Connection Networking

FINAL PROJECT

Ron Ashwal

Aviya Arusi

System Description

DNS

1. System purpose
2. System functionality
3. Flowchart
4. Run the project

DHCP

1. System purpose
2. System functionality
3. Flowchart
4. Run the project

APP

1. System purpose
2. System functionality
3. Flowchart
4. Wireshark
5. Run the project

Questions

Bibliographic

Remarks & Video

DNS

A Domain Name System (DNS) is a decentralized and hierarchical naming system that identifies computers accessible via the internet or other Internet Protocol (IP) networks.

The DNS mostly maps human-friendly domain names to numerical IP addresses required by computers to locate devices and services using the underlying network protocols.

A DNS is the Internet's "phonebook" that allows users to connect to sites and access information with a domain name rather than an IP address (e.g., edition.cnn.com). People read online information via domain names, browsers interact through IP addresses, and DNS translates the domain names into IP addresses to enable browsers to upload internet resources.

How Does a DNS Work?

Every Internet-connected device receives a unique IP address (e.g., 192.168.1.1 or more complex ones like 2400:cb00:2048:1::c629:d7a2) that other devices use to find it. The address is required to see every device, like a street name, to find a particular apartment.

Since people can't remember IPs, a translation must transpire between what the user types into their browser and accessing the address. The DNS server converts hostnames (e.g., www.egs.il) into computer-friendly IP addresses, so users don't need to memorize them.

Say you want to visit Google, so you type "google.com" into your browser. You are not connecting to a service specific to google.com. A DNS resolution indicates that google.com is located at this IP address. After that, your computer will connect to that address via a specific service port and load the Google web page.

Functionality

```
import socket
import dns.resolver

# set the DNS server address and port
DNS_ADDRESS = '127.0.0.1'
# the DNS port is set to 53, which is the default port used for DNS queries
DNS_PORT = 53
```

- This code sets up the DNS server address and port that will be used for resolving DNS queries using the dns.resolver module and the socket module.

```
# create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# use the address
sock.bind((DNS_ADDRESS, DNS_PORT))
print(f'DNS server listening on {DNS_ADDRESS}:{DNS_PORT}')

while True:
    # receive DNS query packet from client
    data, address = sock.recvfrom(1024)
    print(f'Received DNS query from {address[0]}:{address[1]}')
```

- This code creates a UDP socket object using the socket module and binds it to the DNS server address and port specified that we already define.
- The sock.bind() method is used to associate the socket with a specific address and port number. In this case, DNS_ADDRESS – '127.0.0.1' , DNS_PORT – '53'. We use port – 53 is the default port for DNS queries.
- The sock.recvfrom() method can be used with an UDP server to receive data from a UDP client or it can be used with an UDP client to receive data from a UDP server.

```
    id = data[:2]
    query = data[12:]
    domain = ''
    i = 0
    while True:
        length = query[i]
        if length == 0:
            break
        i += 1
        domain += query[i:i+length].decode('utf-8') + '.'
        i += length
    query_type = data[-4:-2]
```

- This code appears to be disassemble a DNS query message to extract the query domain name and query type.The first two bytes of the message are extracted into the id variable, which represents a transaction ID that is used to match responses to queries.

```
# Resolve the domain name using dnspython
resolver = dns.resolver.Resolver()
print(domain[:-1])
if domain[:-1] == "www.appliction.com":
    ip_address = "127.0.0.1"
else:
    ip_address = str(resolver.resolve(domain[:-1], 'A')[0])
```

- The dns.resolver.Resolver object will use the system's DNS configuration to resolve queries. However, you can customize the behavior of the resolver by setting various attributes, such as the DNS server address to use, the timeout for queries, and the type of DNS records to request.
- If the client Query is for our application, we return specific IP.
- Else we use the resolver.resolve() method takes two arguments the domain name to query, and the type of DNS record to request (in this case, 'A' for the IPv4 address). The domain name is passed as the first argument. In this case, the [0] index is used to extract the first IPv4 address record from the list.

```
response = id
response += b'\x81\x80'  # Flags
response += b'\x00\x01'  # Questions
response += b'\x00\x01'  # Answer RRs
response += b'\x00\x00'  # Authority RRs
response += b'\x00\x00'  # Additional RRs
response += query
response += b'\xc0\x0c'  # Pointer to domain name
response += query_type
response += b'\x00\x01'  # Query class: IN (Internet)
response += b'\x00\x00\x00\x3c'  # TTL (60 seconds)
response += b'\x00\x04'  # Data length
response += socket.inet_aton(ip_address)
```

- This code constructs a DNS response packet based on the information extracted from the original DNS query packet and the resolved IP address.
- **Id** - is used to identify the response packet as corresponding to the original query.
- **The b'\x81\x80'** - flags indicate that this is a standard response with recursion desired and that there is one answer record in the response.
- **The counts** are set to 1 for the question section and answer section.
- The **b'\xc0\x0c'** pointer is used to indicate that the domain name in the answer record is a compression of the domain name in the original query message.
- The **query_type** variable contains the query type from the original message.
- The **TTL** is set to 60 seconds, and the **data** length is set to 4 bytes for the IPv4 address.
- **socket.inet_aton()** function is used to convert the IP address string to binary format that will be in the DNS packet.

```
sock.sendto(response, address)
```

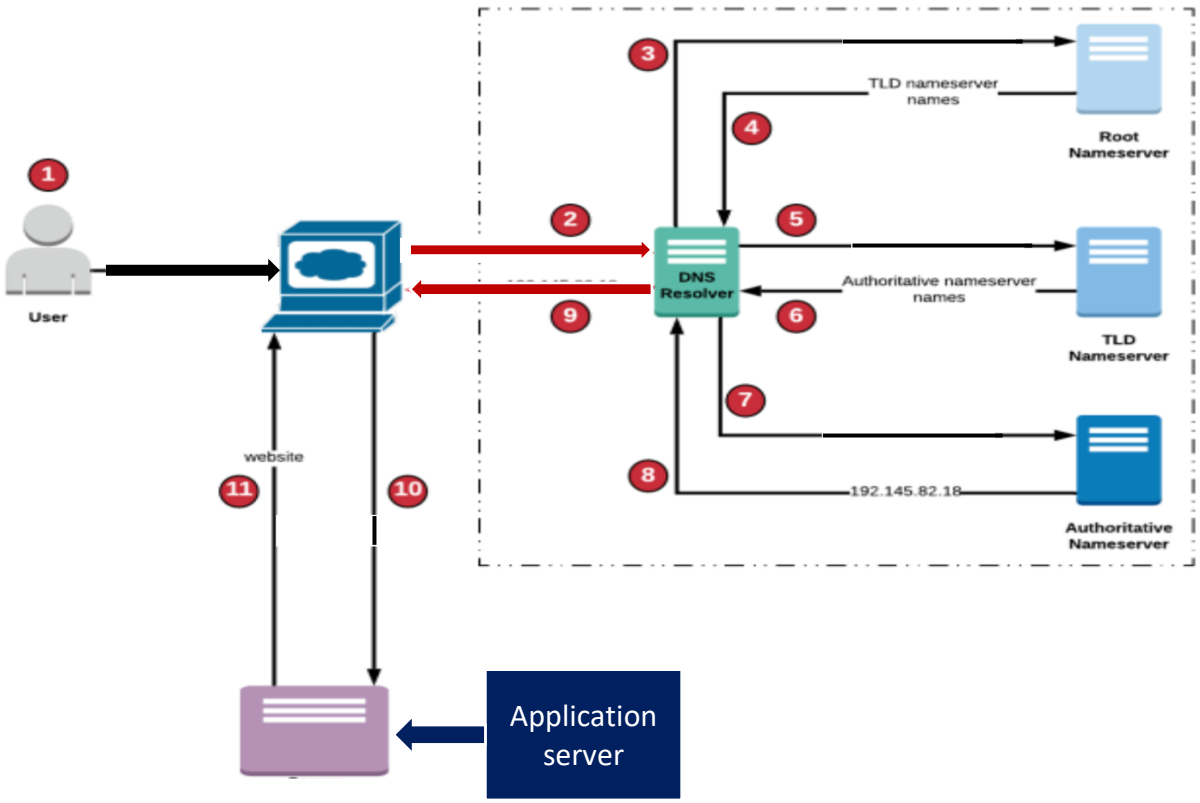
- The method sendto() of the Python's socket class, is used to send datagrams to a UDP socket.
- This basically sent the IP of the query that we receive.

DNS Client

```
# DNS server address and port
DNS_ADDRESS = '127.0.0.1'
DNS_PORT = 53

# Domain name to look up
domain = input("Enter a domain name:")

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Construct the DNS query packet
query = b''
query += b'\xab\xcd'  # Query ID
query += b'\x01\x00'  # Flags
query += b'\x00\x01'  # Questions
query += b'\x00\x00'  # Answer RRs
query += b'\x00\x00'  # Authority RRs
query += b'\x00\x00'  # Additional RRs
for label in domain.split('.'):
    query += bytes([len(label)]) + label.encode('utf-8')
query += b'\x00'  # End of domain name
query += b'\x00\x01'  # Query type: A (IPv4 address)
query += b'\x00\x01'  # Query class: IN (Internet)

# Send the DNS query packet to the server
sock.sendto(query, (DNS_ADDRESS, DNS_PORT))
print(f'Sent DNS query to {DNS_ADDRESS}:{DNS_PORT}')

# Receive the DNS response packet from the server
response, address = sock.recvfrom(1024)
print(f'Received DNS response from {address[0]}:{address[1]}')

# Parse the DNS response packet
ip_address = socket.inet_ntoa(response[-4:])
print('IP address:', ip_address)
```

The user is prompted to enter a domain name.
A UDP socket is created using the socket module.
A DNS query packet is constructed according to the DNS protocol specification. The packet contains a query ID, flags, question count, answer count, authority count, and additional count, as well as the domain name to look up, the query type (IPv4 address), and the query class (Internet). The packet is constructed by concatenating a series of bytes in a specific order.
The DNS query packet is sent to the DNS server using the sendto() method of the socket object. The server's IP address and port number are passed as arguments.
The code waits to receive a response from the server using the recvfrom() method of the socket object. The maximum size of the response is limited to 1024 bytes. The response packet is parsed to extract the IPv4 address of the domain. This is done by taking the last 4 bytes of the response packet (which contain the IP address) and converting them to a string format using the inet_ntoa() method of the socket module.

DNS Flowchart



The query is about our application – www.application.com

DNS Sequence Flow

1. The user enters www.application.com in the address bar of the browser.
2. The request for www.application.com is forwarded to a DNS resolver.
3. The DNS resolver forwards the request for www.application.com to a Root Nameserver.
4. The Root Nameserver for . domain responds to the request with the names of the TLD Nameservers.
5. The DNS resolver then forwards the request for www.application.com to an TLD Nameserver that is associated to .com domain.
6. The TLD Nameserver for .com domain responds to the request with the names of the Authoritative Nameservers that are associated with the application.com.
7. The DNS resolver selects an Authoritative Nameserver and forwards the request for application.com.
8. The Authoritative Nameserver looks in the www.application.com hosted zone for the record, it then gets the associated IP address for a server, 127.0.0.1, and returns the IP address to the DNS resolver.

