
def send_discover_packet(self):
    # Construct a DHCP Discover packet using the current MAC address as the client identifier
    discover = Ether(src=self.mac_address, dst='ff:ff:ff:ff:ff:ff') / \
               IP(src='0.0.0.0', dst='255.255.255.255') / \
               UDP(sport=68, dport=67) / \
               BOOTP(op=1, chaddr=self.mac_address) / \
               DHCP(options=[('message-type', 'discover'), 'end', 'pad'])
    # Print a message indicating that a Discover packet is being sent
    print("sending DHCP Discover")
    # Send the Discover packet over the network
    sendp(discover)
    # Sniff the network for incoming DHCP offer packets and call handle_offer_packet() when one is received
    sniff(prn=self.handle_offer_packet, filter='(port 67 or port 68)', count=1)
```

e. The send_request_packet function constructs and sends a DHCP Request packet to the DHCP server, and then waits for a DHCP ACK packet to arrive. When a packet is received, it calls handle_ack_packet to extract the DNS server IP address.

```
def send_request_packet(self, offer):
    # Extract the offered IP address from the offer packet and construct a DHCP Request packet
    ip_address = offer[BOOTP].yiaddr
    request = Ether(src=self.mac_address, dst='ff:ff:ff:ff:ff:ff') / \
              IP(src='0.0.0.0', dst='255.255.255.255') / \
              UDP(sport=68, dport=67) / \
              BOOTP(op=1, chaddr=self.mac_address, yiaddr=ip_address, siaddr=get_if_addr(conf.iface)) / \
              DHCP(options=[('message-type', 'request'),
                            ('requested_addr', ip_address),
                            ('server_id', offer[DHCP].options[1][1]), 'end', 'pad'])
    # Wait for 1 second before sending the request packet
    time.sleep(1)
    # Print a message indicating that a Request packet is being sent
    print("sending DHCP Request")
    # Send the Request packet over the network
    sendp(request)
    # Sniff the network for incoming DHCP ACK packets and call handle_ack_packet() when one is received
    sniff(prn=self.handle_ack_packet, filter='(port 67 or port 68)', count=1)
```

2. Class DNSClient : This class is a simple client that can be used to query a DNS server and obtain the IP address for a given domain name. It contains two functions:

```
def __init__(self):
    # Set the IP address and port number for the DNS server
    self.server_address = ('127.0.0.1', 53)
    self.domain_ip = None
```

a. query(domain): This function takes a domain name as input and returns the corresponding IP address by sending a DNS query packet to the DNS server and parsing the response. It first creates a UDP socket and constructs a DNS query packet using the create_dns_packet() function. It then sends the packet to the DNS server using the sendto() method and waits for a response using the recvfrom() method. It parses the response packet by extracting the header and answer sections, and then extracts the IP address from the answer section. Finally, it closes the socket and returns the IP address.

```
def query(self, domain):
    # create a UDP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # construct the DNS query packet
    packet = self.create_dns_packet(domain)

    # send the packet to the DNS server
    client_socket.sendto(packet, self.server_address)

    # receive the response from the DNS server
    response, server_address = client_socket.recvfrom(4096)

    # parse the DNS response packet
    header = response[:12]
    answer_section = response[12 + len(packet):]

    # extract the IP address from the answer section
    ip_bytes = answer_section[-4:]
    self.domain_ip = socket.inet_ntoa(ip_bytes)

    # close the socket
    client_socket.close()

    return self.domain_ip
```

b. create_dns_packet(domain): This function takes a domain name as input and constructs a DNS query packet. It first initializes an empty byte string packet, and then adds the various fields of the DNS header, such as the ID, QR flag, opcode, question count, answer count, and so on. It then constructs the question section of the packet by splitting the domain name into its component parts, encoding each part as ASCII and adding it to the packet, along with the appropriate length and termination bytes. Finally, it adds the QTYPE and QCLASS fields to the packet and returns it.

```python
def create_dns_packet(self, domain):
    # construct the DNS query packet
    packet = b''
    packet += b'\x00\x01'  # ID
    packet += b'\x01\x00'  # QR=0, Opcode=0, AA=0, TC=0, RD=1
    packet += b'\x00\x01'  # QDCOUNT (1 question)
    packet += b'\x00\x00'  # ANCOUNT
    packet += b'\x00\x00'  # NSCOUNT
    packet += b'\x00\x00'  # ARCOUNT
    # construct the question section
    parts = domain.split('.')
    for part in parts:
        packet += bytes([len(part)])
        packet += part.encode('ascii')
    packet += b'\x00'  # end of domain name
    packet += b'\x00\x01'  # QTYPE=A
    packet += b'\x00\x01'  # QCLASS=IN

    return packet
```

3. deconstruct_packet(packet): This static function takes in a packet and returns a dictionary containing the packet type, sequence number, source address, and data. It uses the struct module to unpack the data from the packet.

```python
def deconstruct_packet(packet):
    """
    Helper function to deconstruct a packet into its constituent parts.
    :param packet: The packet to deconstruct
    :return: A dictionary with the packet's type, sequence number, source address, and data
    """
    # Unpacks the sequence number and packet type from the packet's header
    seq, packet_type = struct.unpack(FORMAT, packet[0][:HEADER_SIZE])
    # Returns a dictionary containing the packet's type, sequence number, source address, and data
    return {'type': packet_type, 'seq': seq, 'src_address': packet[1], 'data': packet[0][HEADER_SIZE:]}
```

4. class RUDPClient:

a. __init__() : This function is the constructor for the RUDPClient class. It initializes the following instance variables:

self.sock: a socket object for communication.

self.received_packets: a dictionary to hold received data payloads.

self.outgoing_seq: a sequence number for outgoing packets.

self.server_address: the server's address in the form of a tuple (ip, port).

self.all_data_received: a flag indicating whether all data has been received.

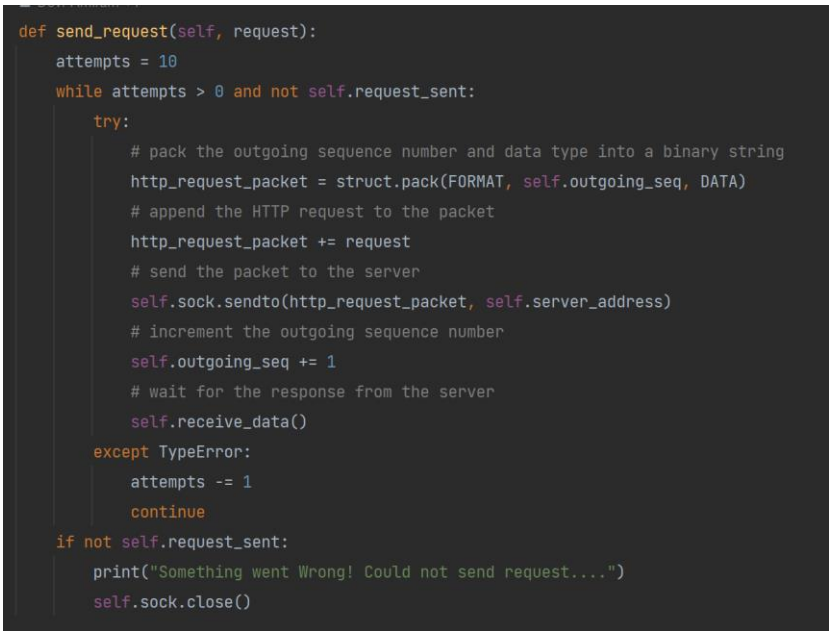
self.request_sent: a flag indicating whether a request has been sent.

