



**Trinity College Dublin**  
Coláiste na Tríonóide, Baile Átha Cliath  
The University of Dublin

---

# **CSU33031 Computer Networks**

## **Assignment #1: File Retrieval Protocol**

---

**Abdelaziz Abushark, Std# 20332134**

October 27th, 2022

### **Contents**

1	Introduction .....	2
2	Theory of Topic.....	2
	Overall Topology .....	2
2.1	IP, Ports and Socket.....	3
2.2	UDP and Datagrams .....	3
2.3	Header & Sequence Number .....	4
2.4	Client .....	4
2.5	Server (Ingress).....	4
2.6	Worker .....	4
2.7	Multiplexing .....	4
2.8	Encryption .....	5
2.9	Forward Error Correction.....	5
2.10	Communication and Packet Description .....	6
3	Implementation.....	7
3.1	Client.py .....	8
3.2	Server.py.....	9
3.3	Worker.py, Worker2.py, Worker3.py .....	11
3.4	multiplexEncrypt.py .....	13
3.5	User Interaction.....	14
3.5.1	Terminal Usage .....	15
3.6	Packet Encoding .....	17
3.7	Docker and Network capture.....	17
4	Discussion .....	17
4.1	The reason behind using Docker .....	17
4.2	Why I used these Functionalities .....	18
4.3	Wireshark.....	19

4.4 Other Design Decisions .....	21
4.5 Features .....	21
5 Summary .....	21
6 Reflection .....	22
7 References .....	22

## 1 Introduction

We were given a problem to study protocol development and the data that is retained in a header to support a protocol's functionality. The problem focuses on the mechanism of retrieving files from ingress(server) based on UDP datagrams. It works by client sending a request to a server which called ingress which this server will distribute the request to available workers. In which the worker will retrieve the file and send it to the server which then will forward it to the client and client will then request another file from the server and server will pick a different worker to return this file and so on.

The approach that I have taken the address this problem was by studying more about the topic and deciding which programming language to use. I realized using Python was easier for me as built-in libraries helped me in the development. I addressed this problem by planning 3 development stages which helped me a lot in delivering the tasks on time. I first decided to develop both client and server and made sure the client and server were working on terminal and that client was sending request and server was receiving them. Secondly, I designed the protocols and topologies that I am going to use to develop this project and Finally I developed the 3 workers and connected them to the server and implemented the functionalities (protocols & topologies)

The idea behind this assignment will first be described in this report. The implementation's specifics will next be discussed, followed by a look at the choices that were made when building this protocol. Finally, a summary and reflection on the entire task will be given.

## 2 Theory of Topic

### Overall Topology

This assignment took a significant amount of time because I had to research the theory to implement my approach. I'll lay you the idea behind my answer to prevent any misunderstandings.

The system consists of 3 main components. Client, Server, and Workers (1,2and3). Client sends a request to a server which called ingress which then distributes it to the available workers. The available worker will then retrieve the file and send it back to the ingress and then will forward it back to the client. Figure 1 represents the interactions between the 5 components when the program starts running.