9. The DNS resolver now has the IP address that the browser needs. The DNS resolver will cache the IP address for www.application.com for an amount of time that is specified for a quicker retrieval the next time www.application.com is requested.
10. The browser sends a request for www.application.com to the IP address that it got from the DNS resolver.
11. The server or other resource at 127.0.0.1 the website for www.application.com to the browser which displays the page.

Run the code

1. Kill all the port that maybe in use:
   Run in the cmd – sudo kill -9 `sudo lsof -t -i: 53`
2. Run in the cmd – sudo python3 DNS.py
3. Run in the cmd – sudo python3 DNS_Client.py
4. Enter the URL that you want

DHCP

Dynamic Host Configuration Protocol (DHCP) is a client/server protocol that automatically provides an Internet Protocol (IP) host with its IP address and other related configuration information such as the subnet mask and default gateway. RFCs 2131 and 2132 define DHCP as an Internet Engineering Task Force (IETF) standard based on Bootstrap Protocol (BOOTP), a protocol with which DHCP shares many implementation details. DHCP allows hosts to obtain required TCP/IP configuration information from a DHCP server.

The DHCP server stores the configuration information in a database that includes:

- Valid TCP/IP configuration parameters for all clients on the network.
- Valid IP addresses, maintained in a pool for assignment to clients, as well as excluded addresses.
- Reserved IP addresses associated with particular DHCP clients. This allows consistent assignment of a single IP address to a single DHCP client.
- The lease duration, or the length of time for which the IP address can be used before a lease renewal is required.

A DHCP-enabled client, upon accepting a lease offer, receives:

- A valid IP address for the subnet to which it is connecting.
- Requested DHCP options, which are additional parameters that a DHCP server is configured to assign to clients. Some examples of DHCP options are Router (default gateway), DNS Servers, and DNS Domain Name.

DHCP provides the following benefits.

- **Reliable IP address configuration**. DHCP minimizes configuration errors caused by manual IP address configuration, such as typographical errors, or address conflicts caused by the assignment of an IP address to more than one computer at the same time.
- **Reduced network administration**. DHCP includes the following features to reduce network administration:

Functionality

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

- In this project we use Scapy library and her functions.
  Scapy is a powerful interactive packet manipulation library written in Python. Scapy is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more.

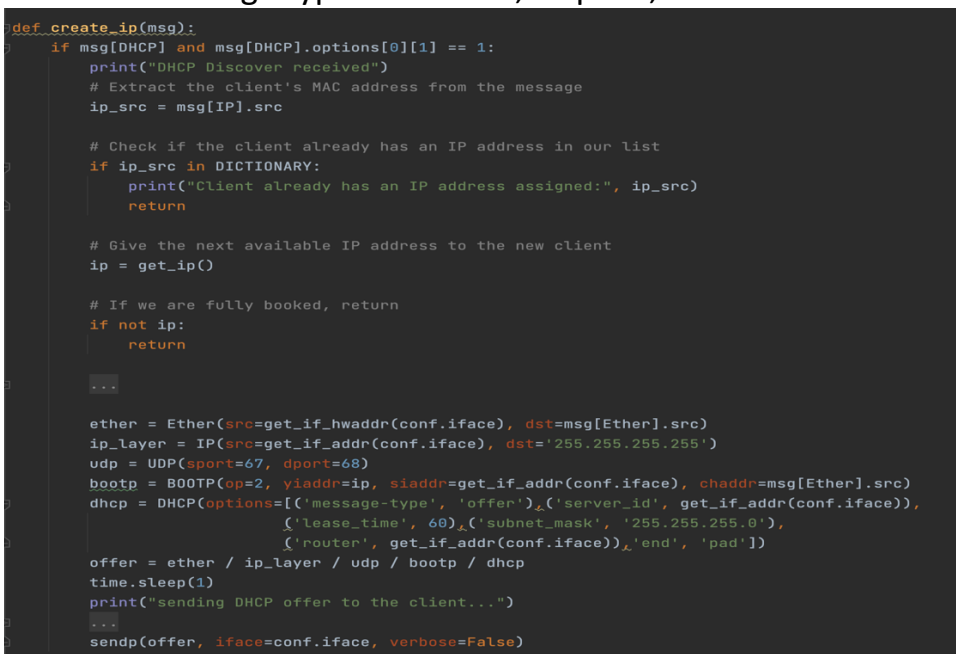
```
print("Waiting for incoming request...")
DATA_BASE = ['192.168.0.%d' % i for i in range(100, 200)]
DICTIONARY = {ip: False for ip in DATA_BASE}
COUNT = 0


def increment():
    global COUNT
    COUNT = COUNT + 1

def decrement():
    global COUNT
    COUNT = COUNT - 1
```

- **DATA BASE** a list of IP address, the list contains 100 IP addresses, ranging from 192.168.0.100 to 192.168.0.199.
- **DICTIONARY** a dictionary that store all the IP from the DATA BASE with parameter False for each one of them.
- **Increment/decrement** for tracking on the number of the IP that in use.

```
def get_ip():
    global COUNT
    if COUNT == 100:
        print("we already serve 100 IP's...")
        exit(1)
    # Checking for each ip in the dict if he false or true
    for ip in DICTIONARY:
        # If we find ip: false send him
        if DICTIONARY[ip] is False:
            DICTIONARY[ip] = True
            increment()
            return ip
    return None
```

- This function returns available IP or None if all the IP addresses are in used. Checking for each IP in the data base if we use it, else return the IP.

Create IP function take the broadcast message and reply to it in a different way, depends on the message type – discover, request, release.



```
def create_ip(msg):
    if msg[DHCP] and msg[DHCP].options[0][1] == 1:
        print("DHCP Discover received")
        # Extract the client's MAC address from the message
        ip_src = msg[IP].src

        # Check if the client already has an IP address in our list
        if ip_src in DICTIONARY:
            print("Client already has an IP address assigned:", ip_src)
            return

        # Give the next available IP address to the new client
        ip = get_ip()

        # If we are fully booked, return
        if not ip:
            return

        ...

        ether = Ether(src=get_if_hwaddr(conf.iface), dst=msg[Ether].src)
        ip_layer = IP(src=get_if_addr(conf.iface), dst='255.255.255.255')
        udp = UDP(sport=67, dport=68)
        bootp = BOOTP(op=2, yiaddr=ip, siaddr=get_if_addr(conf.iface), chaddr=msg[Ether].src)
        dhcp = DHCP(options=[('message-type', 'offer'),('server_id', get_if_addr(conf.iface)),
                             ('lease_time', 60),('subnet_mask', '255.255.255.0'),
                             ('router', get_if_addr(conf.iface)),'end', 'pad'])
        offer = ether / ip_layer / udp / bootp / dhcp
        time.sleep(1)
        print("sending DHCP offer to the client...")
        ...
        sendp(offer, iface=conf.iface, verbose=False)
```

- The DHCP Offer message take the IP of the client and check if he already exists in the list, create new IP for him change IP in DICTIONARY to true. Create the DHCP offer for the client, Ethernet source and destination addresses, sets the IP source and destination addresses, sets the UDP source and destination ports, sets the BOOTP fields in the packet, sets the DHCP options in the packet.

```
elif msg[DHCP] and msg[DHCP].options[0][1] == 3:
    # Extract the client's MAC address and requested IP address from the request
    mac = msg[Ether].src.replace(':', '')
    requested = msg[BOOTP].yiaddr

    # Give the requested IP address to the client
    ip = requested

    ...
    ether = Ether(src=get_if_hwaddr(conf.iface), dst=msg[Ether].src)
    ip_layer = IP(src=get_if_addr(conf.iface), dst='255.255.255.255')
    udp = UDP(sport=67, dport=68)
    bootp = BOOTP(op=5, yiaddr=ip, siaddr=get_if_addr(conf.iface), chaddr=msg[Ether].src)
    dhcp = 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)),('dns', '127.0.0.1'),'end', 'pad'])
    ack = ether / ip_layer / udp / bootp / dhcp
    time.sleep(1)
    print("sending DHCP ack to the client...")
    ...
    sendp(ack, iface=conf.iface, verbose=False)
```

- The DHCP Acknowledge message is the last message as part of the D.O.R.A process that the server sends to the client in which additional details about the network are sent. Build the DHCP ACK, Ethernet source and destination addresses, sets the IP source and destination addresses, sets the UDP source and destination ports, sets the BOOTP fields in the packet, sets the DHCP options in the packet

```
elif msg[DHCP] and msg[DHCP].options[0][1] == 7:
    print("DHCP Release received")
    ip = msg[BOOTP].yiaddr
    decrement()
    # Remove the IP address from the list
    if str(ip) in DICTIONARY and DICTIONARY[str(ip)] == True:
        DICTIONARY[str(ip)] = False
        print("IP address {} released".format(ip))
```

- The DHCP Release message is to free the IP that used to other clients.

DHCP Client

1. Import necessary modules and libraries such as Scapy, subprocess, and time.

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

2. Spawn a new process for DHCP.py and continue executing the current file.

```
# Spawn a new process for file1 and continue executing file2
subprocess.Popen(['python3', 'DHCP.py'])
time.sleep(1)
```

3. Get the MAC address of the current interface and set the IP address to None.

```
mac_address = get_if_hwaddr(conf.iface)
ip_address = None
```

4. Define offer_packet() function that will be called when the DHCP offer packet is captured. The function prints a message and calls send_request_packet() function with the received offer packet as an argument.

```
def offer_packet(packet):
    print("Receiving DHCP offer")
    send_request_packet(packet)
```

5. Define ack_packet() function that will be called when the DHCP ACK packet is captured. The function prints a message.

```
def ack_packet(packet):
    print("Receiving DHCP ack")
```

6. Define send_discover_packet() function that sends the DHCP discover packet to start the DHCP process. The function builds a DHCP discover packet using Scapy and sends it using the sendp() function. It also sets up a packet sniffer using Scapy's sniff() function to capture the DHCP offer

packet. The function uses offer_packet() as the callback function for the captured packets.

```
def send_discover_packet():
    # First the DHCP Discover message is the first message sent by the client throughout the network
    # (that is, for all users of this method of sending they call Broadcast)
    # and this is so that the DHCP server knows that the client wants to connect to the network

    # Build the DHCP discover
    # * Ether - Ethernet source and destination addresses
    # * IP - This sets the IP source and destination addresses
    # * UDP - This sets the UDP source and destination ports
    # * BOOTP - This sets the BOOTP fields in the packet
    # * DHCP - This sets the DHCP options in the packet

    ether = Ether(src=mac_address, dst='ff:ff:ff:ff:ff:ff')
    ip_layer = IP(src='0.0.0.0', dst='255.255.255.255')
    udp = UDP(sport=68, dport=67)
    bootp = BOOTP(op=1, chaddr=mac_address)
    dhcp = DHCP(options=[('message-type', 'discover'), 'end', 'pad'])
    discover = ether / ip_layer / udp / bootp / dhcp
    print("sending DHCP Broadcast...")
    # The sendp function in Scapy is used to send a packet
    # * The discover parameter is the packet that is being sent
    # * The verbose=False argument is used to suppress any output
    sendp(discover, verbose=False)

    # This line of code sets up a packet sniffer using Scapy's sniff function
    sniff(prn=offer_packet, filter='(port 67 or port 68)', count=1)
```

7. Define send_request_packet() function that sends the DHCP request packet as a response to the received DHCP offer packet. The function builds a DHCP request packet using Scapy and sends it using the sendp() function. It also sets up a packet sniffer using Scapy's sniff() function to capture the DHCP ACK packet. The function uses ack_packet() as the callback function for the captured packets.

