| 318848413 | 325092781 |
| --- | --- |

# Networking Project - FTP

## Table Of Contents

- This is an hyper-linked supported table.

---

Subjects

# About This Project

In this Project, we are implementing DHCP and DNS servers alongside a client and server in our application, while getting familiar with all the protocols we learned (TCP, UDP, RUDP), File transfer protocol, GUI techniques and much more in the python language.
This exercise is divided into 4 main parts:

**DHCP** - A DHCP server to assign the client his network configuration.
**DNS** - A DNS server to provide the client with the address of the domains in its records.
**FTP Client** - The client side which connects to the FTP app.
**FTP Server** - The FTP server which runs the FTP app.

Few main points:

- All testing were done on Windows 10 Home/Windows 11 Pro and on PyCharm IDE.
- The testings include: WireShark monitoring and terminal executing programs.
- Rarely, due to system resources, the GUI may crash at the start-up.
  Re-run it to solve this problem.
- Since our DHCP server can also respond to other DHCP request,
  It can cause the process at the start to take longer (since it is dealing with another client).
  You can turn off you WI-FI, wait till the server is available or re-run the client.
- Notice that when choosing to upload/download in TCP, the socket enters TIME_WAIT mode (which takes about 30 to 60 seconds) in which the client can't reconnect to the server.

Files included:

- ☐ Application:
  - ☑ FTP_client.py - Code file for the client side in the app.
  - ☑ FTP_server.py - Code file for the server side in the app.
  - ☑ Send.py - Methods to simulate delay/packet loos environment.
- ☐ Domains:
  - ☑ Folders for domains to download from or upload files to.
- ☐ Graphical_Interface:
  - ☑ GUI.py - Object for the graphic interface.
  - ☑ Download.py - Object for the download window.
  - ☑ StopUpload.py - Object window for stopping upload mid way through (in TCP).
  - ☑ UploadType.py - Object window for choosing upload type (in RUDP).
- ☐ Servers:
  - ☑ DHCP.py - Code file for the DHCP server.
  - ☑ DNS.py - Code file for the DNS server.
- ☐ WireShark: The WireShark recordings.

# Abstract

This research study discusses the detail overview in creating DHCP and DNS servers, creating FTP based client-server system, and designing an app with GUI (Graphical User Interface) in the Python programming language.

This research also deals with Python threading and the different types of protocols and their headers manipulating while learning their struct and understanding how to change, copy and modify it to our like.
This is implanted while using the Scapy library and many of its methods.

UDP and TCP socket programming style, creating RUDP (Reliable UDP) with DUP ACKs and Go-Back-N technique and creating delay and packet loss environment are also considered and learned in the research stage.

Non the less, the design of the app itself using Tkinter and customTkinter libraries, while applying it to work with the whole system was challenging.

The main objective of this research study is to demonstrate the principles and concepts behind uploading/downloading a file, while implementing what we learned during the course.
However, given a free hand, we were asked in this project to show our creativity.

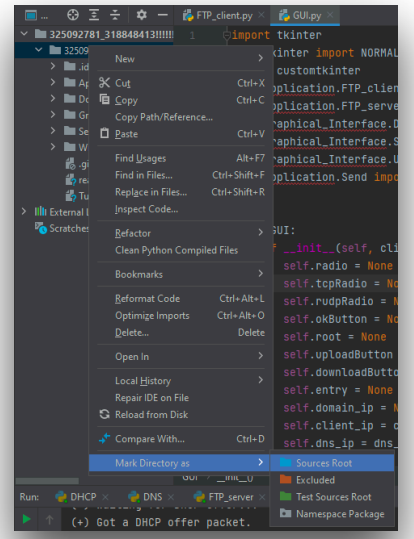
# Instructions To Run Program

To make it easy for testing and checking the code in this submission,
we are adding instructions on how anyone approaching this should run the included files.

## **Step 1:** Prepare your workspace.

Open the project with your IDE.
You may be asked to download some of the libraries that we import.

- We recommend you turn off your Wi-FI to avoid real time DNS and DHCP request being answered by the servers provided in this project.
- We recommend you run this on Windows OS.
- Make sure you have Npcap installed on your computer.
- Please define the main project folder as Source Root to avoid errors:




- Go to "Graphical_Interface" folder, and on “GUI.py”, “StopUpload.py” and “UploadType.py” put comment on the next line:

```
# Remember to delete later.
self.root.geometry("+1920+0")
```

This line was supposed to open the windows of the GUI in another screen (when working on 2 screen setup) and can cause problems when starting the app.

## **Step 2:** Run the files.

To run each .py file, it is enough to run it from your IDE.

If you are using PyCharm, just go over to the file and open it, right click on the window and choose to run the file or just click on the run button in the upper right.


Please run the files as following:

1) First run DHCP.py and DNS.py (in the Servers folder). Doesn't matter which one first.

**DHCP**

```
(*) Starting DHCP server...
(+) Server IP:  192.168.1.33
(+) Next Client IP suggestion:  192.168.1.132
(+) Next Client IP broadcast:  192.168.1.255

(*) Waiting for DHCP discovery...
```

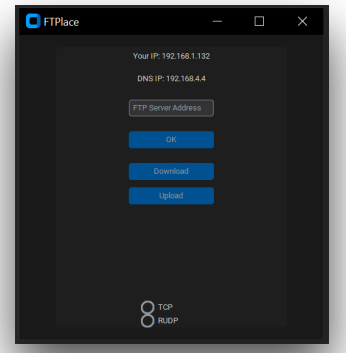
**DNS**

```
(*) Starting DNS server...
(*) Waiting for DNS request...
```

2) Next, run the FTP_server.py (in the Application folder).

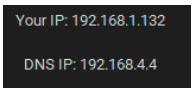
```
(*) Starting application server...
(+) Binding was successful with protocol socket.
(*) Waiting for the user to choose protocol for communication...
```

3) Finally, run the FTP_client.py (in the Application folder).
   - All files in the Graphical_interface don't need to run manually since they are part of the GUI, and run automatically when you run the FTP_client.py.




## **Step 3:** Testing program.

When running the FTP_client.py, the GUI should open up immediately after receiving network configuration from the DHCP server.
This configuration would be presented on the GUI window and on the terminal:




First, choose what type of communication do you seek with the server: TCP or RUDP.


Then, the client will notify of the chosen connection to the server:

**Client**

```
*******************************
(+) Binding was successful.
(+) Sent the TCP communication protocol to the server.
```

**Server**

```
(*) The user choose to communicate using TCP.
```

Next, Enter the domain you want to connect to on the server and press "OK".
This will send a DNS query to the DNS server and the server will send a respond:

**Client**

```
********************************
(*) Connecting to DNS server...
(*) Creating DNS request packet.
.
Sent 1 packets.
(+) Sent the DNS request.
(*) Waiting for the DNS response...
(+) Received the DNS response.
(+) DNS answer:  192.168.2.2
```

**DNS**

```
(*) Starting DNS server...
(*) Waiting for DNS request...
(*) Creating DNS response packet.
(+) DNS query was successful.
.
Sent 1 packets.
(+) Sent the DNS response.

(*) Waiting for DNS request...
```

If you enter a wrong domain (that isn't in the DNS server's records), this is what you will get:

**Client**

```
********************************
(*) Connecting to DNS server...
(*) Creating DNS request packet.
.
Sent 1 packets.
(+) Sent the DNS request.
(*) Waiting for the DNS response...
(+) Received the DNS response.
(-) DNS failed. Try again.
```

**DNS**

```
(*) Starting DNS server...
(*) Waiting for DNS request...
(*) Creating DNS response packet.
(-) DNS query failed.
.
Sent 1 packets.
(+) Sent the DNS response.

(*) Waiting for DNS request...
```

Once the DNS sends back a valid response, the client can now connects to the domain.

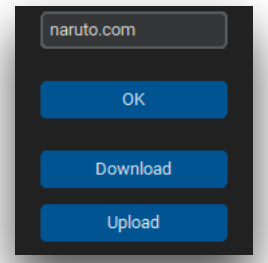
**Client**

```
********************************
(*) Creating UDP socket...
(+) Sent the server the domain.
```

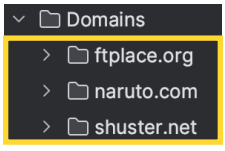
**Server**

```
********************************
(+) Binding was successful with domain socket.
(*) Waiting for the user to enter domain...
(+) Connected to : naruto.com
```

The "Download" and "Upload" buttons should be available:



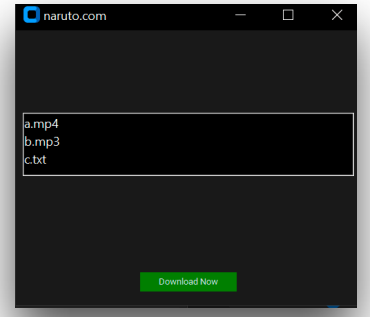

The domain name is one of the following:




- Each domain has a folder in the server (in the Domains folder given in the project) From which you can download files and upload files to.

After entering the domain, the DNS server should provide the client with its ip address.
After that, all communications are done on the local host (127.0.0.1).
The client now connects to the domain on the server via TCP/RUDP and the server is listening.
The client can now choose whether to upload or download files from it.

To Download: Click the “Download" button and a window of the files in the domain folder will be presented while sorted by alphabet.