```python
def __init__(self):
    """
    Constructor for the ReliableUDP class
    :param host_adress: port and ip
    """
    # Create a UDP socket
    self.sock = s.socket(s.AF_INET, s.SOCK_DGRAM)
    # Set socket to non-blocking mode
    self.sock.setblocking(False)
    # Set socket timeout to 1 second
    self.sock.settimeout(TIMEOUT)
    # Dictionary for holding received data payloads
    self.received_packets = {}
    # Sequence number for outgoing packets
    self.outgoing_seq = random.randint(0, (2 ** 16))
    # Tuple: (ip, port)
    self.server_address = None
    self.all_data_received = False
    self.request_sent = False
    self.request_accepted = False
    self.connected = False
```

b. connect(server_ip, server_port): This function attempts to establish a connection with the server. It does this by creating a SYN packet and sending it to the server. It then waits for a SYN-ACK response from the server, sends an ACK packet in response, and returns True if the connection is established. If it fails to establish a connection within ATTEMPT_LIMIT attempts, it returns False.

```python
def connect(self, server_ip, server_port):
    """
    Connects to the server with the specified ip and port
    :param server_ip: the ip address of the server
    :param server_port: the port number of the server
    :return: True if connection was successful, False otherwise
    """
    self.server_address = server_ip, server_port

    attempts = 10
    for i in range(attempts):
        # create a SYN packet
        syn_packet = struct.pack(FORMAT, self.outgoing_seq, SYN)
        # send the SYN packet to the server
        self.sock.sendto(syn_packet, (server_ip, server_port))

        time.sleep(1)  # 2 seconds grace period

        print(f"Connection: attempt: {i + 1}")
        # receive syn ack
        try:
            # wait for SYN-ACK response from server
            type, seq, address, data = deconstruct_packet(self.sock.recvfrom(CHUNK)).values()

            # verify that the packet is a SYN-ACK packet
            if type == SYN_ACK:
                # send ACK packet to server
                print("Connection with server established...\n")
                return True
        except socket.timeout:
            print("Somthing went wrong. trying again...\n")
    self.server_address = None
    return False
```

c. .send_request(request): This function sends an HTTP request to the server. It does this by creating a REQUEST packet containing the request and sending it to the server. It then waits for a response from the server by calling self.receive_data(). If it fails to send the request after 10 attempts, it closes the socket and prints an error message.



```
def send_request(self, request):
    attempts = 10
    while attempts > 0 and not self.request_sent:
        try:
            # pack the outgoing sequence number and data type into a binary string
            http_request_packet = struct.pack(FORMAT, self.outgoing_seq, DATA)
            # append the HTTP request to the packet
            http_request_packet += request
            # send the packet to the server
            self.sock.sendto(http_request_packet, self.server_address)
            # increment the outgoing sequence number
            self.outgoing_seq += 1
            # wait for the response from the server
            self.receive_data()
        except TypeError:
            attempts -= 1
            continue
    if not self.request_sent:
        print("Something went Wrong! Could not send request....")
        self.sock.close()
```

d. receive_data(): This function receives data from the server. It loops until all data has been received. It receives packets from the server, verifies the packet type, and stores the payload in the self.received_packets dictionary. If the packet is a DATA_PACKET, it sends an ACK packet in response. If the packet is a FILE_SIZE_INFO packet, it stores the total number of packets to be received. If the packet is a CLOSE_CONNECTION packet, it sets self.all_data_received to True and breaks out of the loop. If the packet is a REQUEST_ACK packet, it sets self.request_sent to True. If a timeout occurs, it continues looping until all data has been received.

```python
def receive_data(self):
    # set the number of packets to be received to infinity
    packets_to_be_received = float('inf')
    while not self.all_data_received:
        try:
            # receive a packet from the server
            type, seq, address, data = deconstruct_packet(self.sock.recvfrom(CHUNK)).values()
            # if the packet type is INFO, store the number of packets to be received
            if type == INFO:
                self.received_packets[seq] = {'src address': address, 'data': data}
                packets_to_be_received = int(data.decode()[19:]) - len(self.received_packets)
                print(f"Received file size info. Number of packets to be downloaded is: {packets_to_be_received}\n")
                # send an ACK packet to acknowledge the receipt of the INFO packet
                self.ack(seq)
            # if the packet type is DATA, store the packet and send an ACK packet
            if type == DATA:
                self.received_packets[seq] = {'src address': address, 'data': data}
                self.ack(seq)
                packets_to_be_received -= 1
                print(f"Downloaded packet - {seq} : {data}\n")
            # if the packet type is ACK, store the packet
            if type == ACK:
                self.received_packets[seq] = {'src address': address, 'data': b'REQUEST ACK'}
            # if all packets have been received and the packet type is FIN_ACK, send a FIN_ACK packet and close the socket
            if packets_to_be_received == 0 and type == FIN_ACK:
                self.received_packets[seq] = {'src address': address, 'data': b'FIN'}
                self.ack(seq)
                fin_ack_packet = struct.pack(FORMAT, (self.outgoing_seq - 1), FIN_ACK)
                self.sock.sendto(fin_ack_packet, address)
                self.all_data_received = True
                self.sock.close()
                break
```

```python
# if a packet is not received within the timeout period, and all packets have been received, send a FIN packet
except socket.timeout:
    if packets_to_be_received == 0:
        fin_packet = struct.pack(FORMAT, self.outgoing_seq, FIN)
        # send the FIN packet to the server
        self.sock.sendto(fin_packet, self.server_address)
```

7. ack(seq): This function sends an ACK packet in response to a received packet. It creates an ACK packet containing the sequence number of the packet to acknowledge, sends it to the source address of the packet, and increments self.outgoing_seq.

```
def ack(self, seq):
    """
    Sends an acknowledgement packet for a specified sequence number to the src address
    :param seq: the sequence number of the packet to acknowledge
    """
    # pack the outgoing sequence number and packet type into a binary string
    ack = struct.pack(FORMAT, self.outgoing_seq, ACK)
    # append the sequence number of the packet being acknowledged to the ACK packet
    ack += f"ACK: {seq}".encode()
    # send the ACK packet to the source address of the packet being acknowledged
    self.sock.sendto(ack, self.received_packets[seq]['src address'])
    # increment the outgoing sequence number
    self.increment_seq()
    # print a message to indicate that an ACK packet has been sent for the specified sequence number
    print(f"Sent ACK for seq: {seq}\n")
```

5. client_request(url, file_name) function: This is the main function that performs the whole process of sending a DHCP request to obtain an IP address, querying a DNS server for the IP address of the downloadmanager.com domain, establishing a connection with the application server using RUDP protocol, sending a GET request for a specific file, receiving and downloading the file, and saving it to a file with the given file name.

a. dhcp_client.send_discover_packet() function: This function sends a DHCP discovery packet to the network to initiate the process of obtaining an IP address. DHCP (Dynamic Host Configuration Protocol) is used to dynamically assign IP addresses to devices on a network.