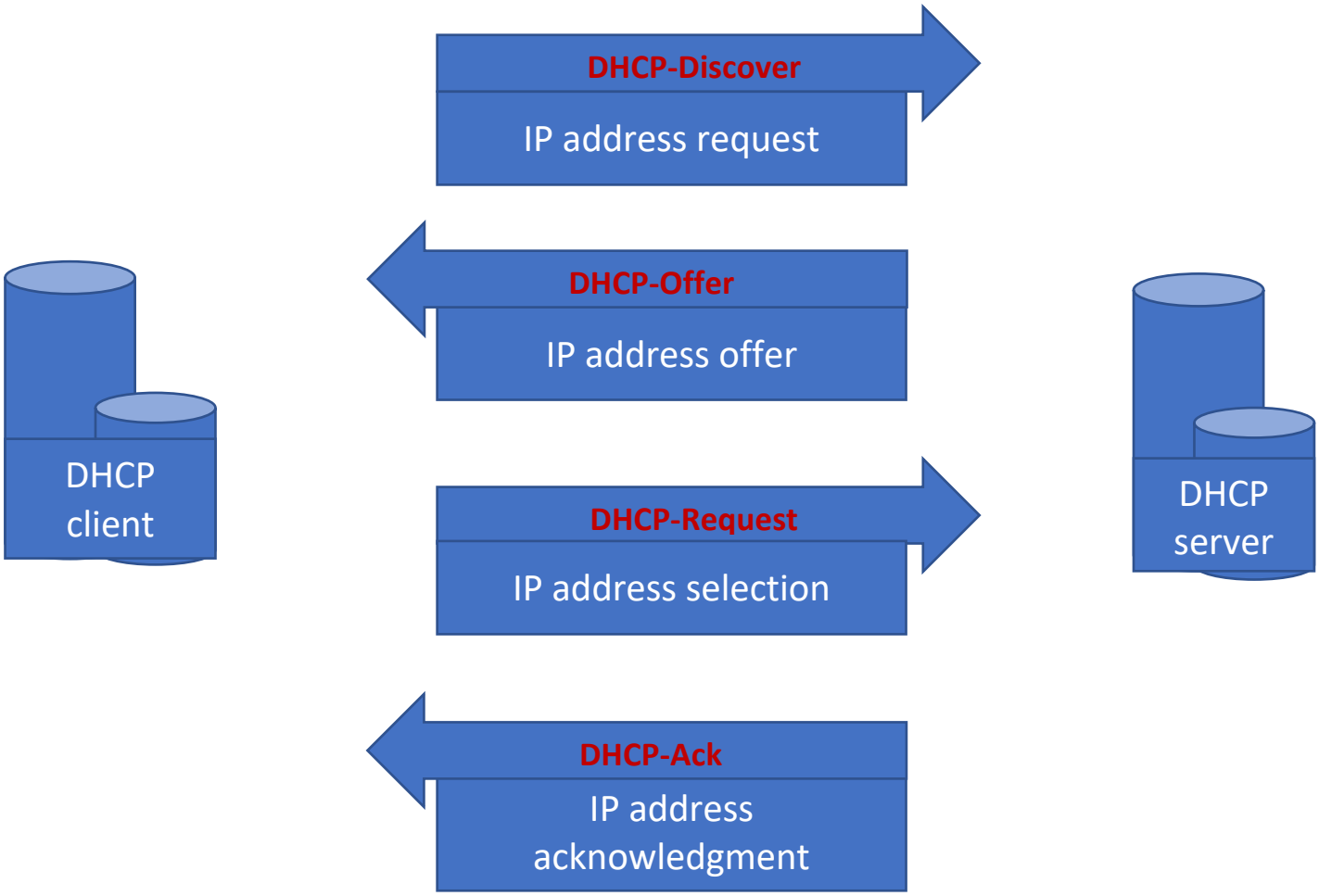
```
def send_request_packet(offer):
    ...
    global ip_address
    ip_address = offer[BOOTP].yiaddr
    print(ip_address)

    # ...

    ether = Ether(src=mac_address, dst='ff:ff:ff:ff:ff:ff')
    ip_layer = IP(src='0.0.0.0', dst='255.255.255.255')
    udp = UDP(sport=68, dport=67)
    bootp = BOOTP(op=1, chaddr=mac_address, yiaddr=ip_address, siaddr=get_if_addr(conf.iface))
    dhcp = DHCP(options=[('message-type', 'request'),
                         ('requested_addr', ip_address),
                         ('server_id', offer[DHCP].options[1][1]), 'end', 'pad'])
    request = ether / ip_layer / udp / bootp / dhcp
    print("sending DHCP Request...")
    ...
    sendp(request, verbose=False)

    sniff(prn=ack_packet, filter='(port 67 or port 68)', count=1)
```

8. send_release_packet() sends a DHCP release packet using the Scapy library.
First, the function sets the global variable ip_address which is assumed to contain the IP address that the DHCP lease was previously assigned to. Then, the code builds a DHCP release packet using the Scapy library by specifying the Ethernet, IP, UDP, BOOTP, and DHCP layers of the packet. Finally, the sendp() function in Scapy is used to send the DHCP release packet to the network. The function sets the verbose parameter to False to suppress any output from Scapy.

```
def send_release_packet():
    print("Sending DHCP release")
    global ip_address
    ...
    ether = Ether(src=mac_address, dst='ff:ff:ff:ff:ff:ff')
    ip_layer = IP(src= ip_address, dst='255.255.255.255')
    udp = UDP(sport=68, dport=67)
    bootp = BOOTP(op=1, chaddr=mac_address, yiaddr=ip_address)
    dhcp = DHCP(options=[('message-type', 'release'), 'end', 'pad'])

    release = ether / ip_layer / udp / bootp / dhcp
    print("sending DHCP Release")
    ...
    sendp(release, verbose=False)
```

The release packet to the server.

DHCP Flowchart



Discover - The DHCP client broadcasts this message to find a DHCP server.

Offer - The DHCP server broadcasts this message to lease an IP configuration to the DHCP client.

Request - The DHCP client uses this message to notify the DHCP server whether it accepts the proposed IP configuration or not.

Acknowledgment - The DHCP server uses this message to confirm the DHCP client that it can use the offered IP configuration.

Run the code

1. Run in the cmd – sudo python3 DHCP.py
2. Run in the cmd – sudo python3 DHCP_CLIENT.py

APP

Http

HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance, text, layout description, images, videos, scripts, and more.

A Web document is the composition of different resources
Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called requests and the messages sent by the server as an answer are called responses.

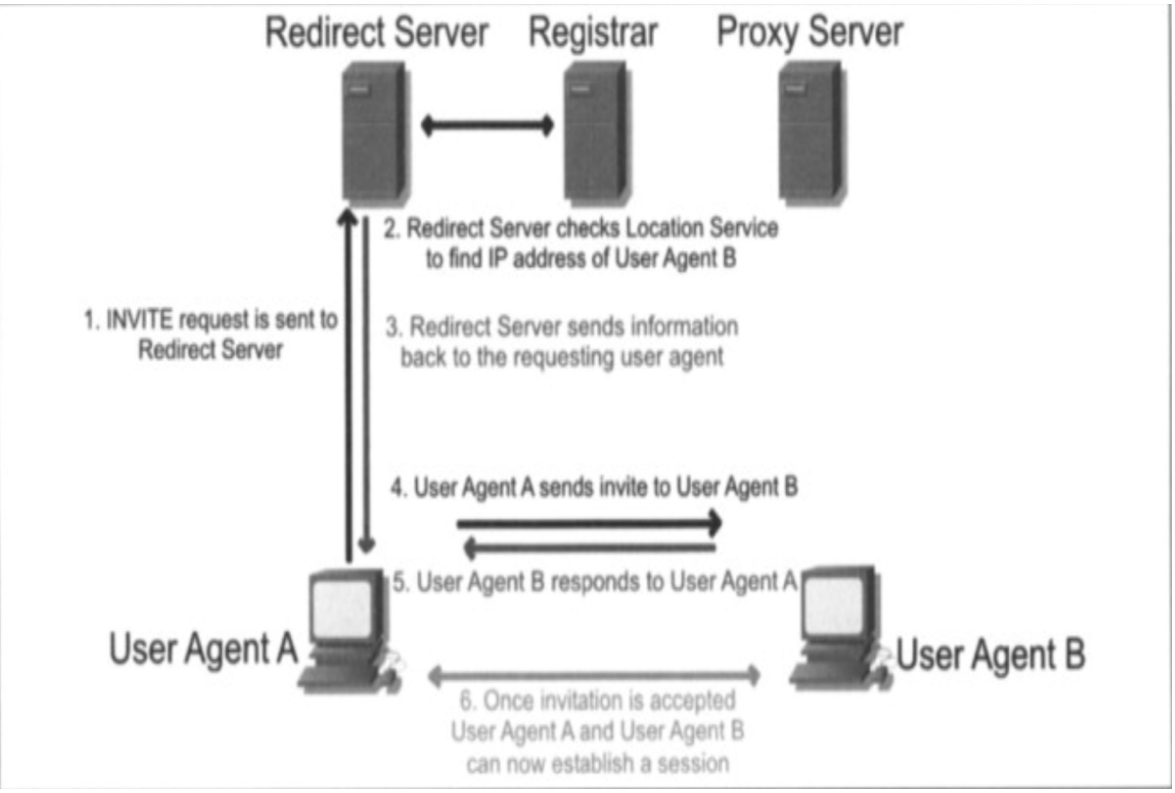
HTTP as an application layer protocol, on top of TCP (transport layer) and IP (network layer) and below the presentation layer.Designed in the early 1990s, HTTP is an extensible protocol which has evolved over time. It is an application layer protocol that is sent over TCP, or over a TLS-encrypted TCP connection, though any reliable transport protocol could theoretically be used. Due to its extensibility, it is used to not only fetch hypertext documents, but also images and videos or to post content to servers, like with HTML form results. HTTP can also be used to fetch parts of documents to update Web pages on demand.

Redirect

Thomas Porter, Michael Gough, in How to Cheat at VoIP Security, 2007

Requests through Redirect Servers

When a Redirect server is used, a request is made to the Redirect server, which returns the IP address of the User agent being contacted. As seen in Figure 3.22, User Agent A sends an INVITE request for User Agent B to the Redirect server, which checks the location service for the IP address of the client being invited. The Redirect server then returns this information to User Agent A. Now that User Agent A has this information, it can now contact User Agent B directly. The INVITE request is now sent to User Agent B, which responds directly to User Agent A. Until this point, SDP is used to exchange information. If the invitation is accepted, then the two User agents would begin communicating and exchanging media using RTP.



Redirect servers can also be used to route traffic between different servers or to load balance traffic across multiple servers. They are

commonly used in network infrastructure, web development, and other applications where efficient routing of requests is important.