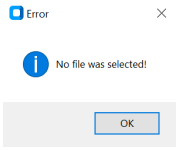

**Client**

```
**********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Connected to the server.
(+) Notified the server we want to download.
(+) Sent request to download: c.txt
(*) Downloading the file...
(+) Done with downloading file.
```

**Server**

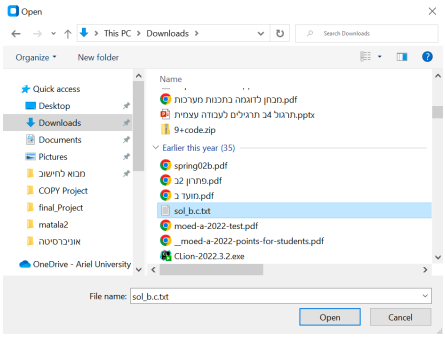
```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a TCP connection and preparing to download...
(+) File name to download: c.txt
(*) Sending the file...
(+) Done with sending file.

(*) Waiting for request...
```

If you choose nothing to download, it would show the following message:

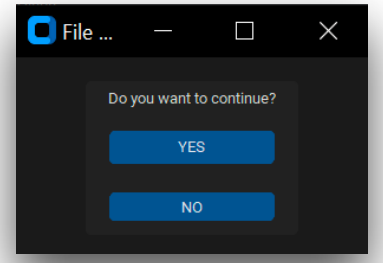



Click the file you want to download, choose the download button and select the path where you want to download and save the file. For example, we choose to download c.txt:

To Upload: Click the “Upload” button, select the file you want to upload and click “open”.




If you choose to upload in TCP, half way you would be asked if to continue the upload or not.



For example, we choose to upload sol_b.c.txt:

**Client**

```
********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Sent the file's size to the server.
(+) Sent the file's name to the server.
(*) Sending the file...
(+) Continuing to upload.
Sent: 0 / 0
(+) Done with sending file.
```

**Server**

```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a TCP connection and preparing to upload...
(+) Received the file's size.
(+) File size: 3604 bytes
(+) Received the file's name.
(+) File name: sol_b.c.txt
(+) Done.

(*) Waiting for request...
```

## **Step 4:** Testing on Wireshark.

To test the program on wireshark, choose you network interface to detect the DNS/DHCP packet, or choose the LoopBack interface to detect all packets for the app.

We included 3 pcapng files which can be opened in Wireshark and monitored.

- 1 file for the DHCP and DNS connection.
- 1 file for the TCP connection between the client - server.
- 1 file for the RUDP connection between the client - server.
- A detailed overview and explanation is given in the end of this pdf.

# General Questions

These are some of the questions we were given in this project.

Questions:

1. Name at least 4 main differences between TCP and QUIC protocols.
2. Name at least 2 main differences between Cubic and Vegas.
3. Explain the BGP protocol, and in what is it different from OSPF protocol? Does it use shortest paths?
4. Based on the code in this project, add the relevant information to the following table based on the message process of your project.
   Explain how these messages are going to change were we to have a NAT between the user to the servers, and would you use QUIC in such a case?

| Application | Port Src | Port Des | IP Src | IP Des | Mac Src | Mac Des |
|---|---|---|---|---|---|---|
| | | | | | | |

5. Explain the differences between ARP and DNS protocols.

Answers:

1. There are several main differences between TCP and QUIC protocols:

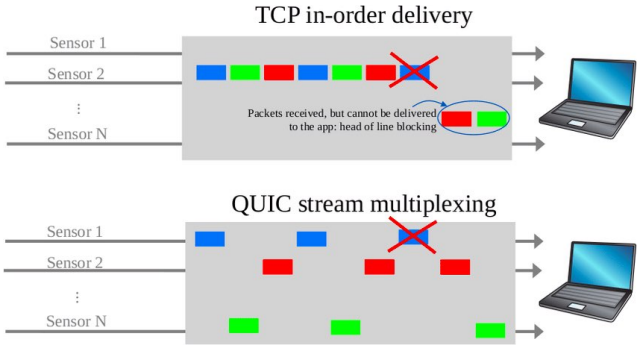
   **1)** - Connection - The TCP protocol uses a 3-way handshake to create a connection between the client and the server. QUIC, however, was built on top of the UDP protocol which is an unreliable protocol and so it only requires 1 packet to establish the connection (+ TLS).

   **2)** - Faster - QUIC is much faster in terms of performances than TCP, rather we are talking about packet loss, high delay and bandwidth, throughput and time for transfer, the QUIC will outperform the TCP.


   **3)** Encryption - The QUIC packets are encrypted each one individually, which removes the decryption delay for partially delivered packets. Furthermore, when a sender opens a connection, QUIC sends a response packet that includes the data needed for future packets that requires encryption. This allows faster speed times and a large decrease in response. However, with TCP you need to set up the TCP connection and then initialize the security protocol through other packets.

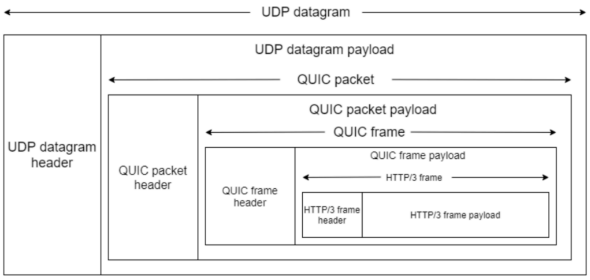
**4)** Security - QUIC gives full end-to-end security by offering encryption in the transport layer (integrated TLS) which authenticates the payload and the headers.
However, TCP is more vulnerable since it doesn't provides this kind of encryption.

**5)** Head-Of-Line-Blocking - TCP presents the head-of-line blocking "problem" which occurs when a packet holds up the entire line of packets behind because of delay.
QUIC solves this by defining separate streams of data within the same connection and provides multiplexing in the transport layer.

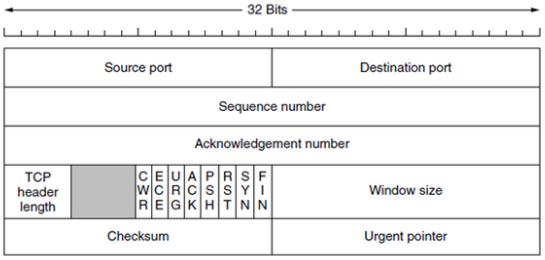



**6)** Headers -

**QUIC**




**TCP**




2. There are few main differences between Cubic and Vegas congestion control algorithms:

**1)** - Congestion detection - Unlike Cubic which detects congestion only after it has happened based on packet loss, Vegas detects congestion based on increasing RTT from one packet to the other in the current connection. Thus, it detects congestion in the network even before a packet loss occurs, and decreases the window size accordingly.

**2)** - Window size - Cubic uses a binary search algorithm that govern the growth of the congestion window. While Cubic continues increasing its window size until packet loss is detected (where it cuts in half the size of the windows and increases slowly when approach that value), Vegas increases or decreases its sending window size based on the RTTs (Round Trip Times) of sending packets.

**3)** - Optimization - Vegas can't co-exist well with other CC algorithms since it is the fastest to detect congestion and throttle the connection, before other algorithms that use packet loss as the main indication.

3. **BGP (** Border Gateway Protocol) - is a data routing protocol between autonomous systems. In this protocol, every router keeps route to every other linking router (and not the distance itself).
Every period of time, the router shares his data with his neighbors.
The neighbors go through the data and update their routes if they found a better one.
The BGP doesn't decide which routes will be chose, and this decision is left for the manager of the autonomous area.
2 routers can communicate using 4 types of messages the BGP allows:

*OPEN* - Opening a TCP connection between 2 routers.
*KEEPALIVE* - A message that is sent when opening a connection is successful.
*UPDATE* - Used to provide data on the route.
*NOTIFICATION* - Notifies of an error in last message or closing the connection.

The BGP is different than OSPF in few aspects:
- OSPF is an internal gateway protocol while BGP is an external gateway protocol.
- OSPF uses IP protocol to communicate and BGP uses TCP protocol.
- OSPF focuses on the distances and uses Dijkstra algorithm,
Whereas BGP focuses on the routes themselves and uses Best path algorithm.
- OSPF usually used on small scale network and BGP on large scale networks.

The routers can chose routes to destination based on policy decision, hot potato routing or even shortest AS-PATH, meaning this algorithm does use shortest path.

4. DHCP:

| Application | Port Src | Port Des | IP Src | IP Des | Mac Src | Mac Des |
|---|---|---|---|---|---|---|
| DHCP | 68 | 67 | 192.168.1.x | 192.168.1.x | 0a:00:27:00:00:12 | ff:ff:ff:ff:ff:ff |
| | | | 0.0.0.0 -> | 255.255.255.255 | | |

DNS:

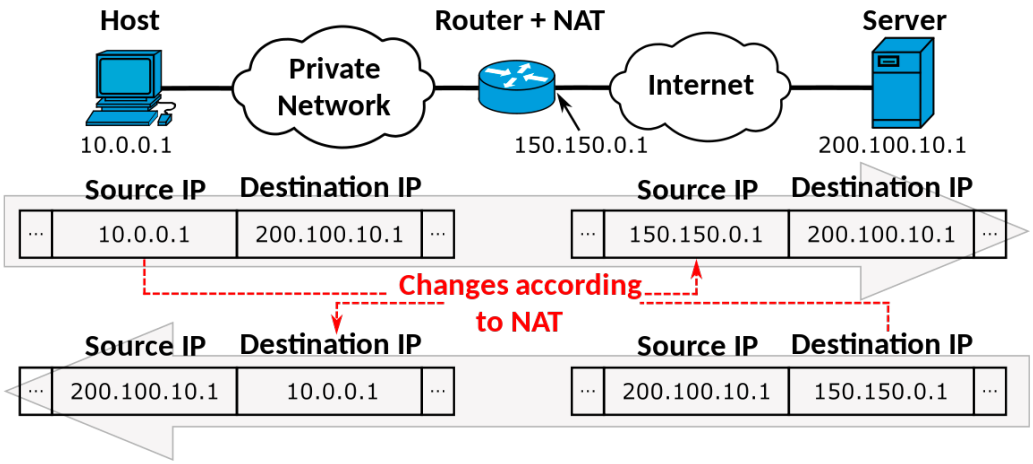
| Application | Port Src | Port Des | IP Src | IP Des | Mac Src | Mac Des |
|---|---|---|---|---|---|---|
| DNS | 1024 | 53 | 192.168.1.x | 192.168.4.4 | 08:d2:3e:fe:11:1d | 20:b0:01:32:58:a6 |

Client-Server:

| Application | Port Src | Port Des | IP Src | IP Des | Mac Src | Mac Des |
|---|---|---|---|---|---|---|
| Client-Server | 20781 | 30413 | 127.0.0.1 | 127.0.0.1 | | |

If NAT protocol were to be involved in our project, the tables above will be changed.
NAT protocol stands for Network Address Translation.
It is a technique used to map a private IP address to a public IP address of IP packets while in transit via a router.
In our project, we received a (private) IP, but with NAT there will be a new one that represents the public one.
If we had several clients the NAT could merge all the private IPs to single public IP
This is how things will change in the messages:

**Port Src** - NAT will generate a random port number which will replace the source port of the client.
**Port Des** - NAT will use the senders port number to replace the destination port number.
**IP Src** - NAT will use the public IP address of the NAT system to replace the private IP address of the client.
**IP Des** - The IP destination address should stay the same.
**MAC source and destination addresses** - NAT shouldn't affect the MAC addresses, Since they are part of the local network, and they don't take part in routing packets across the internet.