```
# Create a DHCPClient object
dhcp_client = DHCPClient()

# Call the send_discover_packet() function to initiate the DHCP process
dhcp_client.send_discover_packet()

dns_ip = dhcp_client.DNSserver_ip
```

b. dns_client.query(domain_name) function: This function sends a DNS (Domain Name System) query to the DNS server to obtain the IP address of the given domain name. DNS is used to translate domain names into IP addresses that can be used to establish network connections.

```
# Create a DNSClient object
dns_client = DNSClient()

# Query the DNS server for the IP address of downloadmanager.com
app_server_ip = dns_client.query("downloadmanager.com")
print('http app domain: downloadmanager.com, http app ip: ' + app_server_ip)
```

c. rudp_c.connect(app_server_ip, port_number) function: This function establishes a connection with the application server using RUDP (Reliable UDP) protocol, which is a protocol that provides reliable data transfer over an unreliable network. It takes the IP address of the application server and a port number as input parameters.

d. rudp_c.send_request(request_data) function: This function sends an HTTP GET request to the application server to request a specific file. The request data is passed as a byte array.

f. rudp_c.receive_data() function: This function receives the response data from the application server and downloads the file. It collects all the packets received from the server, reorders them if necessary, and saves the file to disk.

```
# Create a RUDPClient instance and connect to the application server
rudp_c = RUDPClient()
rudp_c.connect(app_server_ip, 20000 + DOVI_LAST3_ID_DIG)

# Encode the HTTP GET request for the specified file and send it using RUDP
http_request = f"GET /{file_name} HTTP/1.1\r\nHost: {url}\r\n\r\n".encode()
rudp_c.send_request(http_request)

# Check if the request was sent successfully, and start waiting for data
if not rudp_c.request_sent:
    return
print("Preparing to download file...\n")
rudp_c.receive_data()

# Check if all data packets have been received, and write the file to disk
if rudp_c.all_data_received:
    print("File downloaded. Preparing file.\n")
    data = b''
    # Sort the received packets by sequence number and concatenate their payload
    packets = sorted(rudp_c.received_packets.items(), key=lambda item: item[0])
    for i in range(len(packets)):
        data += packets[i][1]['data']
    output = data.decode('utf-8')
    print("Saving file...\n")
    with open(file_name, 'w') as f:
        f.write(output)
    print("File successfully saved!\n")
    return
else:
    print("Something went wrong!")
    return
```

Each of these functions performs a specific task that is essential to the overall process of requesting and downloading a file from an application server.

6. Class TCPClient: sends an HTTP GET request to a specified server over TCP and saves the response to a file. The function takes in three arguments:

- **url**: A string representing the URL of the server
- **file_name**: A string representing the name of the file to be saved

In the function, a TCP client object is created. The client connects to the server using the IP address and a specified port number:

```
def TCP_request(url, file_name, app_server_ip):
    tcp_c = TCPClient()

    # Connect to the server
    tcp_c.connect(app_server_ip, 20000 + DOVI_LAST3_ID_DIG)
```

The function constructs an HTTP GET request using the specified file name and URL, and sends it to the server:

```
# Construct the HTTP request
http_request = f"GET /{file_name} HTTP/1.1\r\nHost: {url}\r\n\r\n".encode()

# Send the HTTP request to the server
try:
    tcp_c.socket().sendall(http_request)
    print("Request sent.\n")
except ConnectionResetError as e:
    print(f"Error sending request to server: {e}")
    return
```

The function receives the HTTP response from the server in chunks until there is no more data to be received:

```
# Receive the HTTP response from the server
data = b''
while True:
    try:
        chunk = tcp_c.socket().recv(CHUNK)
    except ConnectionResetError as e:
        print(f"Error receiving response from server: {e}")
        return
    if not chunk:
        break
    data += chunk
```

The response is then parsed to extract the HTTP response status code If the status code is 200, the function saves the response body to a file. If the status code is not 200, an error message is printed and the function returns without saving the file.

```
if response_status_code == 200:
    print("HTTP request successful.\n")
else:
    print(f"HTTP request failed with status code {response_status_code}\n")
    return

# Save the file
print("Saving file...\n")
output = data.decode('utf-8')[3:]
with open(file_name, 'w') as f:
    f.write(output)

print("File successfully saved!\n")
return
```

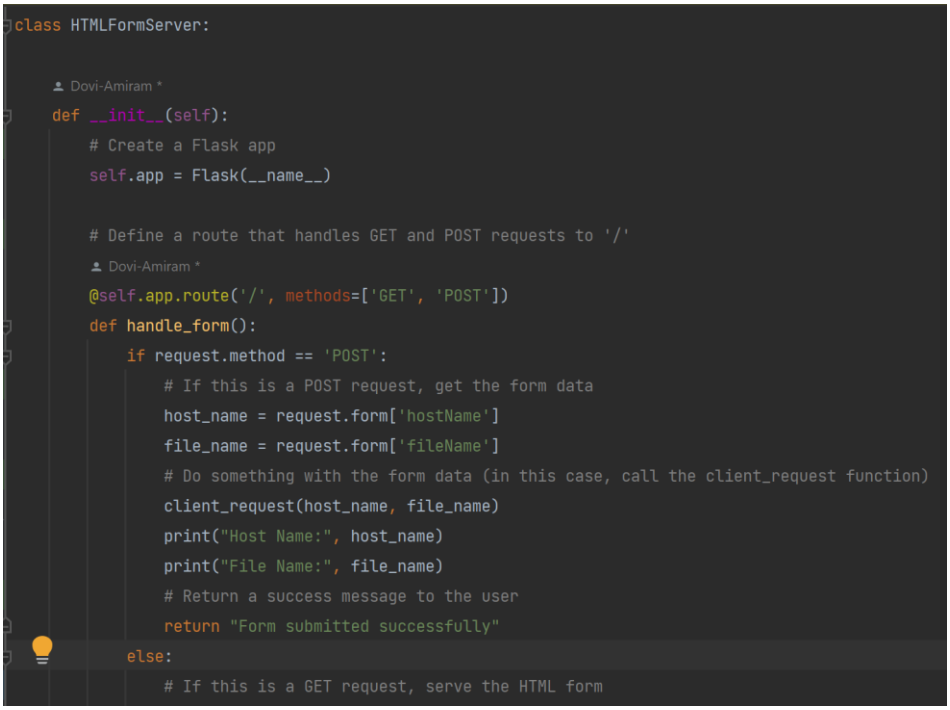
7. Class HTMLFormServer: It uses the Flask web framework to create a web server that serves an HTML form to clients and handles their form submissions.

The **__init__** method of the class initializes the Flask app and creates a route for handling form submissions. The route is specified as '/', which is the root path of the server. It supports both GET and POST requests.

When the server receives a POST request, it extracts the values of the hostName and fileName form fields and passes them to the client_request function. It then prints the values to the console and returns a "Form submitted successfully" message to the client.

When the server receives a GET request, it serves an HTML file that contains a form with two fields: hostName and fileName. The form is submitted using the POST method when the user clicks the "Submit Request" button.

The **run** method of the class starts the Flask app and listens for incoming requests on the specified host and port. By default, the host is set to 'localhost' and the port is set to 5000.