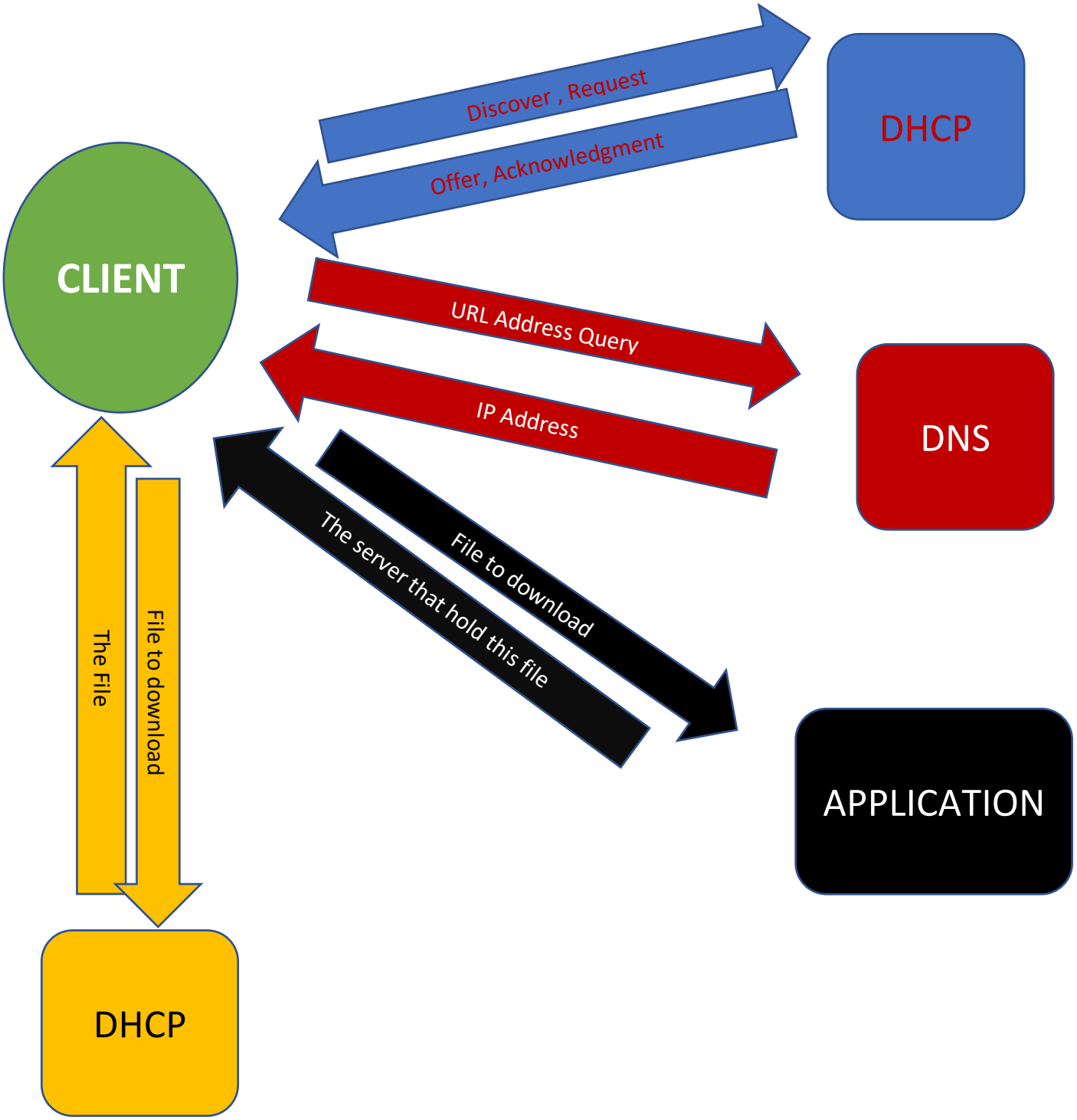
Our project:

A client sends an HTTP request to a server for a specific resource (such as a file or web page), but instead of serving the resource directly, the server sends a redirect response to the client with the URL of another server where the resource can be found. The client then sends a new request to the redirected server to retrieve the resource.

HTTP redirects are often used in situations where a resource has been moved to a different location or when a website has been reorganized. They can also be used to balance server load, by redirecting requests to different servers based on server availability or user location.

Our app is designed to provide end-to-end service to our customers. When a customer first accesses our application, we assign an IP address to them as part of the registration process. **(DHCP)**
From there, our app gives IP to the customer of website they wish to visit based on the URL they enter. **(DNS)**
If the customer wishes to use any of our sub-applications, we provide them with the corresponding address.
Once they inside our app, we offer a range of capabilities that the customer can access, such as selecting a preferred protocol or choosing which information to receive. Ultimately, our goal is to provide a seamless and customizable experience for our customers.

Flow chart


Discover , Request
Offer, Acknowledgment
DHCP
CLIENT
URL Address Query
IP Address
DNS
File to download
The server that hold this file
The File
File to download
APPLICATION
DHCP

Flow chart Description

1. Client – DHCP server

- DHCP file running
- CLIENT send a – Discovery packet
- DHCP send a – Offer packet
- CLIENT send a – Request packet
- DHCP send a – Acknowledgment packet

2. Client – DNS server

- DNS file running
- CLIENT send a – URL address
- DNS send a – IP that directs to this URL

3. Client – Application

- APP file running
- CLIENT send a – File request to the IP from the DNS
- APP send a – Redirect to the server
- CLIENT send a – File request to the server
- SERVER send – The requested file

4. Client – DHCP server

- CLIENT send a – Release packet
- DHCP release the IP of the client

Functionality

APP:

```python
import subprocess
from http.server import BaseHTTPRequestHandler, HTTPServer

class RedirectHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        recieve = self.path.split('/')
        data_to_send = recieve[1]
        protocol = recieve[2]
        url = None
        if protocol == "tcp":
            url = "http://localhost:20139"
            subprocess.Popen(['python3', 'server_tcp.py'])
        else:
            url = "http://localhost:20139"
            arg1 = data_to_send
            subprocess.Popen(['python3', 'server_rudp.py', arg1])
        self.send_response(302)
        self.send_header("127.0.0.1", url+"/"+data_to_send+"/"+protocol)
        self.end_headers()
```

This is a Python class RedirectHandler which inherits from BaseHTTPRequestHandler class provided by the http.server module. It handles HTTP GET requests and redirects them to a TCP or RUDP server running on the local machine.
The do_GET method is overridden to handle HTTP GET requests. The requested URL is split by the forward slash ('/') character to extract the data to be sent (data_to_send) and the protocol (protocol) used for communication.
If the protocol is "tcp", the URL for the TCP server is set and a subprocess is created to run the TCP server using the subprocess.Popen function. Similarly, if the protocol is not "tcp", it is assumed to be "rudp" and the URL for the RUDP server is set. The data to send is passed as an argument to the RUDP server using subprocess.Popen.

```
if __name__ == "__main__":
    server_address = ("", 30701)
    httpd = HTTPServer(server_address, RedirectHandler)
    print("Server application started on port 30701...")
    httpd.serve_forever()
```

The HTTPServer class is a built-in Python module that provides a simple HTTP server. It takes the server address and a request handler as arguments. In this case, the RedirectHandler is used as the request handler.

The print statement outputs a message to the console indicating that the server application has started on port 30701.

Finally, the httpd.serve_forever() method starts the server and blocks the main thread, keeping the server running indefinitely. This method listens for incoming requests and dispatches them to the appropriate handler until the server is shut down.

Server tcp:

```
class ImageHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        url = None
        data = self.path
        word_list = data.split("/")
        type = word_list[1]
        protocol = word_list[2]
        if type == "cat":
            url = 'https://upload.wikimedia.org/wikipedia/commons/a/a5/Red_Kitten_01.jpg'
            content_type = "image/jpeg"
        elif type == "dog":
            url = 'https://upload.wikimedia.org/wikipedia/commons/9/94/My_dog.jpg'
            content_type = "image/jpeg"
        elif type == "video":
            url = 'https://www.youtube.com/watch?v=4rhpbCcSXVs'
            content_type = "video/mp4"
        else:
            url = 'http://www.amitdvir.com/'
            content_type = "text/html"

        with urllib.request.urlopen(url) as f:
            data_file = f.read()

        self.send_response(200)
        self.send_header("Content-type", content_type)
        self.end_headers()
        self.wfile.write(data_file)
```

Depending on the type of request, the url and content_type variables are set to different values. If the request is for a cat or dog image, a URL for a corresponding image is set along with the content type of image/jpeg. If the request is for a video, a YouTube URL is set with a content type of video/mp4. Otherwise, the URL is set to the author's website with a content type of text/html.
The urllib.request.urlopen() method is then used to retrieve the content of the URL specified in url. The content is read and stored in data_file.
The self.send_response() method is used to send an HTTP response back to the client with a status code of 200 (OK). The self.send_header() method is used to specify the content type of the response. Finally, self.wfile.write() method is used to write the content of data_file to the response body.

```
if __name__ == "__main__":
    server_address = ('localhost', 20139)
    httpd = HTTPServer(server_address, ImageHandler)
    print("The tcp server started on port 20139...")
    httpd.serve_forever()
```

The server_address variable is a tuple that specifies the IP address and port number on which the server should listen. In this case, it is set to listen on the localhost interface at port 20139.
The HTTPServer class is a built-in Python module that provides a simple HTTP server. It takes the server address and a request handler as arguments. In this case, the ImageHandler class is used as the request handler.
Finally, the httpd.serve_forever() method starts the server and blocks the main thread, keeping the server running indefinitely. This method listens for incoming requests and dispatches them to the appropriate handler until the server is shut down.

Client rudp:

```
else:
    recieve = url.split(':')
    recieve_port = recieve[2].split('/')
    print(recieve_port[0])
    port = recieve_port[0]
    print("rudp")

    # create socket object
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # set timeout for receiving data
    client_socket.settimeout(15)

    # perform 3-way handshake
    flag = True
    while flag:
        syn_packet = pickle.dumps({"image_data": None, "serial number": -1, "packet code": '[SYN],' + type_of_file})
        client_socket.sendto(syn_packet, (SERVER_ADDRESS, int(port)))
        print("sent SYN ack")
        # receive SYN-ACK packet
        try:
            # Set the timeout to 1 second
            client_socket.settimeout(3)
            data, server_address = client_socket.recvfrom(1024)
            syn_ack_packet = pickle.loads(data)
            # check for SYN-ACK flag
            if '[SYN, ACK]' in syn_ack_packet["packet code"]:
                ack_packet = pickle.dumps({"image_data": None, "serial number": -1, "packet code": '[ACK]'})
                # send ACK packet
                client_socket.sendto(ack_packet, server_address)
                print("sent ACK ack")
                flag = False
        except TimeoutError:
            continue

    # Set the socket to non-blocking mode
    client_socket.setblocking(False)
```

SYN: The client sends a SYN (synchronize) message to the server, indicating that it wants to establish a connection. The SYN message contains a random sequence number that the server will use to identify the connection.

SYN-ACK: The server responds with a SYN-ACK (synchronize-acknowledge) message. The SYN-ACK message contains the same sequence number that the client sent in its SYN message, as well as a random sequence number that the client will use to identify the connection.