Because our project is about FTP, the protocol QUIC wouldn't be useful and won't benefit us much, because we implemented TCP / RUDP protocol that promises the file get/ send correctly with not a lot of risk, meaning - reliably.
But if we were ask to handle security too, then we would consider using QUIC because it has his built-in encryption by default.

5. The differences between ARP and DNS protocols:

   **1)** - Operation - While the ARP protocol translates IPv4 to Ethernet addresses, DNS translates domain names to IP addresses.

   **2)** - Filed Of Responsibility - While the DNS protocol operates anywhere online, ARP resolves IP addresses of hosts within the same subnet (operates locally).

   **3)** - Layers - While the ARP protocol runs in the Link layer, the DNS protocol runs in the Application layer.

   **4)** - Goals - While the ARP protocol is used for discovering the link layer address, The DNS protocol is used for computing resources connected to the network.

   **5)** - Packet - While the ARP protocol uses a type of packet called an ARP packet, (which is encapsulated within the data link layer frame), to send ARP request/reply, The DNS uses UDP/TCP packets to in its server/client communication (which includes DNS query, response, update and notify packets).

# DHCP

- Since we already explained widely in the code using comment, we will only explain here the defined variables/methods and the code itself briefly.
- We'll explain the code and methods both from the client and the DHCP server aspect.
- We used the Scapy library to construct, send, sniff and extract info from the DHCP packet.

Defined Variables:

```
DHCP_CLIENT_PORT = 68
DHCP_SERVER_PORT = 67
```

- DHCP_CLIENT_PORT - The client's port.
- DHCP_SERVER_PORT - The DHCP server's port.

Methods In Client:

`connectDHCP()` – A method that its only purpose is to connect to the DHCP server, and get a network configuration for the FTP client.
It does so by first creating and sending a DHCP Discovery packet as broadcast to reveal and find our DHCP server, then it waits for the DHCP Offer from the server which contains our new IP address, the DHCP server's IP, subnet mask and the our DNS server's IP (which is 192.168.4.4).
It then creates and sends a DHCP Request to the server to let him know that we are interested in the offers address. Finally, we wait for the DHCP ACK packet from the DHCP server, and return the client's new IP and its DNS IP (which we present to the client in our app).

Main Methods In DHCP Server:

`create_offer()` – A method to create and send the client a DHCP Offer packet.

`create_ack()` – A method to create and send the client a DHCP ACK packet.

Sub Methods In DHCP Server:

`generate_random_ip(list ip_list)` –
A method that generate a random IP of the form 192.168.1.$x$ where $0 \leq x \leq 255$.
This IP will serve as an offer for the DHCP clients or as the IP of the DHCP server itself.
Once it generates an IP, we add it to the `ip_list` so that we keep track and won't generate and giveaway the same IP twice.

`generate_highest_ip(list ip_list)` –
A method that generate an IP of the form 192.168.1.$x$ where $0 \leq x \leq 255$,
but this time $x$ goes from $255$ to $0$ in decreasing order until reaching available address.
This is because this methods generate a broadcast IP meaning the IP the client will use when contacting the DHCP server. Using `ip_list` we keep track of the used IP's and if no available IP was found, we return the IP address: 255.255.255.255.

Code (DHCP Server):

First we run the DHCP.py file. The DHCP server is up and does as follow in the main:

- Create an list called “IP_LIST” to hold all the IP’s that are generated locally in this code to avoid a scenario where 2 user receives the same IP/Broadcast IP or receive the DHCP server’s IP.
- Generates the DHCP server’s IP using generate_random_ip() and prints it.
- Start a while True loop to accept new clients and in the loop:
  - Generates the client’s offered IP using generate_random_ip() and prints it.
  - Generates the client’s broadcast IP using generate_highest_ip() and prints it.
  - Waits for the DHCP Discovery packet from the client.
  - Calls create_offer() to create and send the DHCP Offer packet.
  - Waits for the DHCP Request packet from the client.
  - Calls create_ack() to create and send the DHCP ACK packet.
  - Delete the generated client's broadcast IP from the “IP_LIST” since the current client is finished and this address is no longer in use.
  - Sleep for 2 seconds to wait for the next client to connect.

When the client runs the FTP_client.py, in the main function it first calls connectDHCP(), which as we described earlier communicates with the DHCP server and helps getting the client network configuration.

Example of the process in the terminal:

**Client**

```
(*) Creating DHCP discover packet.
(*) Sending DHCP discover...
.
Sent 1 packets.
(*) Waiting for DHCP offer...
(+) Got a DHCP offer packet.
(*) Creating DHCP request packet.
.
Sent 1 packets.
(+) Sent DHCP request.
(*) Waiting for DHCP ACK...
(+) Got a DHCP ACK packet.
```

**DHCP**

```
(*) Starting DHCP server...
(+) Server IP:  192.168.1.33
(+) Next Client IP suggestion:  192.168.1.132
(+) Next Client IP broadcast:  192.168.1.255

(*) Waiting for DHCP discovery...
(+) Got a DHCP discovery packet.
(*) Creating DHCP offer packet...
(+) Sending DHCP offer.
.
Sent 1 packets.
(*) Waiting for DHCP request...
(+) Got a DHCP request packet.

(*) Creating DHCP ACK packet...
(+) Sending DHCP ACK.
.
Sent 1 packets.
(+) Next Client IP suggestion:  192.168.1.185
(+) Next Client IP broadcast:  192.168.1.255

(*) Waiting for DHCP discovery...
```

In this example, The DHCP server’s IP is 192.168.1.33 (and will stay that way till the next time we run the DHCP server). The DHCP server offers our client the IP address; 192.168.1.132 and the broadcast IP: 192.168.1.255 (which is the highest of this format).

This is presented later when the GUI opens up: `Your IP: 192.168.1.132`

When the process is done, the DHCP server is already ready with the IP’s is going to offer to the next client that is going to connect to him, that is, send him DHCP Discovery packet.

- This makes the process easy when we want to check the code and since we don't need to turn off and on the DHCP server.

# DNS

- Since we already explained widely in the code using comment, we will only explain here the defined variables/methods and the code itself briefly.
- We'll explain the code and methods from the client, GUI and the DNS server aspect.
- We used the Scapy library to construct, send, sniff and extract info from the DHCP packet.

Defined Variables:

```
DNS_IP = '192.168.4.4'
CLIENT_PORT = 1024
DNS_PORT = 53

# List of domains and their IP address (All local)
Domains = {
    'ftplace.org': '192.168.2.1',
    'naruto.com': '192.168.2.2',
    'shuster.net': '192.168.2.3'
}
```

- DNS_IP - The DNS Server's IP. We define it to be 192.168.4.4.
- CLIENT_PORT - The client's port.
- DHCP_SERVER_PORT - The DHCP server's port.
- Domains - A list of domains held in the DNS server and their IP's. This is our DNS records.

Methods In GUI:

`connectDomain()` – A method that is called each time the user clicks on the "OK" button.
The method calls connect connectDNS() to receive an IP if the domain entered is valid,
or None if not. If it returned an IP, the method opens a UDP socket and connects to the domain in the servers by letting the server know which domain the user is interested in.

`GetDomain()` – A method that is meant to receive and return the domain name that is entered.

`enable_buttons()` – A method that enables the Upload/Download button.
This method is called whenever the DNS returns a valid response and we can connect to the server.

`clear_entry()` – A method that clears the domain text box in the GUI.
This method is called whenever the DNS returns a non-valid response.

Methods In Client:

`connectDNS (object gui_object, string client_ip, string dns_ip)` –
A method that its sole purpose is to connect to the DNS server,
and get the IP for the domain the client wants to connect to.
It receives the client's IP to know who will be the source IP in the DNS query packet,
and the DNS IP to know who is the destination. The method gets whatever the user entered as domain,
and creates and send the DNS query Request packet using Scapy.
After that, it waits and sniffs the DNS Response and check whether its a valid response, meaning the DNS returned an IP for the domain. In such case, the method returns the IP and enables the buttons.
Otherwise, its an invalid response, meaning the DNS server didn't find the domain in its records.
In such case, the method returns None.

Code (DNS Server):

First we run the DNS.py file. The DNS server is up and does as follow in the main:

- Start a while True loop to accept new clients and in the loop:
  - Sniffing the DNS Request packet which was sent from the client.
  - Extracting from it the client's IP (to know to where to send the DNS Response), and the domain name the client request the IP for.
  - Creating the DNS Response packet. We divide this part into 2 case:
    1) The domain is in the DNS Server's records.
       We create the response as non recursive, authoritative, type A, with ID 0xABCD, with valid response and in the rdata of the packet we put the IP answer.
    2) The domain isn't in the DNS Server's records.
       We create the response as non recursive, authoritative, type A, with ID 0xABCD, with invalid response and in the rdata of the packet we put None.