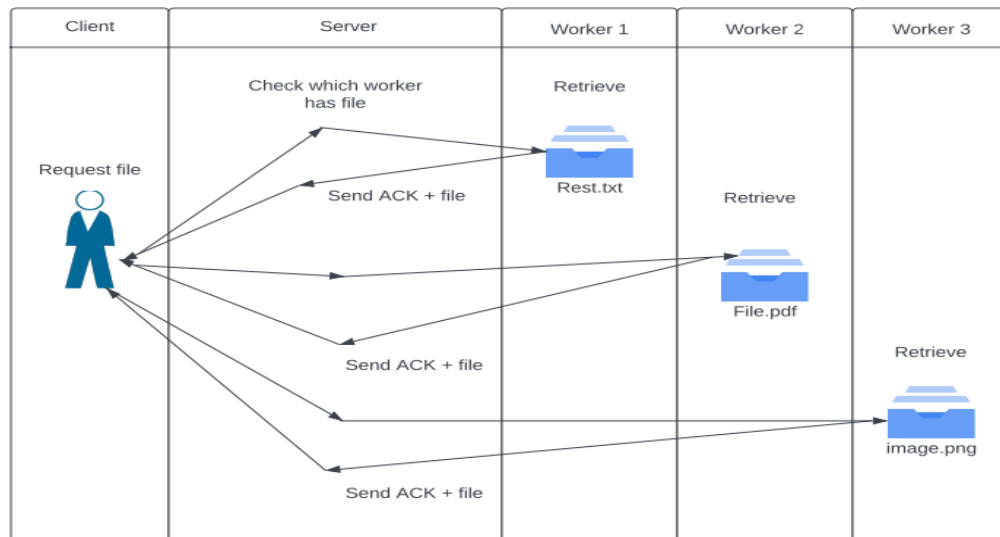


Figure 1: Activity Diagram of explaining the Topology

## 2.1 IP, Ports and Socket

The network layer of the Open Systems Interconnection (OSI) paradigm is what we're talking to when we talk about the Internet Protocol (IP). IP is used to transport packets, which are identified by IP addresses, from a source host to a destination host. Whether IPv4 or IPv6 is utilized, IP addresses are a distinct and universal sequence of 32 or 128 bits. An endpoint for transmitting and receiving packets over a network is a socket. A communication endpoint is symbolized by a port. A port number is used to identify a process or service, whereas an IP address is used to identify a host in a network.

## 2.2 UDP and Datagrams

UDP, which stands for *User Datagram Protocol*, is a method used to transfer large files across the Internet. Look at figure 2. Due to its unreliability, it is connectionless and mostly used for low-latency and loss-tolerating connections. Because there is no built-in flow or error control, it is better suited for applications that can offer their own. Datagrams are the term used to describe packets transmitted through UDP. A header and a data segment make up a UDP datagram. Look at figure 3. The datagram length, source and destination port numbers, a checksum field, and the header's 8-byte size make it comparatively compact.

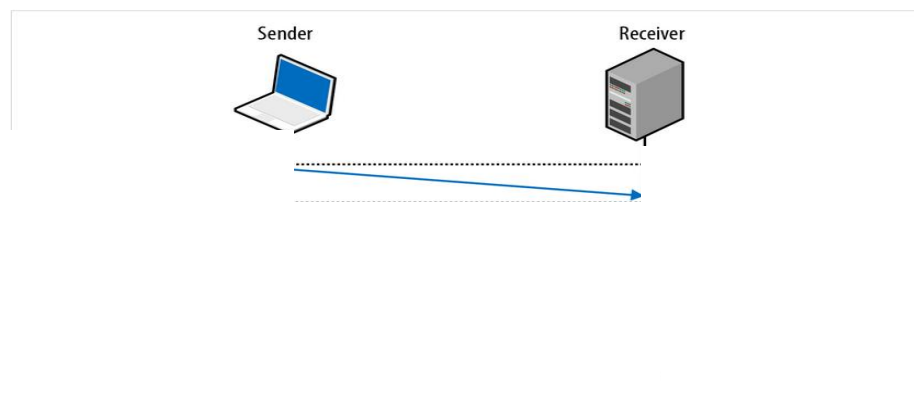


Figure 2: shows User Datagram protocol

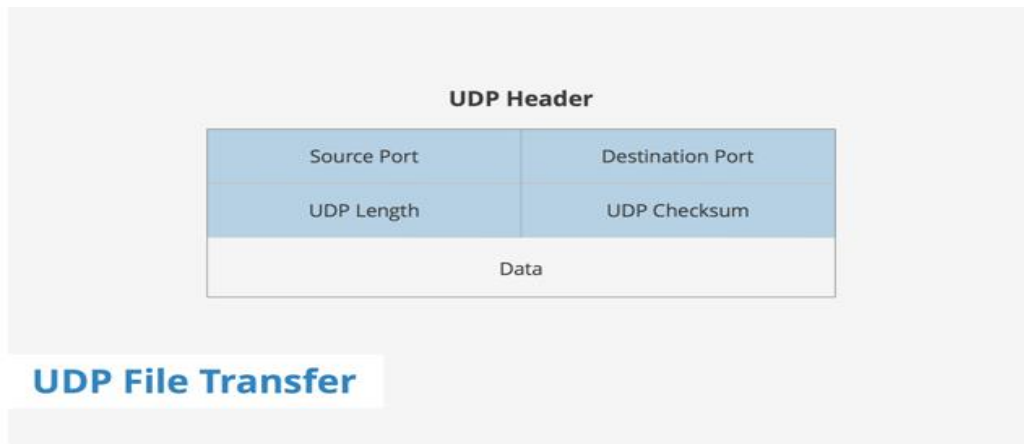


Figure 3: shows User Datagram protocol header

## 2.3 Header & Sequence Number

A piece of data called the header is included at the beginning of each packet. Numerous pieces of metadata, including the type of packet it is attached to, the sequence number, the subject number, etc., are contained in the header. We can create the data in the correct order regardless of packet loss or reordering since the sequence number lets us know the order in which packets have been sent. In this UDP header, I implemented the section for sequence numbers that holds the order of the file chunks. But I didn't alter the supplied default header in any way (i.e., Source port, destination port, and UDP length). With this additional component, we can monitor the sequence of the packets and helped us lessen packet loss in addition to receiving them.