ACK: The client sends an ACK (acknowledge) message to the server, confirming that it has received the server's SYN-ACK message. The ACK message contains the server's sequence number incremented by one. At this point, the connection is considered established and both client and server can exchange data.

```
chunk_list = {}
while True:
    try:
        # read data from server
        data_bytes, server_address = client_socket.recvfrom(4096)  # change
        data = pickle.loads(data_bytes)

        # if all data received, break the loop
        if data["packet code"] == "EXIT":
            break

        # add received data to file
        chunk_list.update({data["serial number"]: data["image_data"]})

        # create packet with serial number and
        packet = pickle.dumps(
            {"packet code": "ACK", "last serial number": data["serial number"]})
        # send the packet to server
        client_socket.sendto(packet, server_address)
    except socket.timeout:
        continue
    except BlockingIOError:
        continue

# send exit packet to the server to let him know that this is the last of the chunck's ack
# create packet with EXIT massage
packet = pickle.dumps(
    {"packet code": "EXIT", "last serial number": 0})
# send the packet to server
client_socket.sendto(packet, server_address)

# Set the socket to blocking mode
client_socket.setblocking(True)
```

In this part we receive all the data in chunks and send a acknowledgment on each one of them. The data received pickle.dump() is a function in Python's standard library pickle module that allows you to serialize a Python object into a byte stream (i.e., a sequence of bytes) and save it to a file or send it over a network. The serialized byte stream can later be deserialized using pickle.load() to reconstruct the original Python object.

```
    # create FIN packet
    fin_packet = pickle.dumps({"image_data": None, "serial number": -1, "packet code": '[FIN, ACK]'})

    # send the FIN packet
    client_socket.sendto(fin_packet, server_address)

    # receive the FIN-ACK packet
    data, server_address = client_socket.recvfrom(1024)
    fin_ack_packet = pickle.loads(data)

    # check for the FIN-ACK flag
    if '[ACK]' in fin_ack_packet["packet code"]:

        # receive the FIN packet
        data, server_address = client_socket.recvfrom(1024)
        fin_packet = pickle.dumps(data)

        if '[FIN, ACK]' in fin_packet["packet code"]:
            # create ACK packet with a random sequence number
            ack_packet = pickle.dumps({"image_data": None, "serial number": -1, "packet code": '[FIN, ACK]'})
            # send the ACK packet
            client_socket.sendto(ack_packet, server_address)
except Exception:
    pass
# close the socket
client_socket.close()
print("open file for writing")
if type_of_file == "dog" or type_of_file == "cat":

    with open(type_of_file + ".jpg", 'wb') as f:
        for i in range(len(chunk_list)):
            f.write(chunk_list[i])
elif type_of_file == "html":
    with open(type_of_file + ".txt", 'wb') as f:
        for i in range(len(chunk_list)):
            response_file += chunk_list[i]
        f.write(response_file)
```

We can see here again the three handshake in the end of the process, between the client and the server.
After this the client take the bytes that he get from the server and open a file with the right name.

RUDP

the RUDP that we coded is implement the UDP principle and we adding the reliability like in the TCP protocol.

The code followed by 3 steps of Congestion Control:

1. Slow start - In the first phase of congestion control, the sender sends the packet and gradually increases the number of packets until it reaches a threshold. We know from the flow control discussion, that the window size is decided by the receiver.
2. Congestion Avoidance - The next phase of congestion control is congestion avoidance. In the slow start phase, window size (CWND) increases exponentially. Now, if we continue increasing window size exponentially, there'll be a time when it'll cause congestion in the network. To avoid that, we use a congestion avoidance algorithm. The congestion avoidance algorithm defines how to calculate window size (CWND).

   After each acknowledgment, TCP ensures the linear increment in window size (CWND), slower than the slow start phase.

3. Congestion Detection - The third phase is congestion detection. Now in the congestion avoidance phase, we've decreased the rate of increment in window size (CWND) in order to reduce the probability of congestion in the network. If there is still congestion in the network, we apply a congestion detection mechanism.

The server gets SIN packet from client and preform the 3 hand shake.
After that the server gets the client massage about the file type that he want.
The server go to the url and download the file chunk by chunk and store it in list and dictionary.
After that in every round he calculate the window size by the Cubic algorithm.
He add/remove packet to the cwnd_dictionary that hold the packet that need the to send in this window time.
The server send the packet and wait for ACK (by try).
If the ACK did not arrived he gets to the except and resend the packet.
If the server gets the ACK he remove the specific chunk from the indicator_dictionary
And the cwnd_dictionary.
The server check if there is 3 duplicate ACK and if it is he cut the window size.
After the finish of the sending all the packet from the indicator_dictionary he send to the client massage with EXIT code (-1) so the client know to stop the listening for the file chunk, write the file to the folder and preform the FIN protocol.

First we add the 3 hand shake protocol and the FIN-ACK protocol:

```
# create socket object
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# bind socket to address and port
server_socket.bind((SERVER_ADDRESS, SERVER_PORT))
flag = True
# perform 3-way handshake
while flag:
    # receive the  packet
    data, client_address = server_socket.recvfrom(1024)
    syn_client_packet = pickle.loads(data)
    print("The rudp server get request")
    # check for the SYN flag
    if '[SYN]' in syn_client_packet["packet code"]:
```

This is the window increase:

```
# Send data from chunk_list:
while len(indicator_dic) > 0:

    if not congestion_avoidance:
        # slow start:
        if cwnd < ssthresh:  # if the window smaller then the ssthresh
            cwnd += 1
        # linear:
        else:
            cwnd += 1 / cwnd
    else:  # congestion avoidance:
```

This is the Cubic variable's:

```
# cubic variables
w_max = 256
K = 0.4
beta = 0.7
C = 0.4
T = 0
last_time = time.time()
```

This is the Cubic calculation:

```
# Calculate the scaling factor K based on the current window size and elapsed time since the last window reduction
elapsed_time = time.time() - last_time
K = ((w_max * (1 - beta)) / C) ** (1 / 3)
T = elapsed_time
# Calculate the new congestion window size using the cubic function
curr_cwnd = (C * (T - K) ** 3 + w_max)
```

This is the window adding and removing:

```
    if len(cwnd_dic) < cwnd:  # add chunks to list
        for key in indicator_dic:
            if len(cwnd_dic) < cwnd:
                cwnd_dic.update({key: indicator_dic[key]})
            else:
                break
    else:  # remove chunks from the list
        for key in cwnd_dic:
            if len(cwnd_dic) > cwnd:
                cwnd_dic.pop(key)
            else:
                break
```

Send the chunk:

```
    # If congestion window is not full, send a new packet
    if len(cwnd_dic) > 0:
        for key in cwnd_dic:
            data = pickle.dumps({"image_data": chunk_list[key], "serial number": key, "packet code": "PUSH"})
            # Send data to client
            server_socket.sendto(data, client_address)

            # Record the time of sending this packet
            send_time = time.time()
```

Try to get the ack:

```
    # Receive ACK packet from client
try:
    # Set the timeout to 1 second
    server_socket.settimeout(1)
    try:
        # Receive ACK packet
        data, client_address = server_socket.recvfrom(8192)  # change from 1024
    except ConnectionResetError:
        continue
    if data is not None:
        ack_packet = pickle.loads(data)
        if ack_packet["last serial number"] in indicator_dic:
            # remove from indicator_dic
            indicator_dic.pop(ack_packet["last serial number"])

        if ack_packet["last serial number"] in cwnd_dic:
            # remove from cwnd_dic
            cwnd_dic.pop(ack_packet["last serial number"])
```

If there is 3 dup ACK's:

```
        if ack_packet["last serial number"] in cwnd_dic:
            # remove from cwnd_dic
            cwnd_dic.pop(ack_packet["last serial number"])
        # check for 3 duplicate ACKs
        if ack_packet["last serial number"] == dupack_dic["serial number"]:
            dupack_dic["counter"] += 1
            # Perform congestion control based on the number of duplicate ACKs
            if dupack_dic["counter"] >= 3:
                ssthresh = max(cwnd / 2, 1)
                cwnd = ssthresh
                dupack_dic["counter"] = 0
        else:
            dupack_dic["serial number"] = ack_packet["last serial number"]
            dupack_dic["counter"] = 1
```

If the recv() did not successes resend the packet:

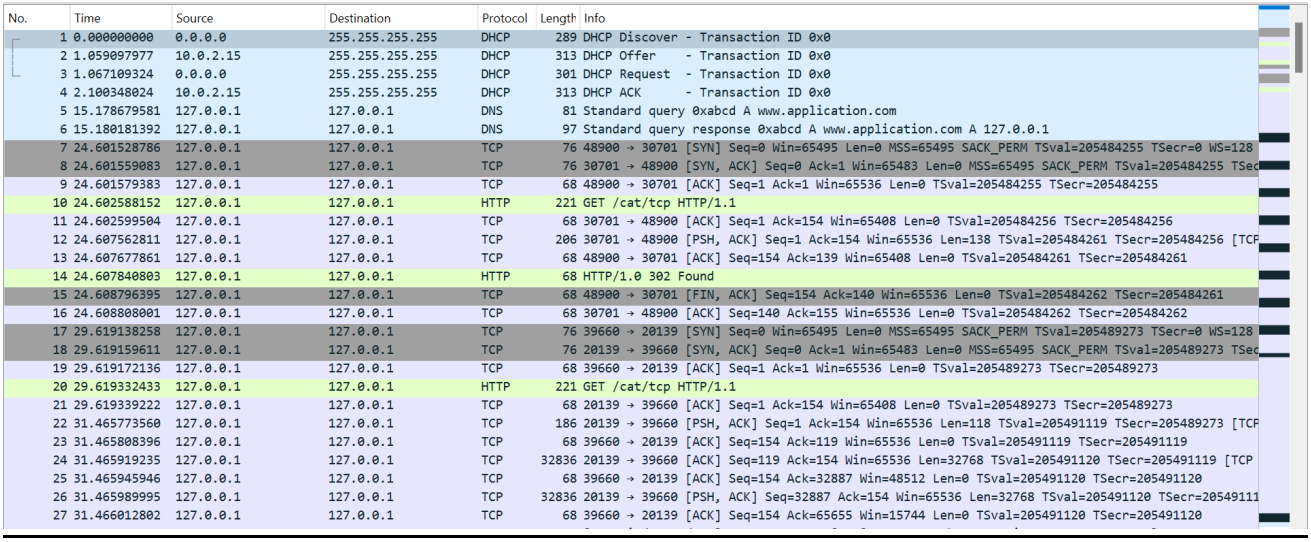
```
except socket.timeout:
    # Timeout occurred, retransmit the first unacknowledged packet
    for key in cwnd_dic:
        data = pickle.dumps({"image_data": chunk_list[key], "serial number": key,
                             "packet code": "RESEND"})
        server_socket.sendto(data, client_address)

        # Perform congestion control
        ssthresh = 1
        cwnd = ssthresh
        send_time = time.time()
        break
```

FIN protocol:

```
# check for the FIN-ACK flag
if '[FIN, ACK]' in fin_packet["packet code"]:
    print("gets the FIN packet from client")
    # create ACK packet with a random sequence number
    ack_packet = pickle.dumps({"image_data": None, "serial number": -1, "packet code": '[ACK]'})

    # send the ACK packet
    server_socket.sendto(ack_packet, client_address)
```

Wireshark

0% packet loss TCP:

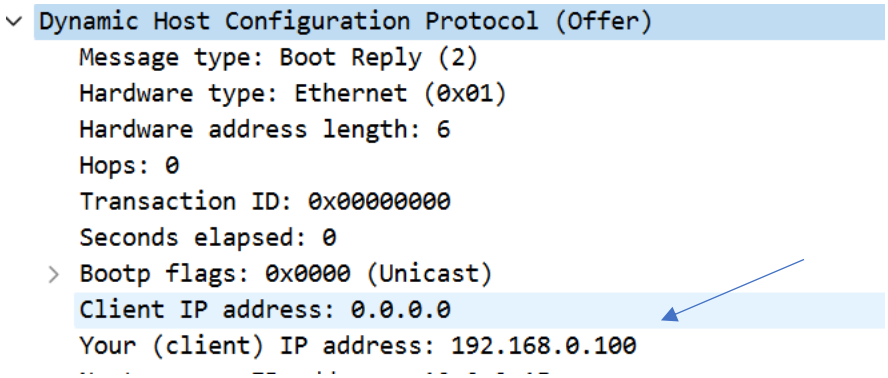


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	1.059097977	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
3	1.067109324	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
4	2.100348024	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0
5	15.178679581	127.0.0.1	127.0.0.1	DNS	81	Standard query 0xabcd A www.application.com
6	15.180181392	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0xabcd A www.application.com A 127.0.0.1
7	24.601528786	127.0.0.1	127.0.0.1	TCP	76	48900 → 30701 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=205484255 TSecr=0 WS=128
8	24.601559083	127.0.0.1	127.0.0.1	TCP	76	30701 → 48900 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=205484255 TSec
9	24.601579383	127.0.0.1	127.0.0.1	TCP	68	48900 → 30701 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=205484255 TSecr=205484255
10	24.602588152	127.0.0.1	127.0.0.1	HTTP	221	GET /cat/tcp HTTP/1.1
11	24.602599504	127.0.0.1	127.0.0.1	TCP	68	30701 → 48900 [ACK] Seq=1 Ack=154 Win=65408 Len=0 TSval=205484256 TSecr=205484256
12	24.607562811	127.0.0.1	127.0.0.1	TCP	206	30701 → 48900 [PSH, ACK] Seq=1 Ack=154 Win=65536 Len=138 TSval=205484261 TSecr=205484256 [TCP
13	24.607677861	127.0.0.1	127.0.0.1	TCP	68	48900 → 30701 [ACK] Seq=154 Ack=139 Win=65408 Len=0 TSval=205484261 TSecr=205484261
14	24.607840803	127.0.0.1	127.0.0.1	HTTP	68	HTTP/1.0 302 Found
15	24.608796395	127.0.0.1	127.0.0.1	TCP	68	48900 → 30701 [FIN, ACK] Seq=154 Ack=140 Win=65536 Len=0 TSval=205484262 TSecr=205484261
16	24.608808001	127.0.0.1	127.0.0.1	TCP	68	30701 → 48900 [ACK] Seq=140 Ack=155 Win=65536 Len=0 TSval=205484262 TSecr=205484262
17	29.619138258	127.0.0.1	127.0.0.1	TCP	76	39660 → 20139 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=205489273 TSecr=0 WS=128
18	29.619159611	127.0.0.1	127.0.0.1	TCP	76	20139 → 39660 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=205489273 TSec
19	29.619172136	127.0.0.1	127.0.0.1	TCP	68	39660 → 20139 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=205489273 TSecr=205489273
20	29.619332433	127.0.0.1	127.0.0.1	HTTP	221	GET /cat/tcp HTTP/1.1
21	29.619339222	127.0.0.1	127.0.0.1	TCP	68	20139 → 39660 [ACK] Seq=1 Ack=154 Win=65408 Len=0 TSval=205489273 TSecr=205489273
22	31.465773560	127.0.0.1	127.0.0.1	TCP	186	20139 → 39660 [PSH, ACK] Seq=1 Ack=154 Win=65536 Len=118 TSval=205491119 TSecr=205489273 [TCP
23	31.465808396	127.0.0.1	127.0.0.1	TCP	68	39660 → 20139 [ACK] Seq=154 Ack=119 Win=65536 Len=0 TSval=205491119 TSecr=205491119
24	31.465919235	127.0.0.1	127.0.0.1	TCP	32836	20139 → 39660 [ACK] Seq=119 Ack=154 Win=65536 Len=32768 TSval=205491120 TSecr=205491119 [TCP
25	31.465945946	127.0.0.1	127.0.0.1	TCP	68	39660 → 20139 [ACK] Seq=154 Ack=32887 Win=48512 Len=0 TSval=205491120 TSecr=205491120
26	31.465989995	127.0.0.1	127.0.0.1	TCP	32836	20139 → 39660 [PSH, ACK] Seq=32887 Ack=154 Win=65536 Len=32768 TSval=205491120 TSecr=20549111
27	31.466012802	127.0.0.1	127.0.0.1	TCP	68	39660 → 20139 [ACK] Seq=154 Ack=65655 Win=15744 Len=0 TSval=205491120 TSecr=205491120

Packet's 1-4: DHCP

1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	1.059097977	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
3	1.067109324	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
4	2.100348024	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0

1. The Discover packet – from source IP is 0.0.0.0 and the destination is 255.255.255.255 the client just get into the network and he sent broadcast massage to all the network, asking for new local IP address.
2. The offer packet – from source IP 10.0.2.15 (the DHCP server address) to all the network. The massage supposes to get to the client with an offered IP address (it can be see in the photo -> 192.168.0.100).



3. The Request packet - from source IP is 0.0.0.0 and the destination is 255.255.255.255 the client sent massage to all the network that he get the IP address that offered to him.

4. The ACK packet – from source IP 10.0.2.15 to all the network the server ACK the last massage.

Packet's 5-6: DNS

```
5 15.178679581  127.0.0.1          127.0.0.1          DNS        81 Standard query 0xabcd A www.application.com
6 15.180181392  127.0.0.1          127.0.0.1          DNS        97 Standard query response 0xabcd A www.application.com A 127.0.0.1
```

5. The client send massage to the server asking for the IP address of the domain name -> www.application.com

```
▾ Queries
  › www.application.com: type A, class IN
```

6. The server response to the client the requested domain name to IP address (127.0.01).

```
▾ Queries
  › www.application.com: type A, class IN
▾ Answers
  › www.application.com: type A, class IN, addr 127.0.0.1
```

Packet's 7-16: the client-application connection:

```
 7 24.601528786  127.0.0.1          127.0.0.1          TCP        76 48900 → 30701 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=205484255 TSecr=0 WS=128
 8 24.601559083  127.0.0.1          127.0.0.1          TCP        76 30701 → 48900 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=205484255 TSec
 9 24.601579383  127.0.0.1          127.0.0.1          TCP        68 48900 → 30701 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=205484255 TSecr=205484255
10 24.602588152  127.0.0.1          127.0.0.1          HTTP      221 GET /cat/tcp HTTP/1.1
11 24.602599504  127.0.0.1          127.0.0.1          TCP        68 30701 → 48900 [ACK] Seq=1 Ack=154 Win=65408 Len=0 TSval=205484256 TSecr=205484256
12 24.607562811  127.0.0.1          127.0.0.1          TCP       206 30701 → 48900 [PSH, ACK] Seq=1 Ack=154 Win=65536 Len=138 TSval=205484261 TSecr=205484256 [TCP
13 24.607677861  127.0.0.1          127.0.0.1          TCP        68 48900 → 30701 [ACK] Seq=154 Ack=139 Win=65408 Len=0 TSval=205484261 TSecr=205484261
14 24.607840803  127.0.0.1          127.0.0.1          HTTP       68 HTTP/1.0 302 Found
15 24.608796395  127.0.0.1          127.0.0.1          TCP        68 48900 → 30701 [FIN, ACK] Seq=154 Ack=140 Win=65536 Len=0 TSval=205484262 TSecr=205484261
16 24.608808001  127.0.0.1          127.0.0.1          TCP        68 30701 → 48900 [ACK] Seq=140 Ack=155 Win=65536 Len=0 TSval=205484262 TSecr=205484262
```

7-9 . 3-hand shake.
sever port -> 30701. client port -> 48900.
10. The client send HTTP/1.1 GET massage (this time asking for cat image to be send in TCP protocol).
11. ACK massage from server to client.

12. PSH massage from server to client with the new redirect address (20139).

```
23 12:52 :03 GMT·
·127.0.0 .1: http
://local host:201
39/cat/t cp····
```

13. ACK massage from the client to server.
14. FOUND massage.
15-16. FIN protocol.

17-114: the transport of the requested image in TCP protocol between the client end the second server.
115. the client informs the entire network that he is leaving.

115	37.794699817	127.0.0.1	255.255.255.255	DHCP	289	DHCP Release - Transaction ID 0x0

10% packet's lost TCP:

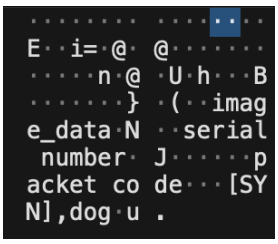
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	0.0.0.0	255.255.255.255	DHCP	289	DHCP Discover - Transaction ID 0x0
2	1.178510646	10.0.2.15	255.255.255.255	DHCP	313	DHCP Offer - Transaction ID 0x0
3	1.238479349	0.0.0.0	255.255.255.255	DHCP	301	DHCP Request - Transaction ID 0x0
4	2.348874951	10.0.2.15	255.255.255.255	DHCP	313	DHCP ACK - Transaction ID 0x0
5	26.392760076	127.0.0.1	127.0.0.1	DNS	81	Standard query 0xabcd A www.application.com
6	26.394164814	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0xabcd A www.application.com A 127.0.0.1
7	63.392448280	127.0.0.1	127.0.0.1	TCP	76	40390 → 30701 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=213876079 TSecr=0 WS=128
8	63.392468362	127.0.0.1	127.0.0.1	TCP	76	30701 → 40390 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=213876080 TSec
9	63.392483109	127.0.0.1	127.0.0.1	TCP	68	40390 → 30701 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=213876080 TSecr=213876080
10	63.392907567	127.0.0.1	127.0.0.1	HTTP	221	GET /cat/tcp HTTP/1.1
11	63.392915555	127.0.0.1	127.0.0.1	TCP	68	30701 → 40390 [ACK] Seq=1 Ack=154 Win=65408 Len=0 TSval=213876080 TSecr=213876080
12	63.396474746	127.0.0.1	127.0.0.1	TCP	206	30701 → 40390 [PSH, ACK] Seq=1 Ack=154 Win=65536 Len=138 TSval=213876084 TSecr=213876080 [TCP
13	63.396618085	127.0.0.1	127.0.0.1	HTTP	68	HTTP/1.0 302 Found
14	63.397321818	127.0.0.1	127.0.0.1	TCP	68	40390 → 30701 [FIN, ACK] Seq=154 Ack=140 Win=65536 Len=0 TSval=213876084 TSecr=213876084
15	63.397333547	127.0.0.1	127.0.0.1	TCP	68	30701 → 40390 [ACK] Seq=140 Ack=155 Win=65536 Len=0 TSval=213876084 TSecr=213876084
16	65.197415751	127.0.0.1	127.0.0.1	TCP	74	63342 → 37676 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=6 TSval=213877884 TSecr=213867873
17	65.197450233	127.0.0.1	127.0.0.1	TCP	68	37676 → 63342 [ACK] Seq=1 Ack=7 Win=512 Len=0 TSval=213877885 TSecr=213877884
18	68.404789833	127.0.0.1	127.0.0.1	TCP	76	40818 → 20139 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=213881092 TSecr=0 WS=128
19	68.404807226	127.0.0.1	127.0.0.1	TCP	76	20139 → 40818 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=213881092 TSec
20	68.404819535	127.0.0.1	127.0.0.1	TCP	68	40818 → 20139 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=213881092 TSecr=213881092
21	68.405164553	127.0.0.1	127.0.0.1	HTTP	221	GET /cat/tcp HTTP/1.1
22	68.405171805	127.0.0.1	127.0.0.1	TCP	68	20139 → 40818 [ACK] Seq=1 Ack=154 Win=65408 Len=0 TSval=213881092 TSecr=213881092
23	68.557111604	10.0.2.15	91.198.174.208	TCP	76	52126 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3496678048 TSecr=0 WS=128
24	68.621919526	91.198.174.208	10.0.2.15	TCP	62	443 → 52126 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
25	68.622644085	10.0.2.15	91.198.174.208	TCP	56	52126 → 443 [ACK] Seq=1 Ack=1 Win=64240 Len=0
26	68.630693083	91.198.174.208	10.0.2.15	TCP	62	[TCP ACKed unseen segment] 443 → 52126 [ACK] Seq=1 Ack=518 Win=65535 Len=0
27	68.700191831	10.0.2.15	91.198.174.208	TCP	56	[TCP ACKed unseen segment] [TCP Previous segment not captured] 52126 → 443 [ACK] Seq=518 Ack=