  In each case, we print whether the DNS query succeeded or failed.

  Either way, we construct and send he DNS Response packet back to the client.

# FTP Client + GUI

- Since we already explained widely in the code using comment, we will only explain here the defined variables/methods and the code itself briefly.
- We won't explain about the RUDP methods since we already did that in the project questions when we explained how our system overcomes delay/packet loss.

Defined Variables:

```
MAX_BYTES = 4096
DHCP_CLIENT_PORT = 68
DHCP_SERVER_PORT = 67
DNS_CLIENT_PORT = 1024
DNS_SERVER_PORT = 53
CLIENT_PORT = 20781
SERVER_PORT = 30413
PACKET_SIZE = 1024
WINDOW_SIZE = 4
TIMEOUT = 2
CC_RENO = b"reno"
LOCAL_IP = '127.0.0.1'
SERVER_ADDRESS = ('localhost', SERVER_PORT)  # A tuple to represent the server.
lock = threading.Lock()  # Lock for threading (receiving messages).
window_start = 0  # Starting index for the window.
next_seq = 0  # Next packet to send.
```

- MAX_BYTES - Maximum bytes to be received on the socket.
- CLIENT_PORT - The client's port. We define It to be 20xxx where xxx is the last 3 digits of our ID.
- SERVER_PORT - The server's port. We define It to be 30xxx where xxx is the last 3 digits of our ID.
- PACKET_SIZE - Default packet size to receive on the socket.
- WINDOW_SIZE - Constant window size for uploading and downloading via RUDP protocol.
- TIMEOUT - Timeout for the socket for uploading and downloading via RUDP protocol.
- CC_RENO - A cognition control (CC) method. (relevant for TCP with permission).
- LOCAL_IP - Represents the local IP 127.0.0.1.
- SERVER_ADDRESS = A tuple to represent the server info.
- Lock - Lock for threading (When receiving messages - reduce printing simultaneously).
- window_start - Starting index for the window.
- next_seq - An index of the next packet to be sent (in RUDP).

Methods In Download.py:

`createWindow(object self, string protocol)` –
Creates the Download window where we define the protocol the user chose as `protocol`, creates, sorts and presents the list of files in the chosen domain folder, and creates a download button that upon pressing it calls downloadNow().

`downloadNow(object self)` –
Called when clicking on the downloadNow button. Check which file from the domain folder the client choose. If no file was chosen, presents an error window. Otherwise, calls the appropriate function with the path and file name (based on the chosen protocol) to receive the file from the server.

Methods In GUI.py:

`chooseRUDP(object self)` –
The client choose RUDP protocol. We call sendCommunicationType with “RUDP”.

`chooseTCP(object self)` –
The client choose TCP protocol. We call sendCommunicationType with “TCP”.

`download_win(object self)` –
Creates an object of type Download (as mention above) and running it.
It would open the download window from which the client can interact.

`upload_win(object self)` –
As the method runs, we open a file dialog to ask the client which file he wants to upload.
if he doesn’t choose a file, we pass. Otherwise, we have the file_path and we check which protocol the client choose and act accordingly. If he choose TCP, we call uploadToServerTCP() with the file_path.
Otherwise, we create UploadType object that presents a window where the client can choose regular, delay or packet loss type of send.
Accordingly, we call uploadToServerRUDP() with the file_path and the sending function he chose.

Methods In Client:

`downloadFromServerTCP(string file_name, string save_path)` –
A method that create a TCP socket and receives the file from the server.
First, a window in the GUI will pop, the user selects the file he wants to download that will be the `file_name` parameter to the function.
Then, we send the server that we want to download and the requested `file_name`.
After that we receive the file chunks and create the file in the path we choose.
Eventually, we close the file and the client socket.

`uploadToServerTCP(object gui_object, string file_path)` –
A method that create a TCP socket and sends the file to the server.
First, we update the server we want to upload, then we send him the file size and name,
and eventually we read the file we want to upload and mid way ask him if he wants to stop the upload or continue. Eventually, we close the file and the client socket.

Code:

In the main function we call connectDHCP() to get configuration,

then we create the GUI object and pass it the client’s IP and DNS IP,

then we create the GUI window and run it for the user to play around and interact.

# FTP Server

- Since we already explained widely in the code using comment, we will only explain here the defined variables/declared methods and the code itself briefly.
- We won't explain about the RUDP methods since we already did that in the project questions when we explained how our system overcomes delay/packet loss.

Defined Variables:

```
MAX_BYTES = 8192
CLIENT_PORT = 20781
SERVER_PORT = 30413
PACKET_SIZE = 1024
WINDOW_SIZE = 4
TIMEOUT = 2  # In seconds
CC_CUBIC = b"cubic"
LOCAL_IP = '127.0.0.1'
lock = threading.Lock()  # Lock for threading (receiving messages).
window_start = 0  # Starting index for the window.
next_seq = 0  # The sequence number of the next expected packet.
```

- MAX_BYTES - Maximum bytes to be received on the socket.
- CLIENT_PORT - The client's port. We define It to be 20xxx where xxx is the last 3 digits of our ID.
- SERVER_PORT - The server's port. We define It to be 30xxx where xxx is the last 3 digits of our ID.
- PACKET_SIZE - Default packet size to receive on the socket.
- WINDOW_SIZE - Constant window size for uploading and downloading via RUDP protocol.
- TIMEOUT - Timeout for the socket for uploading and downloading via RUDP protocol.
- CC_CUBIC - A cognition control (CC) method. (relevant for TCP with permission).
- LOCAL_IP - Represents the local IP 127.0.0.1.
- Lock - Lock for threading (When receiving messages - reduce printing simultaneously).
- window_start - Starting index for the window.
- next_seq - An index of the next packet to be sent (in RUDP).

Methods In Client:

`downloadTCP()` – A method that's getting enveloped once the user chose to download a file from the server using TCP socket, meaning the client sent the server "download" message and selected "TCP" protocol. The server receives first the file name he wants to download, going to the proper directory and sending the client the requested file.

`uploadTCP()` – A method that's getting enveloped once the user chose to upload a file from the server using TCP socket, meaning the client sent the server "upload" message and selected "TCP" protocol. First, we are getting the file size and name, and creating the proper file directory to where the user wants the file to be upload. Then the server write the file to the specific directory.

Code:

In the main function we first receive the protocol and the domain the client wants to connect to from the user using UDP socketing. If he choose RUDP, we create a UDP socket and wait each time for the client to choose whether to upload/download with Reliable UDP. If he choose TCP, we create a TCP socket and also wait for upload/download requests to commit with TCP.

# Project Questions

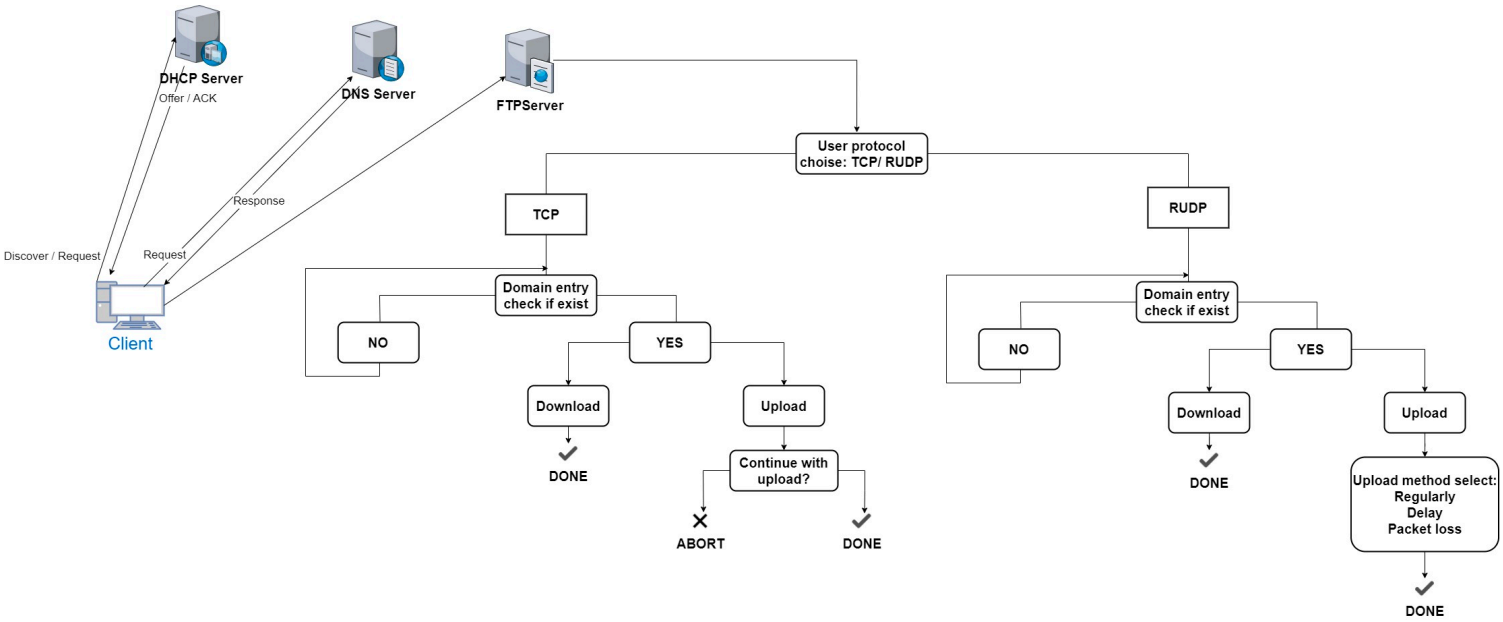
These are some of the questions we were given in this project.