```
class HTMLFormServer:

    Dovi-Amiram *
    def __init__(self):
        # Create a Flask app
        self.app = Flask(__name__)

        # Define a route that handles GET and POST requests to '/'
        Dovi-Amiram *
        @self.app.route('/', methods=['GET', 'POST'])
        def handle_form():
            if request.method == 'POST':
                # If this is a POST request, get the form data
                host_name = request.form['hostName']
                file_name = request.form['fileName']
                # Do something with the form data (in this case, call the client_request function)
                client_request(host_name, file_name)
                print("Host Name:", host_name)
                print("File Name:", file_name)
                # Return a success message to the user
                return "Form submitted successfully"
            else:
                # If this is a GET request, serve the HTML form
```

```
        else:
            # If this is a GET request, serve the HTML form
            return '''
                <!DOCTYPE html>
                <html>
                  <head>
                    <meta charset="UTF-8">
                    <title>Web App</title>
                  </head>
                  <body>
                    <form id="myForm" method="post">
                      <label for="hostName">Host Name:</label>
                      <input type="text" id="hostName" name="hostName"><br><br>
                      <label for="fileName">File Name:</label>
                      <input type="text" id="fileName" name="fileName"><br><br>
                      <input type="submit" value="Submit Request">
                    </form>
                  </body>
                </html>
            '''

Dovi-Amiram *
def run(self, host='localhost', port=5000):
    # Start the Flask app
    self.app.run(host=host, port=port)
```

1.6 Wireshark

Rudp :

0% packet lost :

1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	2.075790925	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
3	3.136641420	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
4	5.167803990	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0
5	5.258087690	127.0.0.1	127.0.0.1	DNS	81	Standard query 0x0001 A downloadmanager.com
6	5.259018410	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0x0001 A downloadmanager.com A 127.
7	5.259240139	127.0.0.1	127.0.0.1	UDP	52	47800 → 20494 Len=8
8	5.261203428	127.0.0.1	127.0.0.1	UDP	52	20494 → 47800 Len=8
9	6.267978264	127.0.0.1	127.0.0.1	UDP	102	47800 → 20494 Len=58
10	11.338269661	10.0.2.15	172.217.22.36	HTTP	211	GET /index.html HTTP/1.1
11	11.836475113	172.217.22.36	10.0.2.15	HTTP	422	Continuation
12	11.837643989	127.0.0.1	127.0.0.1	UDP	72	20494 → 47800 Len=28
13	11.838013127	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
14	11.838162176	127.0.0.1	127.0.0.1	UDP	6212	20494 → 47800 Len=6168
15	11.838375401	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
16	21.850315738	127.0.0.1	127.0.0.1	UDP	10324	20494 → 47800 Len=10280
17	21.850592619	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
18	31.858315414	127.0.0.1	127.0.0.1	UDP	12573	20494 → 47800 Len=12529
19	31.858474513	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
20	41.867997221	127.0.0.1	127.0.0.1	UDP	10517	20494 → 47800 Len=10473
21	41.868185664	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
22	51.880823273	127.0.0.1	127.0.0.1	UDP	8461	20494 → 47800 Len=8417
23	51.881005690	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
24	61.891065889	127.0.0.1	127.0.0.1	UDP	6405	20494 → 47800 Len=6361
25	61.891397822	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
26	71.892992721	127.0.0.1	127.0.0.1	UDP	4349	20494 → 47800 Len=4305
27	71.893354799	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
28	81.905239178	127.0.0.1	127.0.0.1	UDP	2293	20494 → 47800 Len=2249
29	83.910242291	127.0.0.1	127.0.0.1	UDP	52	47800 → 20494 Len=8
30	83.910671926	127.0.0.1	127.0.0.1	UDP	52	20494 → 47800 Len=8
31	83.911165373	127.0.0.1	127.0.0.1	UDP	62	47800 → 20494 Len=18
32	83.911198983	127.0.0.1	127.0.0.1	UDP	52	47800 → 20494 Len=8

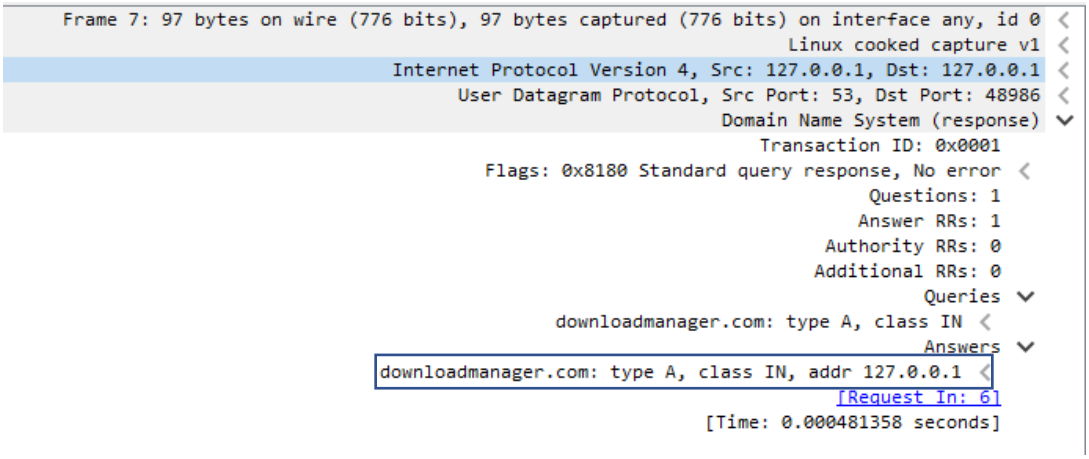
In lines 1-4 we can see the DORA - DHCP process like we explain above.

Inside the ACK packet we can see our new IP:

```
Frame 5: 313 bytes on wire (2504 bits), 313 bytes captured (2504 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 255.255.255.255
User Datagram Protocol, Src Port: 67, Dst Port: 68
Dynamic Host Configuration Protocol (ACK)
    Message type: Unknown (5)
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x00000000
    Seconds elapsed: 0
    Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0
    Your (client) IP address: 192.168.0.100
    Next server IP address: 10.0.2.15
    Relay agent IP address: 0.0.0.0
    Client MAC address: 30:38:3a:30:30:3a (30:38:3a:30:30:3a)
    Client hardware address padding: 32373a63613a31353a37
    Server host name not given
    Boot file name not given
    Magic cookie: DHCP
```

In lines 5-6 we can see the connection with the DNS server.

We can see in the query response the answer :



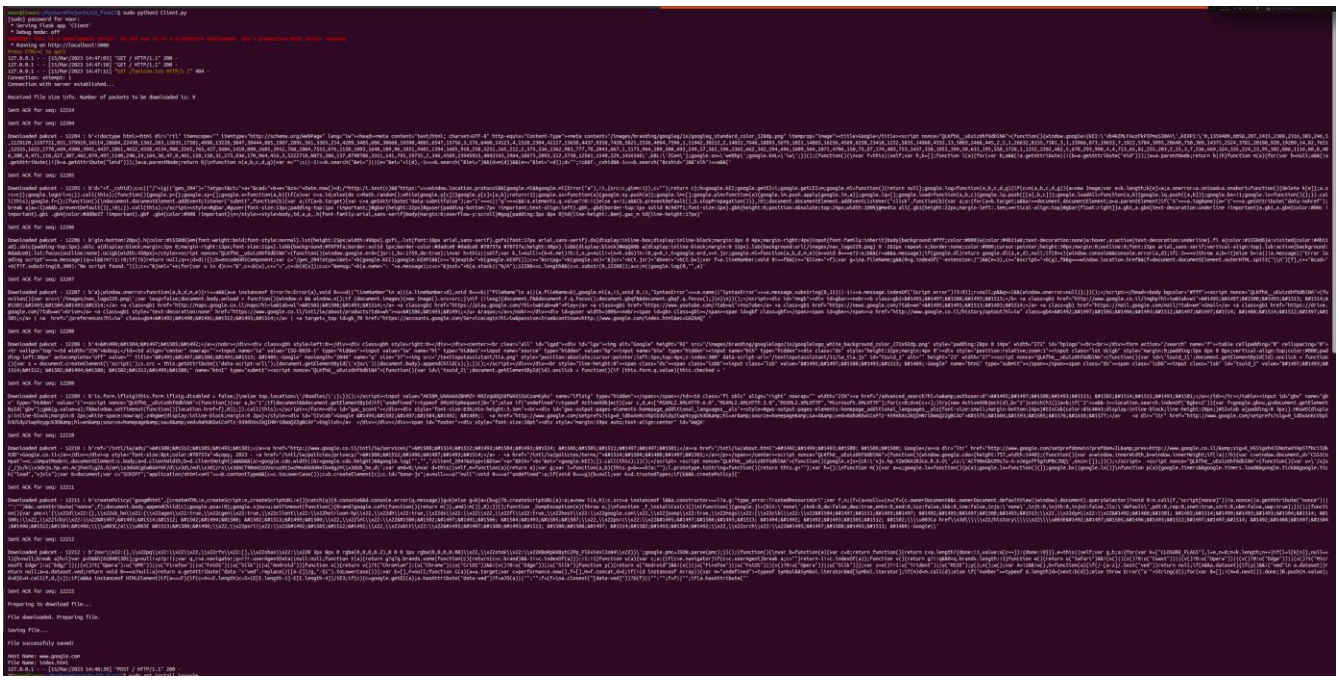
In lines 7-9 this is the open connection between the app server and the client. – we can see that the packet len = 8.

In lines 10 -11 we can see the http redirect to another server.

In lines 12 – 28 we can see the closing connection between the app server and the client. – we can see that the packet len = 8.

Terminal :

Client :



HTTP Server:


maor@linux:~/PycharmProjects/CN_finall$ sudo python3 RUDPDownloadManager.py
Ready to serve...
Connection established with client at IP address: 127.0.0.1
Request packet received...
Extracting URL...
URL for http GET request: http://www.google.com/index.html
Sending HTTP GET request...
GET request was successful!
Getting file: index.html from: www.google.com...
Retreiving file data...
Preparing file for download...
Downloading file...
File info sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
Packet sent successfully.
All packets sent successfully to client.
Something went wrong. . Forcing disconnection with client...
Socket closed.
Download completed successfully!
maor@linux:~/PycharmProjects/CN_finall$

10% packet lost:

1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	0.592358434	127.0.0.53	127.0.0.1	DNS	89	Standard query response 0x1d96 AAAA daisy.ubuntu.com OPT
3	2.119617379	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
4	3.214429414	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
5	5.308650125	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0
6	5.346471179	127.0.0.1	127.0.0.1	DNS	81	Standard query 0x0001 A downloadmanager.com
7	5.346720600	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0x0001 A downloadmanager.com A 127.