0% packet's lost RUDP:

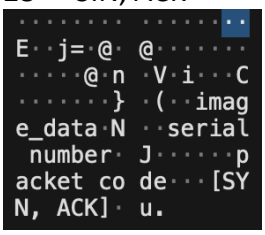
No.	Time	Source	Destination	Protocol	Length	Info
15	13.717117287	127.0.0.1	127.0.0.1	DNS	81	Standard query 0xabcd A www.application.com
16	13.719431680	127.0.0.1	127.0.0.1	DNS	97	Standard query response 0xabcd A www.application.com A 127.0.0.1
17	22.150666556	127.0.0.1	127.0.0.1	TCP	76	43030 → 30701 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=4196352211 TSecr=0 WS=128
18	22.150693384	127.0.0.1	127.0.0.1	TCP	76	30701 → 43030 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=4196352211 TSecr=4196352211 W
19	22.150714021	127.0.0.1	127.0.0.1	TCP	68	43030 → 30701 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4196352211 TSecr=4196352211
20	22.151001135	127.0.0.1	127.0.0.1	HTTP	222	GET /dog/rudp HTTP/1.1
21	22.151012057	127.0.0.1	127.0.0.1	TCP	68	30701 → 43030 [ACK] Seq=1 Ack=155 Win=65408 Len=0 TSval=4196352211 TSecr=4196352211
22	22.155604149	127.0.0.1	127.0.0.1	TCP	206	30701 → 43030 [PSH, ACK] Seq=1 Ack=155 Win=65536 Len=138 TSval=4196352216 TSecr=4196352211 [TCP segment of a
23	22.155771525	127.0.0.1	127.0.0.1	TCP	68	43030 → 30701 [ACK] Seq=155 Ack=139 Win=65408 Len=0 TSval=4196352216 TSecr=4196352216
24	22.156002910	127.0.0.1	127.0.0.1	HTTP	68	HTTP/1.0 302 Found
25	22.157849627	127.0.0.1	127.0.0.1	TCP	68	43030 → 30701 [FIN, ACK] Seq=155 Ack=140 Win=65536 Len=0 TSval=4196352218 TSecr=4196352216
26	22.157861919	127.0.0.1	127.0.0.1	TCP	68	30701 → 43030 [ACK] Seq=140 Ack=156 Win=65536 Len=0 TSval=4196352218 TSecr=4196352218
27	27.165684480	127.0.0.1	127.0.0.1	UDP	121	55406 → 8000 Len=77
28	27.169693108	127.0.0.1	127.0.0.1	UDP	122	8000 → 55406 Len=78
29	27.171897155	127.0.0.1	127.0.0.1	UDP	117	55406 → 8000 Len=73

In packet's 27-28 we see the 3 hand shake protocol preform by the servr and the client ->

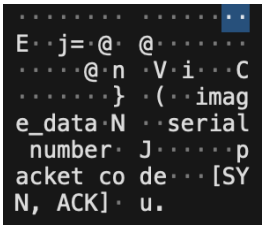
27 -> SYN



28 -> SIN, ACK



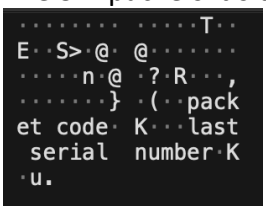
29-> ACK



and now the server start to send using UDP all the chunk of the file by pip line and get ack massage from the client.

No.	Time	Source	Destination	Protocol	Length	Info
2094	28.796629091	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2095	28.796650682	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2096	28.796730016	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2097	28.796865829	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2098	28.796954221	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2099	28.796967781	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2100	28.796998695	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2101	28.796975626	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2102	28.797009630	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2103	28.797019769	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2104	28.796996150	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2105	28.797003686	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2106	28.797034881	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2107	28.797010461	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034
2108	28.797044233	127.0.0.1	127.0.0.1	UDP	99	55406 → 8000 Len=55
2109	28.797017394	127.0.0.1	127.0.0.1	UDP	1078	8000 → 55406 Len=1034

the firs 2 packet's its chunk that send from the server to the client.
The 3-4 packe's it's the client ACK packet (we can see the "last serial number"):



And here is the FIN packet's:

130…	41.052481456	127.0.0.1	127.0.0.1	UDP	104	55406 → 8000 Len=60
130…	41.052492289	127.0.0.1	127.0.0.1	UDP	122	55406 → 8000 Len=78
130…	41.052895770	127.0.0.1	127.0.0.1	UDP	117	8000 → 55406 Len=73

And this is how it looks inside the FIN packet:

Run the code

- **Without packet lost**
    4. Kill all the port that maybe in use:
       Run in the cmd – sudo kill -9 `sudo lsof -t -i: {d}`
       d in {53 , 20139, 30701, 8000}
    5. Run in the cmd – sudo python3 application
    6. Enter the URL that you want
    7. Enter the protocol that you want
    8. Enter the file that you want
- **With packet lost**
    1. Kill all the port that maybe in use:
       Run in the cmd – sudo kill -9 `sudo lsof -t -i: {d}`
       d in {53 , 20139, 30701, 8000}
    2. Run in the cmd – sudo python3 dhcp_dns.py
    3. Enter the URL that you want
    4. Run in the cmd – sudo tc qdisc add dev lo root netem loss 10%
    5. Run in the cmd – sudo python3 app.py
    6. Enter the protocol that you want
    7. Enter the file that you want

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

TCP

The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. Major internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP, which is part of the Transport Layer of the TCP/IP suite.

TCP is connection-oriented, and a connection between client and server is established before data can be sent. The server must be listening (passive open) for connection requests from clients before a connection is established. Three-way handshake (active open), retransmission, and error detection adds to reliability but lengthens latency. Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP) instead, which provides a connectionless datagram service that prioritizes time over reliability. TCP employs network congestion avoidance. However, there are vulnerabilities in TCP, including denial of service, connection hijacking, TCP veto, and reset attack.

Quic

The QUIC transport protocol was apparently designed to address several issues with TCP and TLS, and in particular, to improve the transport performance for encrypted traffic with faster session setup, to allow for further evolution of transport mechanisms and explicitly avoid the emerging TCP ossification within the network.

At its simplest level, QUIC is simply TCP encapsulated and encrypted in a User Datagram Protocol (UDP) payload. To the external network, QUIC has the appearance of a bidirectional UDP packet sequence where the UDP payload is concealed. To the endpoints, QUIC can be used as a reliable full duplex data flow protocol. Even at this level, QUIC has a number of advantages over TCP.

The 4 main difference between TCP and Quic:

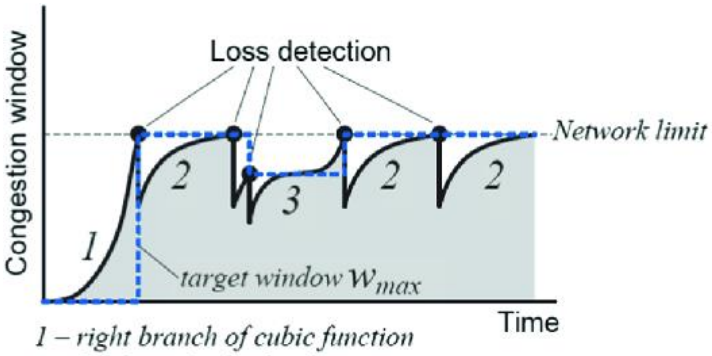
1. Connection Setup: TCP requires a three-way handshake to establish a connection and the minimum RTT(round-trip-time) is 3, while QUIC can uses minimum of only 1 RTT, or even a zero – RTT if there is a connection in the past between the client and the server. This means that QUIC can establish connections faster than TCP, as it does not require the back-and-forth communication of the handshake.
2. Reliability: TCP provides reliable data transmission by retransmitting lost packets. In contrast, QUIC is also designed to provide reliable data transmission, but it uses a different mechanism called forward error correction (FEC) to recover lost packets. This can result in faster recovery times and reduced latency compared to TCP(that send packets in order and if packet 2 did not get to the server so 3 do not sent yet).
3. Congestion Control: TCP uses a congestion control algorithm that is based on detecting packet loss as a signal of congestion. QUIC, on the other hand, uses a more sophisticated congestion control algorithm that takes into account other factors such as round-trip time and available bandwidth. This can lead to more efficient use of available network resources.
4. Security: Both TCP and QUIC support encryption, but QUIC has built-in support for encryption and authentication, while TCP requires additional protocols such as TLS to provide encryption. This can make QUIC easier to use and configure securely than TCP.
5. Connection migration feature: QUIC ensuring the connection will survive even as you switch networks, and TCP don’t.

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

Cubic

CUBIC is a network congestion avoidance algorithm for TCP which can achieve high bandwidth connections over networks more quickly and reliably in the face of high latency than earlier algorithms. It helps optimize long fat networks.

Furthermore cubic is a congestion control protocol for TCP (transmission control protocol) and the current default TCP algorithm in Linux. The protocol modifies the linear window growth function of existing TCP standards to be a CUBIC function in order to improve the scalability of TCP over fast and long- distance networks. It also achieves more equitable bandwidth allocations among flows with different RTTs (round trip times) by making the window growth to be independent of RTT – thus those flows grow their congestion window at the same rate. During steady state, CUBIC increases the window size aggressively when the window is far from the saturation point, and the slowly when it is close to the saturation point. This feature allows CUBIC to be very scalable when the bandwidth and delay product of the network is large, and at the same time, be highly stable and also fair to standard TCP flows. The implementation of CUBIC in Linux has gone through several upgrades.