Questions:

1. Draw a state-diagram for the system.
2. How does the system overcomes packet-loss?
3. How does the system overcomes latency problems?

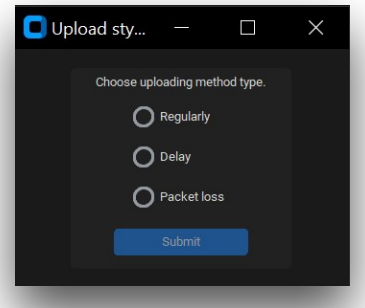
Answers:

1. The Diagram:




2. In our system, we simulate a state of packet loss when uploading In RUDP.

As the user choose to upload in RUDP, it shows him a windows where he can choose in which environment he wants to upload:




After selecting “Packet loss”, all of the data that is uploaded to the server is uploaded in packet loss mode, meaning some packets are sent and some are not based on probability (just like in exercise 5).
We generate a number $x$ where $0 \leq x \leq 1$, and if $x > 0.5$ we send the packet. Else, we don't send it

How does our code deals with this packet lose:
In our code, we implement the Go-Back-N protocol,  a method for transmitting data over an unreliable network, to ensure that the file will be received completely without loss or corruption.

For this, we created a few variables to help us in the process:

We create a thread that will receives ACKs parallel.

- **Starting window index**: To indicate where the window starts in the buffer.

- **Number of chunks**: For tracking how many packets we need to send.

- **Next sequence**:  initialized to 0 and tells us what is the next packet to be sent. This helps us to keep and send in the right order.

- **Window Size:** We define it to be 4.

- **Thread**: Which listens for ACK's or DUP ACK's from the server.

On client side:

While we still got packets to send, each iteration start a loop that its purpose it to send all packets currently in the window.

We check each time if the next sequence number is less then the number of chunks. This is to double check there are still packets to be sent, and so we are sending the requested packet with the function for packet loss we have mentioned above.

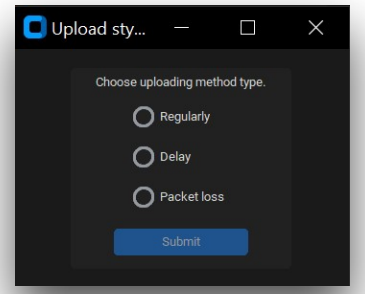
The thread meanwhile listens for ACK's. Each ACK accepted, the server checks the sequence number the ACK is for.
If the ACK is for an index equal or larger then the starting window index, it means its an ACK for a packet we haven't seen before and so we accept it and update the starting window index to be it.
If the ACK is for an index smaller then the starting window index, this means its an ACK for a packet we've seen before meaning its DUP ACK and we need to resent the packet with the sequence number after it.
If no ACK were accepted and a timeout occurs, it either because we need to resend the packet in the beginning of the window, or we done uploading everything.

On the server side:

We receive the packets of the file. While the next sequence number we expect is less then the expected number of chunks, this means there are still packets that need to be sent.
Then, we check each time if the sequence number we received suits the sequence number we expect it to be. if so, we will increase the next sequence number by one and send an ACK with the current sequence number.
Else, we will notify the sender we got packet out of order by sending him DUP ACK and the current sequence number -1.

3. In our system, we simulate a state of latency when uploading In RUDP.

As the user choose to upload in RUDP, it shows him a windows where he can choose in which environment he wants to upload:




After selecting “Delay”, all of the data that is uploaded to the server is uploaded in a delay of 2 seconds (we use the sleep python method before sending).

Our system overcomes this delay issue by setting a timeout to the socket of two seconds, so that in case of that latency problem we created, the socket timer should be triggered and a timeout should occur.
In case of delay and timeout we don't "forget" about the packet and we just send it again and again till it be received in the other side, then continuing with the upload.

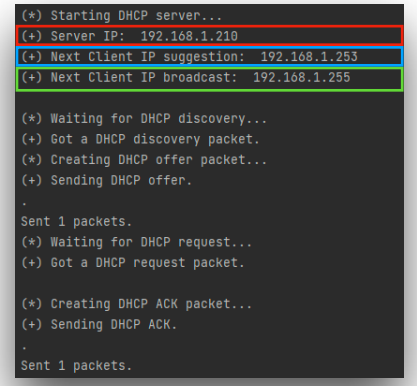
# Tests On Terminal And WireShark

DHCP:

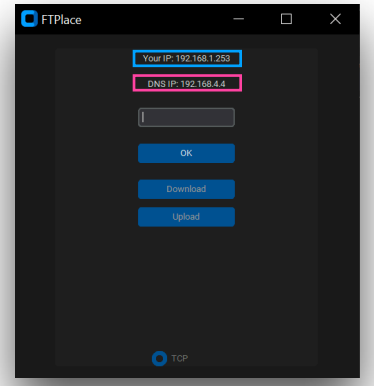
- We send the messages on port 68 and port 67 (the ports for DHCP communication).
- To see the DHCP communication, please filter by “dhcp”.
- We recommend you turn off you Wi-Fi/Ethernet so that the only communication you’ll see is of the client and our DHCP server.

First, we will look at the terminal and GUI output when we first ran the program:

**DHCP**




**GUI**




Let’s explain the colors:

Red - The server’s IP. It will be the source IP for the messages from the DHCP server.

Blue - What the DHCP server offered the client. We will see it inside the DHCP Offer/Request.

Green - The broadcast IP of the client.

It will be the destination IP for the messages from the DHCP server.

Pink - The DNS server’s IP. We fixed it in our code to be 192.168.4.4.

Now, let’s see how this is relevant when we examine it in the wireshark:

dhcp

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 0.0.0.0 | 255.255.255.255 | DHCP | 286 | DHCP Discover - Transaction ID 0xa2c2a |
| 2 | 0.002741 | 192.168.1.210 | 192.168.1.255 | DHCP | 310 | DHCP Offer - Transaction ID 0xa2c2a |
| 3 | 0.005312 | 0.0.0.0 | 255.255.255.255 | DHCP | 304 | DHCP Request - Transaction ID 0xa2c2a |
| 4 | 0.322468 | 192.168.1.210 | 192.168.1.255 | DHCP | 286 | DHCP ACK - Transaction ID 0xa2c2a |

Before we continue, we can see the DHCP Discover and Request packets are sent From 0.0.0.0 to 255.255.255.255.

0.0.0.0 - Indicates that the client doesn’t have an IP yet and isn’t connected to the network.

255.255.255.255 - The broadcast IP for the server to locate DHCP servers.

In the next page, we will examine the important DHCP packets which are the DHCP Offer (the offer the DHCP server sends to the client) and the DHCP Request (which is the request

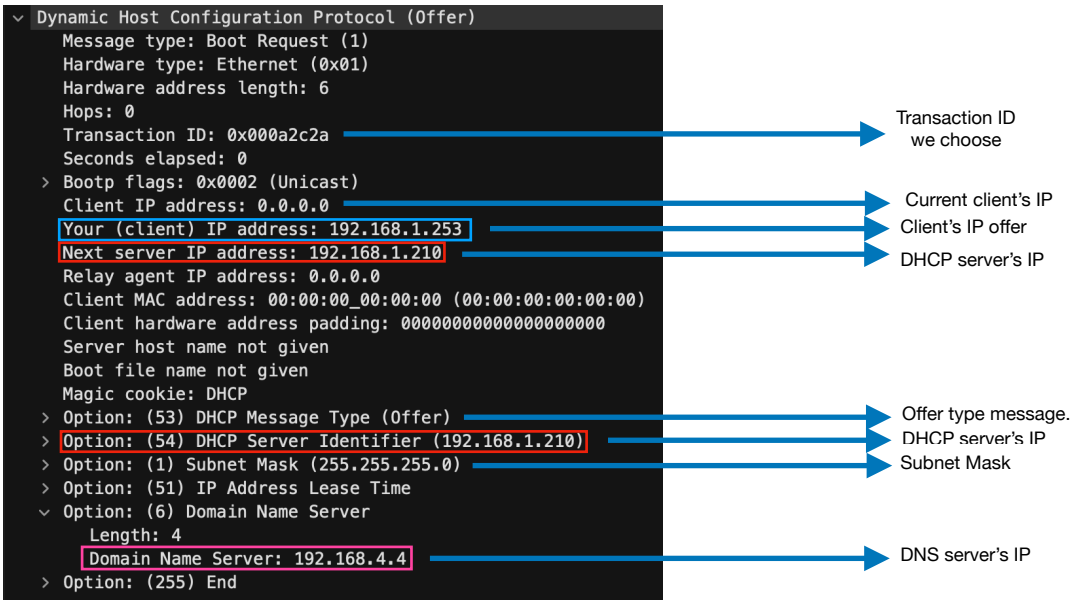
from the client that he is interested in the offer).

We will examine only the DHCP layer.

**DHCP Offer:**

In here, the DHCP send configuration to the client as an offer.