## 2.4 Client

A client class can request any type of files (txt, png, pdf ... etc) from the server (ingress).

## 2.5 Server (Ingress)

A server class will then select a worker and forward the request to the worker and then after the retrieval of the file from the workers server will forward it to the client.

## 2.6 Worker

A worker (workers) class will receive a request from the server about specific file, the worker will retrieve the file and send it to the server.

## 2.7 Multiplexing

I used Multiplexing as it allowed me to transfer multiple complex signals of data into one go in other words when the client requests the files from the server and then the server pick the worker to retrieve the file, when file has been retrieved by the worker then it was sent into the server which is the medium in which during this procedure, multiple signals will be consolidating into a single complex signal that is transported over a common medium which is the ingress then reaching the client . Figure 4 shows the procedure visually. I talked more about why I chose multiplexing specifically in my implementation and discussion sections.



Figure 4: shows how multiplexing works with file retrieval protocol

## 2.8 Encryption

I used AES Encryption for more security over file transfer and for quicker transfer of files between the components of the file retrieval protocol. I used Cipher text so the files will show encryption code when they are retrieved from worker to client via medium which is the server. I then used Cipher text feedback which allows my program to block cipher. As I gave the user the choice if they want to view the name of the file after encryption which is decryption or remain with cipher and keep it encrypted for safer options. Figure 5 shows representation of encryption of plain text and cipher text. I went into more details about the reasons behind using encryption and the difference when capturing traffic in my implementation and discussion sections.

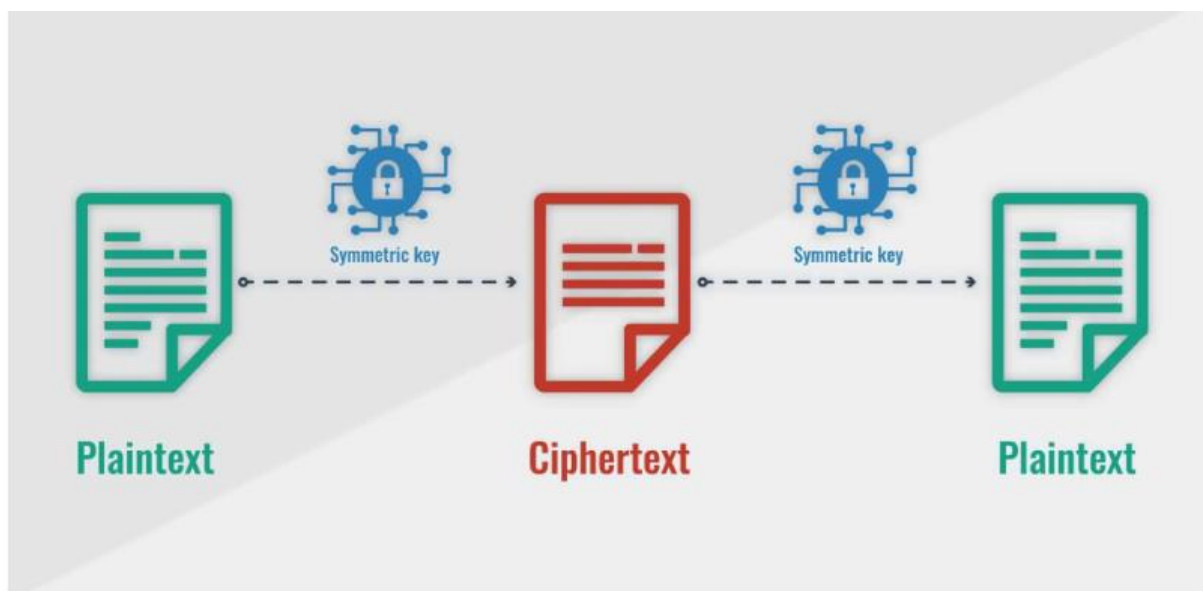


Figure 5: shows the 2 options that the user is allowed to choose between “Cyphertext or Plaintext”

## 2.9 Forward Error Correction

I will talk in brief about why here, but I went into details about this functionality in my implementation and discussion sections as I raised some questions and thoughts about it. I consider Forward Error Correction with Mechanism of FEC the most important functionality in my program because it is like the backbone of the program as it helps my program to detect any irregularity for example when my program executes, it starts counting number of packets coming from the server, calculate the number of packets that are missing, and if that calculation returns with the missing packets, we request it from the ingress. I used figures on discussion section to help the reader understand how this procedure works.