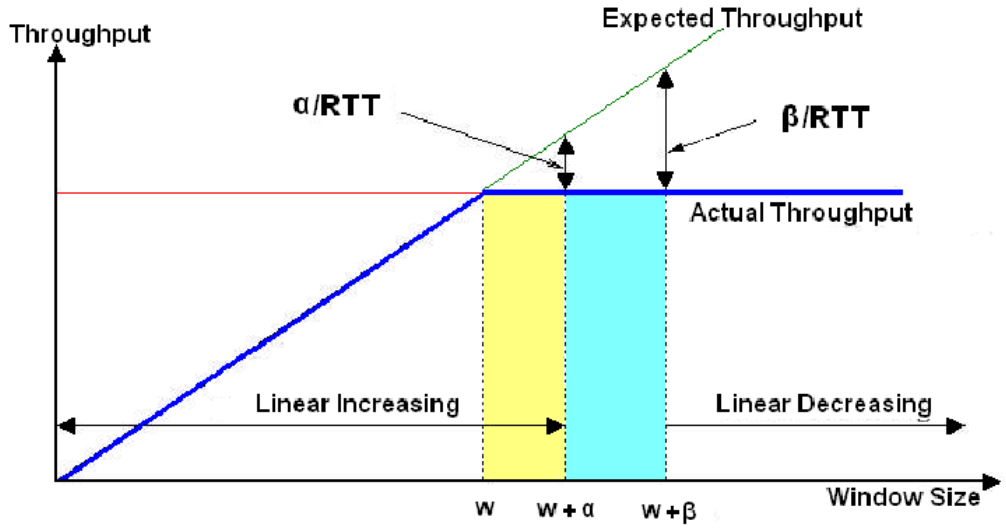
1 – right branch of cubic function
2 – left branch of cubic function
3 – left and right branches of cubic function

Vegas

TCP Vegas is a TCP congestion avoidance algorithm that emphasizes packet delay, rather than packet loss, as a signal to help determine the rate at which to send packets. It was developed at the University of Arizona by Lawrence Brakmo and Larry L. Peterson and introduced in 1994.

TCP Vegas detects congestion at an incipient stage based on increasing Round-Trip Time (RTT) values of the packets in the connection unlike other flavors such as Reno, New Reno, etc., which detect congestion only after it has actually happened via packet loss. The algorithm depends heavily on accurate calculation of the Base RTT value. If it is too small then throughput of the connection will be less than the bandwidth available while if the value is too large then it will overrun the connection.

A lot of research is going on regarding the fairness provided by the linear increase/decrease mechanism for congestion control in Vegas. One interesting caveat is when Vegas is inter-operated with other versions like Reno. In this case, performance of Vegas degrades because Vegas reduces its sending rate before Reno, as it detects congestion early and hence gives greater bandwidth to co-existing TCP Reno flows.



Differences between Cubic & Vegas

First, the choice between CUBIC and TCP Vegas depends on the specific requirements of the network and the traffic patterns involved. CUBIC may be preferred in highly congested or heavily loaded networks, while TCP Vegas may be better suited for lightly loaded or high bandwidth-delay product networks.

Cubic and Vegas are two different congestion control algorithms used in TCP (Transmission Control Protocol) to balance the flow of data between the sender and the receiver. The main differences between the two are:

1. CUBIC uses a window-based approach to congestion control, while TCP Vegas uses a delay-based approach. CUBIC gradually increases its transmission window until congestion is detected, while TCP Vegas monitors the network delay and adjusts the transmission rate based on the observed delay.
2. CUBIC uses packet loss as a signal of network congestion, while TCP Vegas monitors the delay or round-trip time of each data packet to detect congestion. This means that CUBIC may take longer to detect congestion and reduce its transmission rate, while TCP Vegas can detect congestion more quickly and adjust its transmission rate accordingly.
3. CUBIC aims to be fair to other competing traffic on the network by reducing its data transmission rate when other traffic is detected. TCP Vegas also attempts to be fair to other flows by sharing bandwidth proportionally based on the observed delay.
4. CUBIC is generally more effective at maintaining high network utilization and avoiding congestion collapse, especially in highly congested or heavily loaded networks, while TCP Vegas is designed to reduce delay and improve throughput, especially in lightly loaded or high bandwidth-delay product networks.

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

BGP

The Border Gateway Protocol (BGP) is the protocol used throughout the Internet to exchange routing information between networks. It is the language spoken by routers on the Internet to determine how packets can be sent from one router to another to reach their final destination. BGP has worked extremely well and continues to the be protocol that makes the Internet work.

The challenge with BGP is that the protocol does not directly include security mechanisms and is based largely on trust between network operators that they will secure their systems correctly and not send incorrect data. Mistakes happen, though, and problems could arise if malicious attackers were to try to affect the routing tables used by BGP.

OSPF is better suited for routing within a single network or organization, while BGP is better suited for routing between different organizations or networks. OSPF is faster at adapting to changes in the network topology, while BGP is better at handling large-scale networks with complex routing policies.

BGP can be configured to use short routes, but it does not necessarily prioritize them. BGP is designed to take into account different factors when making routing decisions, including the cost of different routes, the availability of alternate paths, and network policies. BGP can be configured to use the shortest path as the primary route, but it may also use alternate paths to balance traffic load or meet other policy requirements. The specific routing decisions made by BGP depend on the configuration of the network and the policies set by the network administrator.

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

Application	Port Src	Port Des	IP Src	IP Des	Mac Src	Mac Des

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.

Application	Port Src	Port Des	Ip Src	Ip Des	Mac Src	Mac Des
HTTP	30701	20139	192.168.0.100	NAT's Public IP	HTTP app's MAC	NAT's MAC
HTTP	30702	20140	192.168.0.100	File server's IP	HTTP app's MAC	File server's MAC
HTTP	30703	20141	192.168.0.100	Client's IP	HTTP app's MAC	NAT's MAC

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	30701	20139	192.168.0.100	NAT's Public IP	HTTP app's MAC	NAT's MAC
HTTP	30702	20140	192.168.0.100	File server's IP	HTTP app's MAC	File server's MAC
HTTP	30703	20141	192.168.0.100	Client's IP	HTTP app's MAC	NAT's MAC

The HTTP application utilizes the QUIC protocol on the default HTTPS rather than the HTTP protocol. The client's message is directed to the NAT device's public IP address, but with a destination port, which is the default port for both HTTPS and QUIC.

Upon receiving the message, the NAT device translates the source IP address to its public IP address and sends it to the HTTP application, which is listening on. The application sends a redirect message back to the client, which contains the IP address of the file server and the Connection ID of the QUIC protocol.

Afterward, the HTTP application establishes a QUIC connection with the file server using a different source port and sends an HTTP request over this connection to download the file. The response from the file server is also sent over the same QUIC connection, utilizing the same Connection ID as the redirect message.
When the file download is complete, the HTTP application sends the file back to the client over the same QUIC connection but with a different source port to avoid conflicts with the NAT device's stateful translation.

In summary, if the HTTP application uses the QUIC protocol, the table will appear similar to the previous table, but with the port numbers changed to, and the protocol changed to QUIC. Utilizing the QUIC protocol can enhance performance and overcome some of the challenges caused by NAT devices, as mentioned in the preceding response.

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

The difference between ARP and DNS:
ARP (Address Resolution Protocol) and DNS (Domain Name System) are both protocols used in computer networking, but they serve different purposes.
ARP is used to map a known IP address to a physical MAC address in a local network. This is done to establish a direct communication between two devices on the same network, and is typically used in Ethernet networks.

DNS, on the other hand, is used to map a domain name (such as "www.google.com") to an IP address. This allows users to access websites and other resources on the internet using easy-to-remember domain names, rather than having to remember IP addresses. DNS is a distributed database system that translates domain names into IP addresses and vice versa.

1.The ARP use link layer frames when DNS use the application.
2.They solve difference problem.
In summary, ARP is used for mapping IP addresses to MAC addresses on a local network, while DNS is used for mapping domain names to IP addresses on the internet.

Bibliographic

- https://learn.microsoft.com/en-us/windows-server/networking/technologies/dhcp/dhcp-top
- https://www.internetsociety.org/deploy360/securing-bgp/?gclid=CjwKCAiAmJGgBhAZEiwA1JZollGBYwhltw-i0286kkfgIewjtAlYms0k_8REJM5G8RgP6Yo57CDDWxoCTtIQAvD_BwE
- https://stackoverflow.com/questions/61150608/modulenotfounderror-no-module-named-dnspython
- https://www.codespeedy.com/get-ip-address-of-a-url-in-python/
- https://en.wikipedia.org/wiki/TCP_Vegas
- https://en.wikipedia.org/wiki/CUBIC_TCP
- https://www.computernetworkingnotes.com/ccna-study-guide/how-dhcp-works-explained-with-examples.html
- https://www.techtarget.com/searchnetworking/tip/A-guide-to-Windows-DHCP-server-configuration
- https://elementor.com/resources/glossary/what-is-a-domain-name-system-dns/?utm_source=google&utm_medium=cpc&utm_campaign=13060922353&utm_term=&gclid=CjwKCAjwiOCgBhAgEiwAjv5whA1UJsMosZiS6ERGG1zxW8Pv1v6O9eQHG55Ty4ei-mF2BsfvttgT6RoC1NYQAvD_BwE
- https://www.domain.com/help/article/what-is-dns-record?psafe_param=1&utm_source=google&utm_medium=genericsearch&gclid=CjwKCAjwiOCgBhAgEiwAjv5whAr98RsTyzJOPe9VWLDxbgNRDVpj8GskrEUeEnNV1wTDMS22Cw0iKxoCfbMQAvD_BwE&gclsrc=aw.ds
- https://stackoverflow.com/questions/21641696/python-dns-module-import-error
- https://pythontic.com/modules/socket/recvfrom
- https://dnspython.readthedocs.io/en/latest/resolver-class.html
- https://www.baeldung.com/cs/tcp-flow-control-vs-congestion-control
- https://www.youtube.com/watch?v=vqoLacbGIc0
- https://www.internalpointers.com/post/making-http-requests-sockets-python
- https://pythontic.com/modules/socket/sendto
- https://application.com/2019/11/25/deep-dive-dns/

- https://mislove.org/teaching/cs4700/spring11/handouts/project1-primer.pdf
- https://www.sciencedirect.com/topics/computer-science/redirect-server
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
- https://www.exploredatabase.com/2021/09/differentiate-between-domain-name-system-and-address-resolution-protocol-DNS-vs-ARP.html
- https://inapp.com/blog/quic-vs-tcp-the-full-story/
- https://netbeez.net/blog/tcp-vs-quic-new-transport-
- https://docs.python.org/3/library/subprocess.html#replacing-os-popen-os-popen2-os-popen3
- https://stackoverflow.com/questions/12605498/how-to-use-subprocess-popen-python
- https://docs.python.org/3/library/http.server.html
- https://www.sciencedirect.com/topics/computer-science/three-way-handshake
- https://wiki.python.org/moin/UsingPickle
-

Remarks

In our project we have 10 Files:

- DHCP, DHCP_CLIENT – For running only this part.

- DNS, DNS_Client – For running only this part
- Client, app - For running only this part
- Application – For running all the parts together without packet lost
- Dhcp_dns, app – for running all the parts together with packet lost
- server_tcp, server_rudp the servers that sending the files

We upload the pcap files of : tcp with and without loss after filters, rudp with and without lost, but are computers couldn't filter the relevant packets.

In order to succeed our project we take advice from Maor, Dovi and Zohar.
And are friend Itamar Epstein that work in the industry.
To see the code in the wireshark we use another computer.
We hope that pdf help to understand our project and how to run it.

Link to our video

https://edu-il.zoom.us/rec/share/Q_HW86WExOvF1kszUwXvM7usse7cpZSyGq18o8XMBysCEHef6ppHqCW7dYubzt5R.ducVLS-8I-orlfOg

Passcode: vS0Ws@

Pls copy the url and enter the code below.