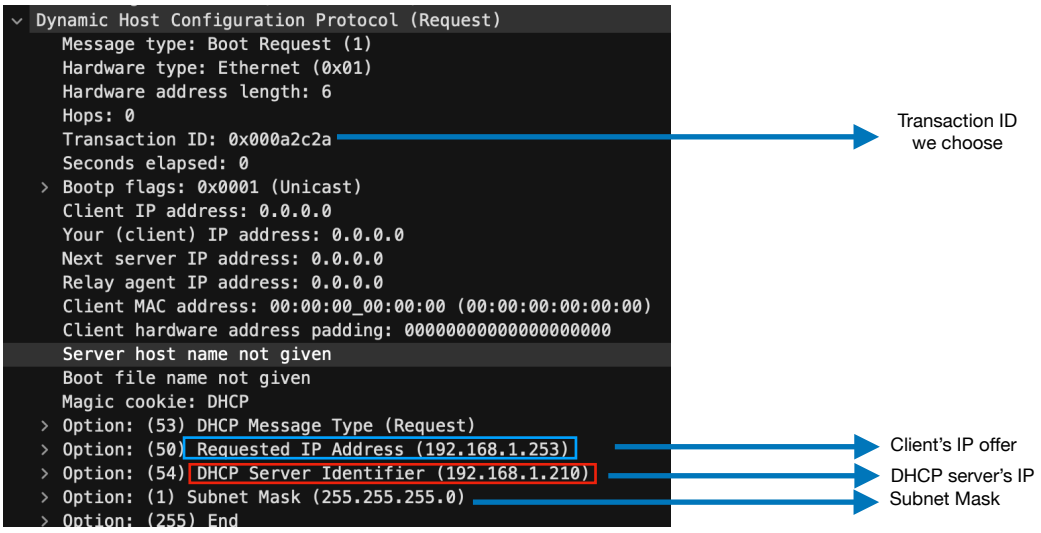
We can see how the parameters shown earlier match here.




**DHCP Request:**

In here, the client request the DHCP configuration the server sent him before.

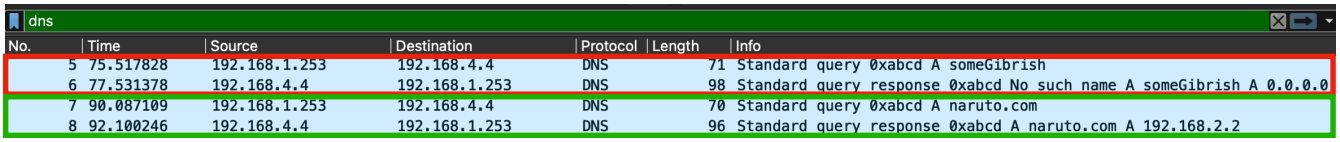
We can see how the parameters shown earlier match here.



## DNS:

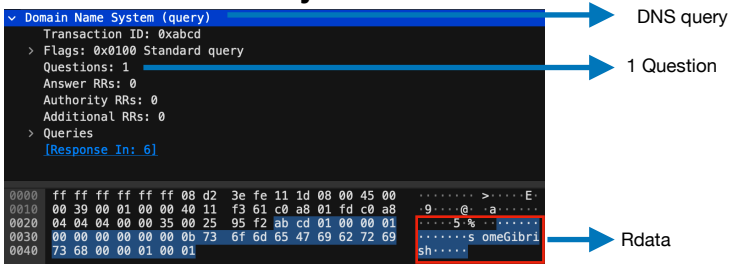
- We send the messages on port 1024 and port 53 (the port for DNS communication).
- To see the DNS communication, please filter by “dns”.
- We recommend you turn off you Wi-Fi/Ethernet so that the only communication you’ll see is of the client and our dns server.
- We gave the DNS server a fixed IP: 192.168.4.4.
- From the DHCP Offer from before, we now know that the client’s IP is: 192.168.1.253.

In our program, we first entered “someGubrish” which resulted an invalid response.
That’s because such domain doesn’t exist in our DNS server.
After that, we entered “naruto.com” which is a real domain that resulted a valid response.
Let’s take a look in the wireshark monitoring:

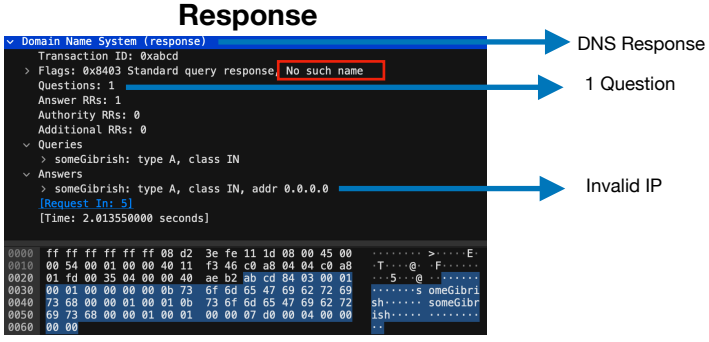



We will examine each part:
Red - As said, in this part the client sent a wrong DNS query to the DNS server.
The DNS server sent a DNS Response that said that such a name doesn’t exist.

**Query**

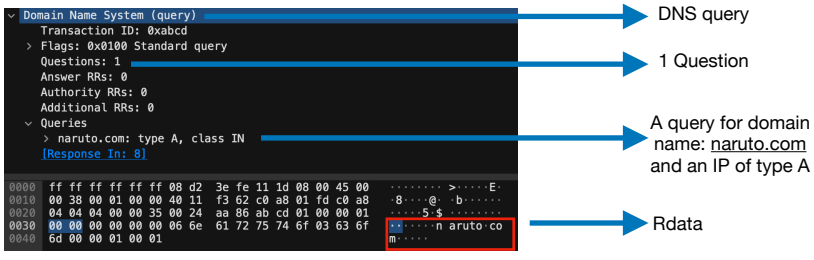



**Response**

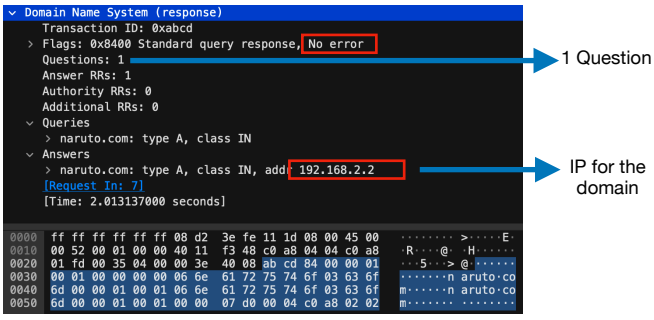



Green - As said, in this part the client sent a valid DNS Request to the DNS server.
The DNS server sent a DNS Response with the domain’s IP.

**Query**




**Response**




As we can see here, the DNS server repented with the right IP in its database: `'naruto.com': '192.168.2.2',`

FTP - TCP:

- We send the messages on port 20781 (20xxx) and port 30413 (30xxx).
- To see the communication, please filter by

  “udp.port == 30413 || tcp.port == 30413”
- To capture this communication, go into the LoopBack interface.
- We will try and examine this communication part by part.
- Since this is TCP communication, each time we communicate with the server starts with 3 way handshake and ends with FIN-ACK.

In here, we started the app and choose the “TCP” communication. 

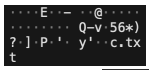
Then, we connected to the “naruto.com” domain and updated the server. 

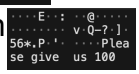

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0.000000 | 127.0.0.1 | 127.0.0.1 | UDP | 35 20781 → 30413 Len=3 |
| 2 | 8.957055 | 127.0.0.1 | 127.0.0.1 | UDP | 42 65430 → 30413 Len=10 |

Part 1 - Downloading from the server:

In here, we choose to download “c.txt” from the domain.

The client first sends a PSH-ACK message to tell the server we want to download, 

then he sends him the file name we are interested to download, 

then the server sends the file to the client and we close the connection 

This is the terminal output:

**Client**

```
**********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Connected to the server.
(+) Notified the server we want to download.
(+) Sent request to download: c.txt
(*) Downloading the file...
(+) Done with downloading file.
```

**Server**

```
*******************************
(*) Creating the server socket (TCP)...
(+) Binding was successful.

(*) Waiting for request...
(+) Client choose to download.

<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a TCP connection and preparing to download...
(+) File name to download: c.txt
(*) Sending the file...
(+) Done with sending file.

(*) Waiting for request...
(+) Client choose to upload.
```

This is the WireShark output:

| | | | | | |
|---|---|---|---|---|---|
| 5 | 24.110954 | 127.0.0.1 | 127.0.0.1 | TCP | 56 20781 → 30413 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 6 | 24.111015 | 127.0.0.1 | 127.0.0.1 | TCP | 56 30413 → 20781 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 7 | 24.111055 | 127.0.0.1 | 127.0.0.1 | TCP | 44 20781 → 30413 [ACK] Seq=1 Ack=1 Win=2619648 Len=0 |
| 8 | 24.111092 | 127.0.0.1 | 127.0.0.1 | TCP | 52 20781 → 30413 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=8 |
| 9 | 24.111108 | 127.0.0.1 | 127.0.0.1 | TCP | 44 30413 → 20781 [ACK] Seq=1 Ack=9 Win=2619648 Len=0 |
| 11 | 24.316602 | 127.0.0.1 | 127.0.0.1 | TCP | 49 20781 → 30413 [PSH, ACK] Seq=9 Ack=1 Win=2619648 Len=5 |
| 12 | 24.316679 | 127.0.0.1 | 127.0.0.1 | TCP | 44 30413 → 20781 [ACK] Seq=1 Ack=14 Win=2619648 Len=0 |
| 13 | 24.523148 | 127.0.0.1 | 127.0.0.1 | TCP | 62 30413 → 20781 [PSH, ACK] Seq=1 Ack=14 Win=2619648 Len=18 |
| 14 | 24.523178 | 127.0.0.1 | 127.0.0.1 | TCP | 44 20781 → 30413 [ACK] Seq=14 Ack=19 Win=2619648 Len=0 |
| 15 | 24.523389 | 127.0.0.1 | 127.0.0.1 | TCP | 44 30413 → 20781 [FIN, ACK] Seq=19 Ack=14 Win=2619648 Len=0 |
| 16 | 24.523471 | 127.0.0.1 | 127.0.0.1 | TCP | 44 20781 → 30413 [ACK] Seq=14 Ack=20 Win=2619648 Len=0 |
| 17 | 24.524343 | 127.0.0.1 | 127.0.0.1 | TCP | 44 20781 → 30413 [FIN, ACK] Seq=14 Ack=20 Win=2619648 Len=0 |
| 18 | 24.524413 | 127.0.0.1 | 127.0.0.1 | TCP | 44 30413 → 20781 [ACK] Seq=20 Ack=15 Win=2619648 Len=0 |

## Part 2 - Uploading to the server and stopping midway:

In here, we choose to upload “c.txt” to the domain.