8	5.347003597	127.0.0.1	127.0.0.1	UDP	52	32878 → 20494 Len=8
9	5.347481476	127.0.0.1	127.0.0.1	UDP	52	20494 → 32878 Len=8
10	6.367163138	127.0.0.1	127.0.0.1	UDP	102	32878 → 20494 Len=58
11	6.624698756	10.0.2.15	142.250.184.100	HTTP	211	GET /index.html HTTP/1.1
12	7.086178142	142.250.184.100	10.0.2.15	HTTP	439	Continuation
13	7.087160567	127.0.0.1	127.0.0.1	UDP	72	20494 → 32878 Len=28
14	7.087271447	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
15	7.087333333	127.0.0.1	127.0.0.1	UDP	6212	20494 → 32878 Len=6168
16	7.087399624	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
17	17.092640675	127.0.0.1	127.0.0.1	UDP	10324	20494 → 32878 Len=10280
18	17.094628441	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
19	27.110152086	127.0.0.1	127.0.0.1	UDP	12610	20494 → 32878 Len=12566
20	27.111861333	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
21	47.126885927	127.0.0.1	127.0.0.1	UDP	10554	20494 → 32878 Len=10510
22	47.127742048	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
23	77.162688855	127.0.0.1	127.0.0.1	UDP	8498	20494 → 32878 Len=8454
24	77.163331079	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
25	87.171788862	127.0.0.1	127.0.0.1	UDP	6442	20494 → 32878 Len=6398
26	87.173063217	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
27	97.187311056	127.0.0.1	127.0.0.1	UDP	4386	20494 → 32878 Len=4342
28	97.188910851	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
29	107.197066825	127.0.0.1	127.0.0.1	UDP	2330	20494 → 32878 Len=2286
30	107.197624641	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
31	109.199602387	127.0.0.1	127.0.0.1	UDP	52	32878 → 20494 Len=8
32	109.200453478	127.0.0.1	127.0.0.1	UDP	52	20494 → 32878 Len=8
33	109.201148068	127.0.0.1	127.0.0.1	UDP	62	32878 → 20494 Len=18
34	109.201527212	127.0.0.1	127.0.0.1	UDP	52	32878 → 20494 Len=8

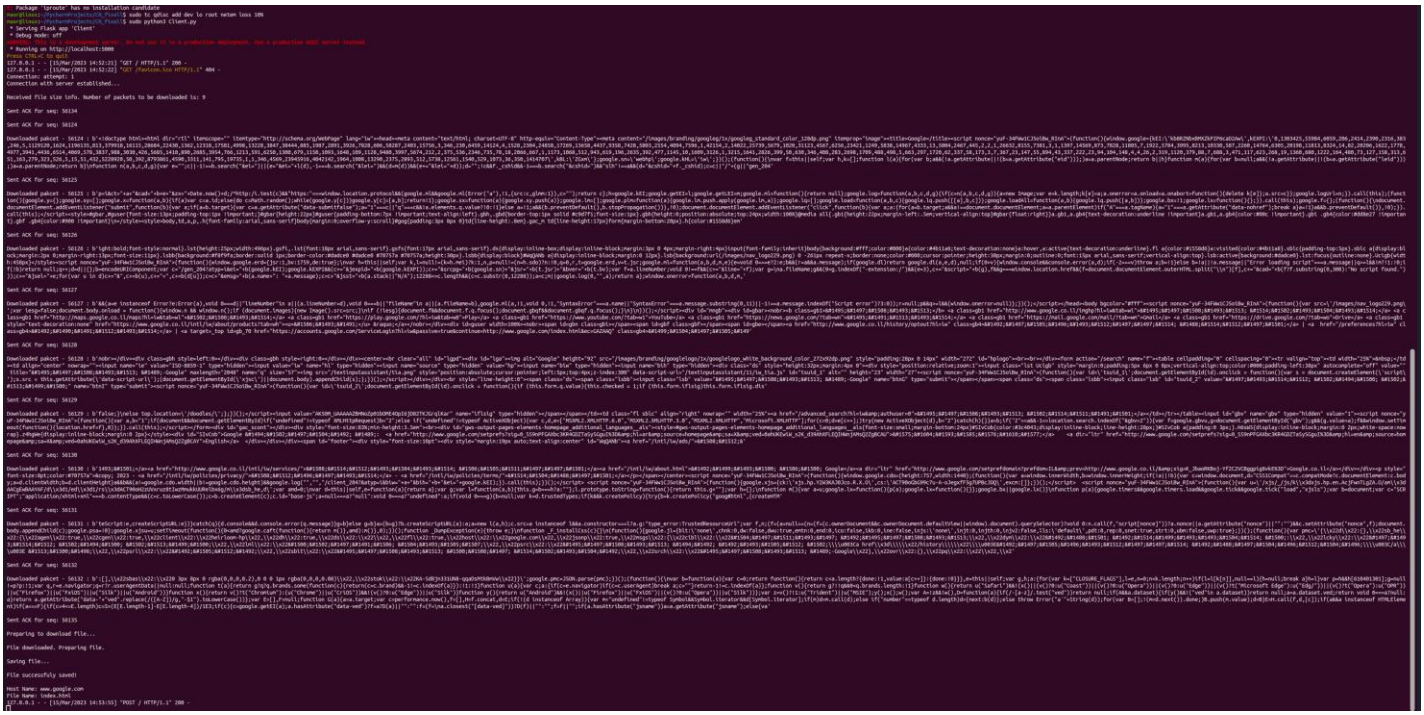
we can see that is the same process like in the 0% packets lost, but if you notice there is more packet that has been sent, because when the packet lost the server didn't get the ACK on this packets and in the RUDP protocol we handle this situation by sending this packets that has been lost again and again until the server get back an ack about the specific packet (we check this with the seq number).

Terminal :

HTTP server:

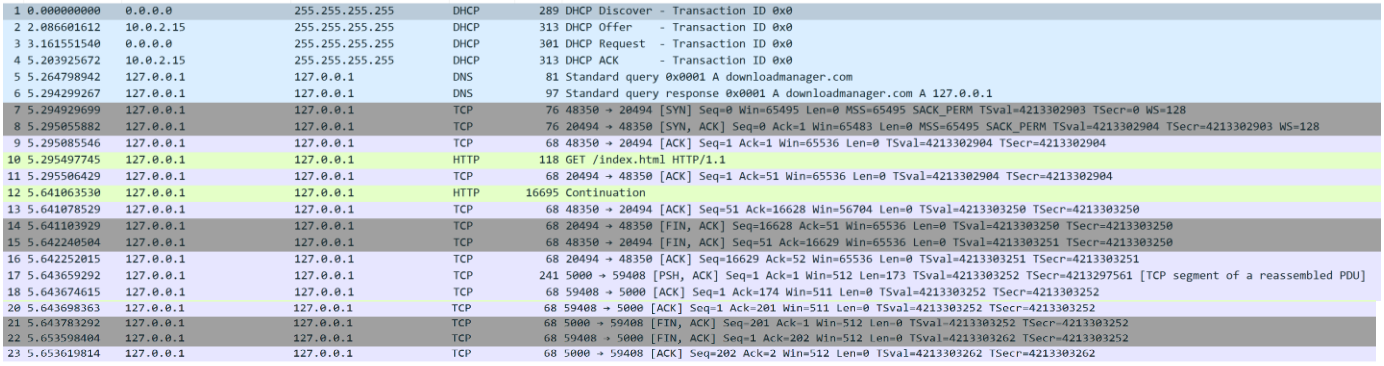
```
maor@linux:~/PycharmProjects/CN_finall$ sudo python3 RUDPDownloadManager.py
Ready to serve...