## 2.10 Communication and Packet Description

Communication between 3 components (client, server, and workers) was done by packet transfer as each packet that is transferred between components contains data in its header that aids in the identifying of the requested item, enables the efficient transfer of files, and informs the recipient of the sender's IP address. I implemented Stop and Wait ARQ protocol in which each frame has a sequence number of either 1 or 0 and when the client requests a file from the ingress, it will send a frame to the ingress and then will wait for the acknowledgment packet before sending the next request to the ingress. I also set a timeout in which if the ingress did not receive the frame within a period of time, client will have to resend the request again in which will be labelled with a sequence number (0,1) depends on the situation of the packet as if the last packet had a sequence number of 0, the next one will be 1 and so on unless it was interrupted and it will repeat the same sequence number as the first packet with same sequence number has not arrived yet. After getting the acknowledgment packet, the ingress then will then select a worker and forward the request to the worker in which (Stop and Wait ARQ) will also happen here and after successfully receiving the request, worker will retrieve the file back to the ingress in which then will be sent back to client by the ingress. Figure 6 visualizes how Stop and Wait ARQ protocol works. Figure 7 is a more detailed visualization of how Stop and Wait works with sequence numbers and handles errors.

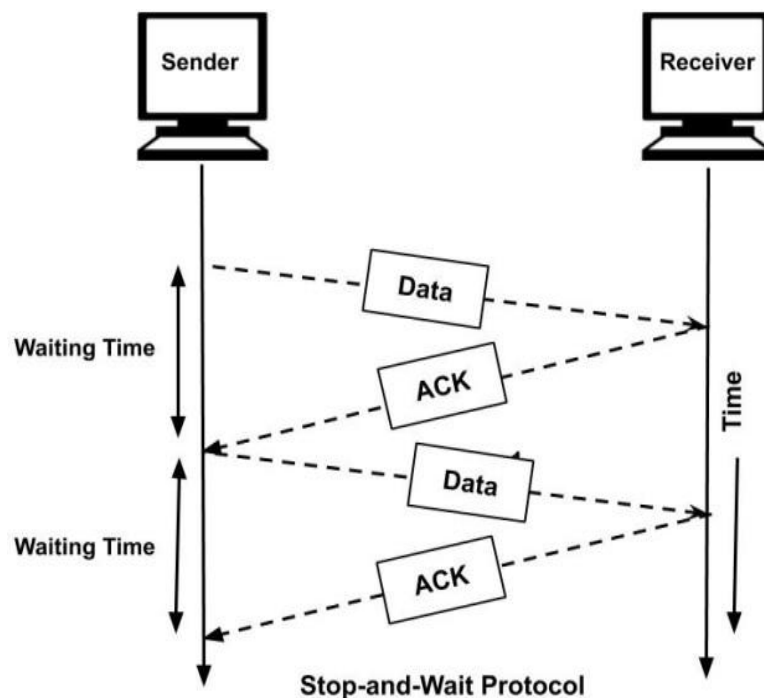


Figure 6: shows the communication between the components with no packet loss

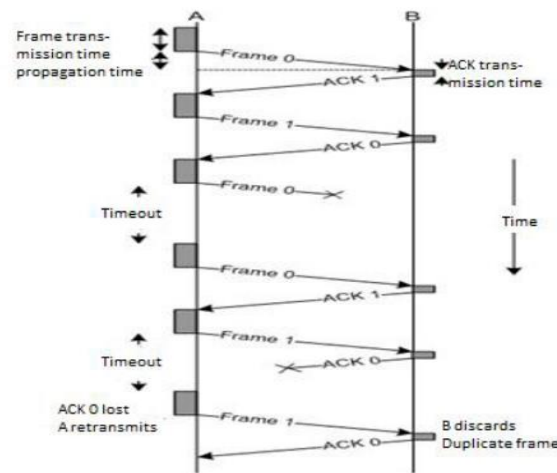


Figure 7: shows the communication between the components with packet loss which explained when the sender does not receive the ACK before the set timeout, it will resend the same packet with the same frame number as well as it shows when there is a lost ACK, it will resend the same ACL with the same packet