The client first sends a PSH-ACK message to tell the server we want to upload, 

then he sends him the file’s size and name we are interested to download,  

then the client chooses to stop the upload and the connection is closed.

**Client**

```
**********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Sent the file's size to the server.
(+) Sent the file's name to the server.
(*) Sending the file...
(-) Stopping upload.
```

**Server**

```
(*) Waiting for request...
(+) Client choose to upload.

<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a TCP connection and preparing to upload...
(+) Received the file's size.
(+) File size: 18 bytes
(+) Received the file's name.
(+) File name: c.txt
(+) Done.
```

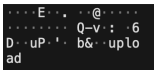
This is the WireShark output:

| | | | | | | |
|---|---|---|---|---|---|---|
| 398 | 193.831693 | 127.0.0.1 | 127.0.0.1 | TCP | 56 | [TCP Port numbers reused] 20781 → 30413 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 399 | 193.831750 | 127.0.0.1 | 127.0.0.1 | TCP | 56 | 30413 → 20781 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 400 | 193.831827 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [ACK] Seq=1 Ack=1 Win=2619648 Len=0 |
| 401 | 193.831831 | 127.0.0.1 | 127.0.0.1 | TCP | 50 | 20781 → 30413 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=6 |
| 402 | 193.831868 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=7 Win=2619648 Len=0 |
| 403 | 194.040438 | 127.0.0.1 | 127.0.0.1 | TCP | 46 | 20781 → 30413 [PSH, ACK] Seq=7 Ack=1 Win=2619648 Len=2 |
| 404 | 194.040513 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=9 Win=2619648 Len=0 |
| 405 | 194.240383 | 127.0.0.1 | 127.0.0.1 | TCP | 49 | 20781 → 30413 [PSH, ACK] Seq=9 Ack=1 Win=2619648 Len=5 |
| 406 | 194.240436 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=14 Win=2619648 Len=0 |
| 407 | 198.032248 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [FIN, ACK] Seq=14 Ack=1 Win=2619648 Len=0 |
| 408 | 198.032283 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=15 Win=2619648 Len=0 |
| 409 | 198.032515 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [FIN, ACK] Seq=1 Ack=15 Win=2619648 Len=0 |
| 410 | 198.032547 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [ACK] Seq=15 Ack=2 Win=2619648 Len=0 |

Reusing ports.

## Part 3 - Uploading to the server:

In here, we choose to upload “c.txt” to the domain fully.

The client first sends a PSH-ACK message to tell the server we want to upload, 

then he sends him the file’s size and name we are interested to download.  

Finally the client sends the file to the server. 

**Client**

```
**********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Sent the file's size to the server.
(+) Sent the file's name to the server.
(*) Sending the file...
(+) Continuing to upload.
Sent: 0 / 0
(+) Done with sending file.
```

**Server**

```
(*) Waiting for request...
(+) Client choose to upload.

<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a TCP connection and preparing to upload...
(+) Received the file's size.
(+) File size: 18 bytes
(+) Received the file's name.
(+) File name: c.txt
(+) Done.
```

This is the WireShark output:

| | | | | | | |
|---|---|---|---|---|---|---|
| 419 | 449.942747 | 127.0.0.1 | 127.0.0.1 | TCP | 56 | [TCP Port numbers reused] 20781 → 30413 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 420 | 449.942868 | 127.0.0.1 | 127.0.0.1 | TCP | 56 | 30413 → 20781 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM |
| 421 | 449.942926 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [ACK] Seq=1 Ack=1 Win=2619648 Len=0 |
| 422 | 449.942980 | 127.0.0.1 | 127.0.0.1 | TCP | 50 | 20781 → 30413 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=6 |
| 423 | 449.943028 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=7 Win=2619648 Len=0 |
| 424 | 450.157468 | 127.0.0.1 | 127.0.0.1 | TCP | 46 | 20781 → 30413 [PSH, ACK] Seq=7 Ack=1 Win=2619648 Len=2 |
| 425 | 450.157534 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=9 Win=2619648 Len=0 |
| 426 | 450.359802 | 127.0.0.1 | 127.0.0.1 | TCP | 49 | 20781 → 30413 [PSH, ACK] Seq=9 Ack=1 Win=2619648 Len=5 |
| 427 | 450.359894 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=14 Win=2619648 Len=0 |
| 428 | 452.633366 | 127.0.0.1 | 127.0.0.1 | TCP | 62 | 20781 → 30413 [PSH, ACK] Seq=14 Ack=1 Win=2619648 Len=18 |
| 429 | 452.633454 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=32 Win=2619648 Len=0 |
| 430 | 452.633777 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [FIN, ACK] Seq=32 Ack=1 Win=2619648 Len=0 |
| 431 | 452.633862 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [ACK] Seq=1 Ack=33 Win=2619648 Len=0 |
| 432 | 452.634546 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 30413 → 20781 [FIN, ACK] Seq=1 Ack=33 Win=2619648 Len=0 |
| 433 | 452.634620 | 127.0.0.1 | 127.0.0.1 | TCP | 44 | 20781 → 30413 [ACK] Seq=33 Ack=2 Win=2619648 Len=0 |

Reusing ports.

FTP - RUDP:

- We send the messages on port 20781 (20xxx) and port 30413 (30xxx).
- To see the communication, please filter by

  “udp.port == 30413 || tcp.port == 30413”
- To capture this communication, go into the LoopBack interface.
- Since this is RUDP, every connection with the server, we simulate the 3 way handshake. We will explain it once since it repetitive.
- We include 4 pcapng files:

  1) FTP-RUDP (Download).pcapng - For the download part.

  2) FTP-RUDP (Upload Normal).pcapng - For the normal upload part.

  3) FTP-RUDP (Upload Delay).pcapng - For the delay upload part.

  4) FTP-RUDP (Upload Packet Loss).pcapng - For the packet loss upload part.

In here, we started the app and choose the “RUDP” communication. 

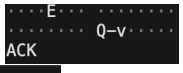
Then, we connected to the “naruto.com” domain and updated the server. 

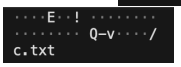

1) - Downloading from the server:

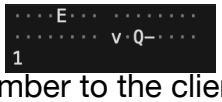
In here, we choose to download “c.txt” from the domain.

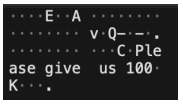
The client first sends that he wants to download (This starts the communication), 

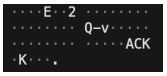
then, the server sends SYN-ACK, 

The client reply with ACK to complete the 3 way handshake. 

After that the client sends the file name to the server, 

the server calculates how many packets the client need to expect, 

Then the server starts sending the packets with their sequence number to the client, 

And then the client sends ACKs for it to the server. 


**Client**

```
**********
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to download.
(+) Received SYN-ACK message.
(+) Sent ACK message.
(+) Sent request to download: c.txt
(+) Received packet # 0
(+) Received the expected packet. Sending ack for it.
(+) Done downloading file.
```

**Server**

```
<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>
(*) Establishing a RUDP connection and preparing to download...
(+) Sent SYN-ACK message.
(+) Received ACK.
(+) Connection established with:  ('127.0.0.1', 20781)
(+) File name to download: c.txt
(*) Reading the file...
(+) Done with reading file.
(+) Sent the number of chunks to the server.
(+) Sent packet # 0
(+) Receive ACK for packet #: 0
(+) Moving window by one.
(+) Sent all packets successfully.
(*) Listening...
```

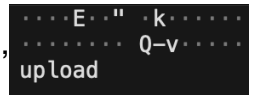
This is the WireShark output:

udp.port == 30413 || tcp.port == 30413

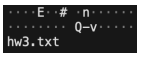
| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 127.0.0.1 | 127.0.0.1 | UDP | 36 | 20781 → 30413 Len=4 |
| 2 | 9.204890 | 127.0.0.1 | 127.0.0.1 | UDP | 42 | 51598 → 30413 Len=10 |
| 3 | 23.240431 | 127.0.0.1 | 127.0.0.1 | UDP | 40 | 20781 → 30413 Len=8 |
| 4 | 23.244542 | 127.0.0.1 | 127.0.0.1 | UDP | 39 | 30413 → 20781 Len=7 |
| 5 | 23.245132 | 127.0.0.1 | 127.0.0.1 | UDP | 35 | 20781 → 30413 Len=3 |
| 6 | 23.245554 | 127.0.0.1 | 127.0.0.1 | UDP | 37 | 20781 → 30413 Len=5 |
| 7 | 23.452447 | 127.0.0.1 | 127.0.0.1 | UDP | 33 | 30413 → 20781 Len=1 |
| 8 | 23.454193 | 127.0.0.1 | 127.0.0.1 | UDP | 69 | 30413 → 20781 Len=37 |
| 9 | 23.455026 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 20781 → 30413 Len=22 |

2) - Uploading normally to the server:

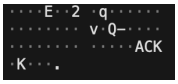
In here, we choose to upload “hw3.txt” from the domain.

The client first sends that he wants to upload (This starts the communication), 


then we have the 3 way handshake which we went over already,

then he sends him the file’s # chunks and name we are interested to upload, 


Then we send the server the file itself, 

and finally the server returns us an ACK for that packet. 