Connection established with client at IP address: 127.0.0.1
Request packet received...

Extracting URL...
URL for http GET request: http://www.google.com/index.html
Sending HTTP GET request...
GET request was successful!
Getting file: index.html from: www.google.com...
Retreiving file data...
Preparing file for download...
Downloading file...
File info sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

Packet sent successfully.

All packets sent successfully to client.

Something went wrong. . Forcing disconnection with client...

Socket closed.

Download completed successfully!
```

Client :



Rudp :

We can see is the same process but with TCP and there for we can the the SIN, FIN, ACK and more…



1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	2.086601612	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
3	3.161551540	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
4	5.203925672	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0
5	5.264798942	127.0.0.1	127.0.0.1	DNS	81	Standard query 0x0001 A downloadmanager.com
6	5.294299267	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0x0001 A downloadmanager.com A 127.0.0.1
7	5.294929699	127.0.0.1	127.0.0.1	TCP	76	48350 → 20494 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=4213302903 TSecr=0 WS=128
8	5.295055882	127.0.0.1	127.0.0.1	TCP	76	20494 → 48350 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=4213302904 TSecr=4213302903 WS=128
9	5.295085546	127.0.0.1	127.0.0.1	TCP	68	48350 → 20494 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4213302904 TSecr=4213302904
10	5.295497745	127.0.0.1	127.0.0.1	HTTP	118	GET /index.html HTTP/1.1
11	5.295506429	127.0.0.1	127.0.0.1	TCP	68	20494 → 48350 [ACK] Seq=1 Ack=51 Win=65536 Len=0 TSval=4213302904 TSecr=4213302904
12	5.641063530	127.0.0.1	127.0.0.1	HTTP	16695	Continuation
13	5.641078529	127.0.0.1	127.0.0.1	TCP	68	48350 → 20494 [ACK] Seq=51 Ack=16628 Win=56704 Len=0 TSval=4213303250 TSecr=4213303250
14	5.641103929	127.0.0.1	127.0.0.1	TCP	68	20494 → 48350 [FIN, ACK] Seq=16628 Ack=51 Win=65536 Len=0 TSval=4213303250 TSecr=4213303250
15	5.642240504	127.0.0.1	127.0.0.1	TCP	68	48350 → 20494 [FIN, ACK] Seq=51 Ack=16629 Win=65536 Len=0 TSval=4213303251 TSecr=4213303250
16	5.642252015	127.0.0.1	127.0.0.1	TCP	68	20494 → 48350 [ACK] Seq=16629 Ack=52 Win=65536 Len=0 TSval=4213303251 TSecr=4213303251
17	5.643659292	127.0.0.1	127.0.0.1	TCP	241	5000 → 59408 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=173 TSval=4213303252 TSecr=4213297561 [TCP segment of a reassembled PDU]
18	5.643674615	127.0.0.1	127.0.0.1	TCP	68	59408 → 5000 [ACK] Seq=1 Ack=174 Win=511 Len=0 TSval=4213303252 TSecr=4213303252
20	5.643698363	127.0.0.1	127.0.0.1	TCP	68	59408 → 5000 [ACK] Seq=1 Ack=201 Win=511 Len=0 TSval=4213303252 TSecr=4213303252
21	5.643783292	127.0.0.1	127.0.0.1	TCP	68	5000 → 59408 [FIN, ACK] Seq=201 Ack=1 Win=512 Len=0 TSval=4213303252 TSecr=4213303252
22	5.653598404	127.0.0.1	127.0.0.1	TCP	68	59408 → 5000 [FIN, ACK] Seq=1 Ack=202 Win=512 Len=0 TSval=4213303262 TSecr=4213303252
23	5.653619814	127.0.0.1	127.0.0.1	TCP	68	5000 → 59408 [ACK] Seq=202 Ack=2 Win=512 Len=0 TSval=4213303262 TSecr=4213303262

Lines 1-4 connection with the DHCP server (DORA)

Lines 5-6 connection with the DNS server.

Lines open connection between the APP server and the client.

Lines 9 -15 redirect request to another server.

Lines 16 – 20 files transfer.

Lines 21 – 22 closing connection.

1.7 Question

1. List at least four major differences between the TCP and QUIC protocols

TCP (Transmission Control Protocol) and QUIC (Quick UDP Internet Connections) are two transport layer protocols used to transfer data over the internet. Here are four major differences between these two protocols:

1. Reliability: TCP is a reliable protocol, which means it guarantees the delivery of data and detects lost packets by retransmitting them. QUIC is also reliable, but it uses a different mechanism to achieve reliability. QUIC uses a combination of forward error correction and retransmissions to ensure the delivery of data.

2. Connection setup: TCP requires a three-way handshake to establish a connection between the client and server. This process involves a series of messages between the client and server before data transfer can begin. QUIC, on the other hand, uses a single round-trip to establish a connection. This makes QUIC faster than TCP for establishing connections.

3. Congestion control: TCP uses a congestion control algorithm that adjusts the flow of data based on the network conditions. QUIC uses a similar congestion control algorithm, but it is more aggressive and can adapt to changing network conditions more quickly than TCP.

4. Packet loss recovery: When a packet is lost during transmission, TCP uses a retransmission mechanism to resend the lost packet. QUIC, on the other hand, uses a more efficient mechanism that allows it to recover from multiple lost packets at once, reducing latency and improving performance. This mechanism is called "retransmission with loss detection."

2. List at least two main differences between Cubic and Vegas

Cubic and Vegas are two different congestion control algorithms used in TCP. Here are two main differences between Cubic and Vegas:

1. Approach: Cubic and Vegas use different approaches to control congestion. Cubic uses a window-based approach that increases the congestion window size slowly at the beginning and then ramps up quickly to utilize available bandwidth. Vegas, on the other hand, uses a delay-based approach that measures the round-trip time (RTT) of packets and adjusts the sending rate based on the observed delay.

2. Responsiveness: Cubic and Vegas different in their responsiveness to changes in the network conditions. Cubic is less responsive than Vegas and takes longer to react to changes in the available bandwidth. This makes Cubic more suitable for long-lived flows with steady-state traffic. Vegas, on the other hand, is more responsive and can quickly adapt to changes in network conditions, making it more suitable for short-lived flows and bursty traffic.

3. Explain what the BGP protocol is, how it differs from OSPF and does it work according to short routes.

BGP (Border Gateway Protocol) is a routing protocol used to exchange routing information between different networks on the Internet. It is primarily used by Internet Service Providers (ISPs) to route traffic between different autonomous systems (AS).

Unlike OSPF (Open Shortest Path First), which is an Interior Gateway Protocol (IGP) used within a single autonomous system (AS), BGP is an Exterior Gateway Protocol (EGP) used to exchange routing information between different autonomous systems.

One of the key differences between BGP and OSPF is the way they calculate routes. OSPF uses the shortest path algorithm to calculate the shortest path between two routers within the same AS, whereas BGP uses a policy-based routing approach to determine the best path for traffic to take between different ASes. BGP allows network administrators to define policies that influence the path selection process, based on factors such as AS path length, network performance, and cost.

BGP does not necessarily always work according to the shortest routes. The selection of the best path in BGP is based on various factors, including AS path length, local preference, MED (Multi-Exit Discriminator), and other policy-based attributes. Therefore, BGP may choose a longer path if it meets certain policy criteria, such as avoiding a particular network or preferring a certain type of connection. The primary goal of BGP is to select the best path according to the policies defined by the network administrator.

4. Given the code you developed in this project, please add the data to this table based on your project's message process. Explain how the messages will change if there is a NAT between the user and the servers and whether you will use the QUIC protocol.

If there is a NAT between the HTTP app and the client, and the app is redirecting the request of the client and fetching the file from another server, and then sending it back to the client, the table might look like this:

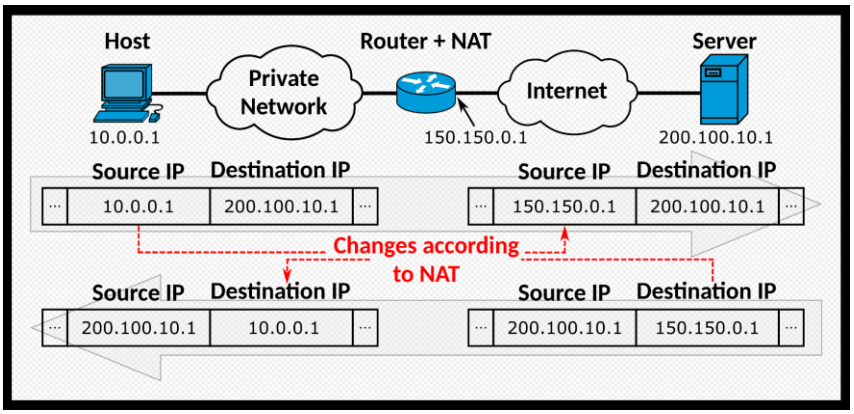
Application	Port Src	Port Des	Ip Src	Ip Des	Mac Src	Mac Des
HTTP	50000	80	192.168.1.100	NAT's Public IP	HTTP app's MAC	NAT's MAC
HTTP	50001	80	192.168.1.100	File server's IP	HTTP app's MAC	File server's MAC
HTTP	50002	880	192.168.1.100	Client's IP	HTTP app's MAC	NAT's MAC

The HTTP app is running on a device with IP address 192.168.1.100 and is listening on port 50000. The client is behind a NAT device with a public IP address, which is used as the destination IP address in the first message. The NAT device translates the source IP address to its own public IP address and forwards the message to the HTTP server.

The HTTP server sends a redirect message back to the client, which contains the IP address of the file server. The HTTP app then sends an HTTP request to the file server to download the file, using a different source port to avoid conflicts with the NAT device's stateful translation.

Once the file is downloaded, the HTTP app sends the file back to the client, using a different source port again to avoid conflicts with the NAT device's stateful translation.

The MAC addresses in this table are specific to the devices involved and will be different for other devices on the network.



If there is a NAT between the user and the servers, the messages processing will change in the following ways:

1. The NAT device will translate the private IP address of the user to a public IP address that is used in the communication with the servers. This means that the IP addresses in the messages exchanged between the user and the servers will be different than the actual IP addresses of the user and the servers.

2. The NAT device will also translate the source port number of the user's messages to a different port number that is used in the communication with the servers. This means that the port numbers in the messages exchanged between the user and the servers will be different than the actual port numbers used by the user.

If the HTTP app is using the QUIC (Quick UDP Internet Connections) protocol, it can help overcome some of the issues caused by NAT devices. QUIC is designed to work well in situations where network conditions are suboptimal, including situations with high latency, packet loss, and NAT devices.

QUIC uses UDP instead of TCP as the underlying transport protocol. This means that it can avoid some of the issues caused by TCP's congestion control mechanisms, which can result in degraded performance in situations with high latency and packet loss.

QUIC also supports encryption and multiplexing, which can help improve security and performance. Encryption can help protect against eavesdropping and tampering, while multiplexing can help improve performance by allowing multiple requests and responses to be sent over a single connection.

In summary, if there is a NAT between the user and the servers, using the QUIC protocol can help improve performance and overcome some of the issues caused by NAT devices. However, the specific impact on message processing will depend on the details of the network topology and the configuration of the NAT device.

If the HTTP app is using the QUIC protocol, the table might look like this:

Application	Port Src	Port Des	Ip Src	Ip Des	Mac Src	Mac Des
HTTP	50000	443	192.168.1.100	NAT's Public IP	HTTP app's MAC	NAT's MAC
HTTP	50001	443	192.168.1.100	File server's IP	HTTP app's MAC	File server's MAC
HTTP	50002	443	192.168.1.100	Client's IP	HTTP app's MAC	NAT's MAC

The HTTP app is using the QUIC protocol on port 443 (the default port for HTTPS) instead of the HTTP protocol on port 80. The client's message is still sent to the NAT device's public IP address, but this time the destination port is 443, which is the default port for HTTPS and QUIC.

The NAT device translates the source IP address to its public IP address and forwards the message to the HTTP app, which is listening on port 50000. The HTTP app sends a redirect message back to the client, which contains the IP address of the file server and the QUIC protocol's Connection ID.

The HTTP app then establishes a QUIC connection with the file server using a different source port (50001 in this case) and sends an HTTP request over this connection to download the file. The response from the file server is also sent over the QUIC connection, which uses the same Connection ID as the redirect message.

Once the file is downloaded, the HTTP app sends the file back to the client over the same QUIC connection, using a different source port (50002 in this case) to avoid conflicts with the NAT device's stateful translation.

In summary, if the HTTP app is using the QUIC protocol, the table will look similar to the previous table, but with the port numbers changed to 443 and the protocol changed to QUIC. The use of the QUIC protocol can help improve performance and overcome some of the issues caused by NAT devices, as discussed in the previous answer.

5. Explain the differences between the ARP protocol and DNS

ARP (Address Resolution Protocol) and DNS (Domain Name System) are two protocols used in computer networks for different purposes. Here are the main differences between ARP and DNS:

1. Purpose: ARP is used to map a physical (MAC) address to an IP address, while DNS is used to map a domain name to an IP address.

2. Scope: ARP operates at the data link layer, which means it is used to resolve addresses within a local network segment. DNS operates at the application layer, which means it can be used to resolve addresses across different networks and the internet.

3. Functionality: ARP is a simple protocol that sends a broadcast message to the network to request the MAC address of a specific IP address. Once the MAC address is obtained, it is cached in the system's ARP cache for future use. DNS, on the other hand, is a more complex protocol that involves a series of recursive queries to different DNS servers to resolve a domain name to an IP address.

4. Implementation: ARP is implemented as a part of the network interface driver in most operating systems and requires no additional software. DNS requires a DNS resolver to be installed on the system, which communicates with one or more DNS servers to resolve domain names.

In summary, ARP is used to map physical addresses to IP addresses within a local network segment, while DNS is used to map domain names to IP addresses across different networks and the internet. ARP operates at the data link layer, while DNS operates at the application layer, and DNS is a more complex protocol that involves recursive queries to multiple servers, while ARP is a simple protocol that sends a broadcast message to obtain the MAC address of a specific IP address.

1.7 Bibliography

DNS server :

https://www.ietf.org/rfc/rfc1035.txt

https://www.youtube.com/@howCode

https://thepacketgeek.com/

DHCP server:

https://www.thepythoncode.com/article/dhcp-listener-using-scapy-in-python

https://www.youtube.com/watch?v=e6-TaH5bkjo

https://he.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol

https://wiki.wireshark.org/DHCP

https://mislove.org/teaching/cs4700/spring11/handouts/project1-primer.pdf

HTTP server:

https://requests.readthedocs.io/en/latest/

https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections

https://stackoverflow.com/questions/27241804/sending-a-file-over-tcp-sockets-in-python

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

TCP protocol:

https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/

QUIC protocol:

https://peering.google.com/#/learn-more/quic

TCP – VEGAS:

https://www.geeksforgeeks.org/basic-concept-of-tcp-vegas/

https://pandorafms.com/blog/tcp-congestion-control/

BGP protocol:

https://en.wikipedia.org/wiki/Border_Gateway_Protocol

OSPF protocol:

https://en.wikipedia.org/wiki/Open_Shortest_Path_First

NAT:

https://www.comptia.org/content/guides/what-is-network-address-translation

https://en.wikipedia.org/wiki/Network_address_translation

ARP:

https://www.fortinet.com/resources/cyberglossary/what-is-arp

for the debugging. Used only to compare different versions **of our own code**:

https://www.diffnow.com/samples

https://stackoverflow.com/questions/10116518/im-getting-key-error-in-python

Html :

https://blog.finxter.com/generating-html-documents-in-python/

https://www.geeksforgeeks.org/how-to-write-to-an-html-file-in-python/

cc :

https://en.wikipedia.org/wiki/CUBIC_TCP

rudp:

https://en.wikipedia.org/wiki/Reliable_User_Datagram_Protocol

https://www.geeksforgeeks.org/reliable-user-datagram-protocol-rudp/

General :

https://www.google.com/webhp?hl=iw&sa=X&ved=0ahUKEwizqcOCxtP9AhXbTKQEHaLwC1QQPAgI

man in wsl

https://www.youtube.com/watch?v=NGLeprazvkM&ab_channel=QuantGuild

We consulted **verbally** with Zohar Simhon and Matan Weiss about the project.

We also consulted with a tutor named Ruet Rabenuo, a developer on Microsoft, about how to construct the html GUI, and generally about all parts of the code an about how to correctly and conveniently build the app. **We did not receive any code from her.**