**Client**

```
********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Received SYN-ACK message.
(+) Sent ACK message.
(+) Sent the file's name to the server.
(*) Reading the file...
(+) Done with reading file.
(+) Sent the number of chunks to the server.
(+) Sent packet # 0
(+) Receive ACK for packet #: 0
(+) Moving window by one.
(+) Sent all packets successfully.
```

**Server**

```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a RUDP connection and preparing to upload...
(+) Sent SYN-ACK message.
(+) Received ACK.
(+) Connection established with:  ('127.0.0.1', 20781)
(+) Received packet # 0
(+) Received the expected packet. Sending ack for it.
(+) Done uploading file.
(*) Listening...
```

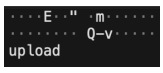
This is the WireShark output:

udp.port == 30413 || tcp.port == 30413

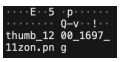
| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 127.0.0.1 | 127.0.0.1 | UDP | 38 | 20781 → 30413 Len=6 |
| 2 | 0.000685 | 127.0.0.1 | 127.0.0.1 | UDP | 39 | 30413 → 20781 Len=7 |
| 3 | 0.001474 | 127.0.0.1 | 127.0.0.1 | UDP | 35 | 20781 → 30413 Len=3 |
| 4 | 2.005082 | 127.0.0.1 | 127.0.0.1 | UDP | 39 | 20781 → 30413 Len=7 |
| 5 | 2.016290 | 127.0.0.1 | 127.0.0.1 | UDP | 33 | 20781 → 30413 Len=1 |
| 6 | 2.017997 | 127.0.0.1 | 127.0.0.1 | UDP | 152 | 20781 → 30413 Len=120 |
| 7 | 2.018703 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |

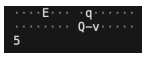
3) - Uploading with delay to the server:

In here, we choose to upload “thumb_1200_1697_11zon.png” which requires 5 chunks in total to be sent to the server so that we can notice the delay.

The client first sends that he wants to upload (This starts the communication), 


then we have the 3 way handshake which we went over already,

then he sends him the file’s # chunks and name we are interested to upload, 




Then we send the server the file itself, meaning the 5 chunks of data.

For every chunk successfully sent we receive ACK from the client.

Since delay mode is on, we can see in the terminal each packet triggers the timeout.

Non the less, in the server’s terminal there’s no affect since we eventually receive the packet in the order we needed.

**Client**

```
**********
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Received SYN-ACK message.
(+) Sent ACK message.
(+) Sent the file's name to the server.
(*) Reading the file...
(+) Done with reading file.
(+) Sent the number of chunks to the server.
(-) Timeout occurred. Need to resend packet # 0
(+) Sent packet # 0
(+) Receive ACK for packet #: 0
(+) Moving window by one.
(-) Timeout occurred. Need to resend packet # 1
(+) Sent packet # 1
(+) Receive ACK for packet #: 1
(+) Moving window by one.
(-) Timeout occurred. Need to resend packet # 2
(+) Sent packet # 2
(+) Receive ACK for packet #: 2
(+) Moving window by one.
(-) Timeout occurred. Need to resend packet # 3
(+) Sent packet # 3
(+) Receive ACK for packet #: 3
(+) Moving window by one.
(-) Timeout occurred. Need to resend packet # 4
(+) Sent packet # 4
(+) Receive ACK for packet #: 4
(+) Moving window by one.
(+) Sent all packets successfully.
```

**Server**

```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a RUDP connection and preparing to upload...
(+) Sent SYN-ACK message.
(+) Received ACK.
(+) Connection established with:  ('127.0.0.1', 20781)
(+) Received packet # 0
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 1
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 2
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 3
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 4
(+) Received the expected packet. Sending ack for it.
(+) Done uploading file.
(*) Listening...
```

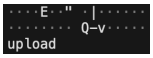
This is the WireShark output (we can see the delay):

udp.port == 30413 || tcp.port == 30413

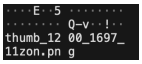
| No. | Time | Source | Destination | Protocol | Length | Info | |
|---|---|---|---|---|---|---|---|
| 1 | 0.000000 | 127.0.0.1 | 127.0.0.1 | UDP | 38 | 20781 → 30413 Len=6 | |
| 2 | 0.000586 | 127.0.0.1 | 127.0.0.1 | UDP | 39 | 30413 → 20781 Len=7 | |
| 3 | 0.001149 | 127.0.0.1 | 127.0.0.1 | UDP | 35 | 20781 → 30413 Len=3 | |
| 4 | 2.016394 | 127.0.0.1 | 127.0.0.1 | UDP | 57 | 20781 → 30413 Len=25 | |
| 5 | 2.017020 | 127.0.0.1 | 127.0.0.1 | UDP | 33 | 20781 → 30413 Len=1 | |
| 6 | 4.030725 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 | Chunk #0 |
| 7 | 4.031300 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 | |
| 8 | 6.032636 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 | Chunk #1 |
| 9 | 6.033237 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 | |
| 10 | 8.048900 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 | Chunk #2 |
| 11 | 8.049633 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 | |
| 12 | 10.052074 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 | Chunk #3 |
| 13 | 10.052778 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 | |
| 14 | 12.055057 | 127.0.0.1 | 127.0.0.1 | UDP | 2848 | 20781 → 30413 Len=2816 | Chunk #4 |
| 15 | 12.055988 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 | |

4) - Uploading with packet loss to the server:

In here, we choose to upload “thumb_1200_1697_11zon.png” which requires 5 chunks in total to be sent to the server so that we can notice the delay.

The client first sends that he wants to upload (This starts the communication), 


then we have the 3 way handshake which we went over already,

then he sends him the file’s # chunks and name we are interested to upload, 


Then we send the server the file itself, meaning the 5 chunks of data.

For every chunk successfully sent we receive ACK from the client.

Since packet loss is on, we can see in the terminal a lot of packets aren’t sent when they should be sent which causes delay in receiving the ACKs and in sending packets.

In the server’s terminal we don’t see an affect because we can’t send packets with higher sequence number before we are sending the expected packet.

This would happen only when we sent packet $x$ and $y$ where $x < y$ but $y$ arrives to the server sooner.

**Client**

```
********************************
(*) Creating the client socket...
(+) Binding was successful.
(+) Notified the server we want to upload.
(+) Received SYN-ACK message.
(+) Sent ACK message.
(+) Sent the file's name to the server.
(*) Reading the file...
(+) Done with reading file.
(+) Sent the number of chunks to the server.
(-) Didn't send packet # 0
(-) Didn't send packet # 0
(-) Didn't send packet # 0
(+) Sent packet # 0
(-) Didn't send packet # 1
(-) Didn't send packet # 1
(+) Sent packet # 1
(-) Didn't send packet # 2
(-) Didn't send packet # 2
(-) Didn't send packet # 2
(-) Didn't send packet # 2
(+) Receive ACK for packet #: 0
(+) Moving window by one.
(+) Sent packet # 2
(-) Didn't send packet # 3
(-) Didn't send packet # 3
(-) Didn't send packet # 3
(+) Receive ACK for packet #: 1
(+) Moving window by one.
(+) Sent packet # 3
(+) Receive ACK for packet #: 2
(+) Moving window by one.
(+) Sent packet # 4
(+) Receive ACK for packet #: 3
(+) Moving window by one.
(+) Receive ACK for packet #: 4
(+) Moving window by one.
(+) Sent all packets successfully.
```

**Server**

```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
(*) Establishing a RUDP connection and preparing to upload...
(+) Sent SYN-ACK message.
(+) Received ACK.
(+) Connection established with:  ('127.0.0.1', 20781)
(+) Received packet # 0
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 1
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 2
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 3
(+) Received the expected packet. Sending ack for it.
(+) Received packet # 4
(+) Received the expected packet. Sending ack for it.
(+) Done uploading file.
(*) Listening...
```

This is the WireShark output (we can see the packet loss by the delay in the ACKs):

udp.port == 30413 || tcp.port == 30413

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 127.0.0.1 | 127.0.0.1 | UDP | 38 | 20781 → 30413 Len=6 |
| 2 | 0.000648 | 127.0.0.1 | 127.0.0.1 | UDP | 39 | 30413 → 20781 Len=7 |
| 3 | 0.001167 | 127.0.0.1 | 127.0.0.1 | UDP | 35 | 20781 → 30413 Len=3 |
| 4 | 2.006895 | 127.0.0.1 | 127.0.0.1 | UDP | 57 | 20781 → 30413 Len=25 |
| 5 | 2.007619 | 127.0.0.1 | 127.0.0.1 | UDP | 33 | 20781 → 30413 Len=1 |
| 6 | 2.009011 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 |
| 7 | 2.009460 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 |
| 8 | 2.009688 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |
| 9 | 2.010066 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 |
| 10 | 2.010158 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |
| 11 | 2.010536 | 127.0.0.1 | 127.0.0.1 | UDP | 4150 | 20781 → 30413 Len=4118 |
| 12 | 2.010609 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |
| 13 | 2.010928 | 127.0.0.1 | 127.0.0.1 | UDP | 2848 | 20781 → 30413 Len=2816 |
| 14 | 2.011113 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |
| 15 | 2.011481 | 127.0.0.1 | 127.0.0.1 | UDP | 54 | 30413 → 20781 Len=22 |

# Bibliography

- This is an hyper-linked supported bibliography.

---


- *Unknown. Summary in Networking .Docx, 2022.*
- *Unknown. Course presentation, principles of RUDP, starting page 52.*
- *soubhikmitra98. Reliable User Datagram Protocol (RUDP). .geeksforgeeks, 2022.*
- *David Fagbuyiro. File Handling in Python – How to Create, Read, and Write to a File. freecodecamp, 2022.*
- *Unknown. Scapy p.09 Scapy and DNS. thepacketgeek, 2019.*
- *Antonello Zanini. Will Google's QUIC Protocol Replace TCP? levelup, 2021.*
- *A. Esterhuizen and A.E. Krzesinski. TCP Congestion Control Comparison. lafibre, unknown.*
- *Katie Terrell Hanna. Network Address Translation (NAT). TechTarget, 2021.*