I have developed a wide variety of packet types, which are used while sending data between client, server, and workers (networks nodes). The next sequence number required from the sender is contained in the acknowledgement packets. The worker's availability to work is contained in the availability packet. It also includes the worker's identification number, which is used to locate the individual. To stop work in progress, use the work cancellation packet. The Client Info Packet provides all the information about the Client, including IP address, name, ID, and port number as well as I have done the same for Workers Info Packet and Server Info Packet. The Workers/Ingress Return Packet includes the worker's/server's identity, worker/server ID and return of the work.



Figure 8: shows the basic structure of the packet without the 3<sup>rd</sup> element, which is the trailer, in which the header of the packet contains the source address, a destination address, protocol, and packet number and payload contains data that a user or device wants to send

### 3 Implementation

I will be talking about the protocol's implementation in this section in details. Python was used to implement the protocol while Docker, which I talked in detail about it in Discussion section, was used to stimulate the components and then I will explain Python classes that I have used.

The server and client had to be created by myself as I was using Python. This was not too difficult because Python already has a socket module that was used to make this work. I initially created a simple client and server to learn how to utilize the socket module to enable program communication. There could only be one client and one server in this initial implementation. After completing this simple design solution, I made the decision to focus on enabling several workers to connect with the server(ingress). Threads were used to do this. The server code was modified such that it now waits for connections and, when one is established, forwards the connection to a procedure that is running in a new thread in addition it was modified to choose which worker to retrieve the file as well as the client code was modified to ask for

direct requests from the server and workers class were added as a directory as it was a simpler way for retrieving file and sending them to the server. The worker code had to be altered to only allocate a port to the ingress that was not already in use by another worker. Although it mostly functioned, if a client closed, the server and workers would keep the thread running until the entire application was closed. The problem was resolved by utilizing try and except statements; now, if the server or client experienced a connection failure, the thread would be stopped rather than the program crashing or continuing forever.

### 3.1 Client.py

The Client class was given to us on Blackboard for the Docker Walkthrough, which I implemented more functionalities and topologies to suit my protocol and my implementation. I used python built-in libraries to help me with this Assignment.

I used a socket module in which will help me connect client to the server and make this worker and that is one of the main reasons I decided to use Python for this assignment it is easier and has built-in functions that will make my life easier. The socket module was imported, a socket object (UDP socket object) was formed, and a bond with a local host and port number was established on code shown below. We are bound to a particular address (172.20.0.0), so senders can target that address to transmit information to us. The argument of the socket. socket () object is what converts the socket object to a UDP protocol socket. Address family is the first argument (socket.AF\_INET). It serves to specify the categories of addresses with which our socket can communicate (in this case, Internet Protocol v4 addresses). The socket just needs the second option (socket.SOCK\_DGRAM) to be a UDP protocol socket. As this assignment was to use UDP protocol so SOCK\_DGRAM for it but if I was to use TCP protocol, I would have used SOCK\_STREAM.

```
def client(filename):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
    ADDR = (HOST, PORT)
    client_buffer = 512
    key = "*asdasdasdhssss*"

    # sending the filename to server
    protocol = AESCipher(key)
    file_name = protocol.encrypt(filename.encode())
    packets = file_name

    source = PORT
    destination = PORT
```

Listing 1: client.py

In the basic approach above, we are merely transmitting the data. We did not calculate the checksum since there are no UDP headers. Our data is thus in risk. In order to protect my data, I decided to use another library called Zlib which is shown in the listing below, which is used for calculating the packets (checksum) in other words it is a checksum calculator as well I used while loop to calculate the number of packets and to print them out on the command line.

```
Import zlib
def calculate_status(packets):
    return zlib.crc32(packets)
```



```

while packets:
    packets = protocol.decrypt(packets)
    print(
        f" [SEND TRAFFIC]:No #{udp_header[4]} |{udp_header[3]}| Status: "
        f"{ new_header[3] == calculate_status(packets)}")
    if new_header[3] == calculate_status(packets):
        pass
    else:
        file_chunks.append(packets)

```

Listing 2: client.py (checksum calculator)

```

if (forward_error) !=0:
    print("[CLIENT]: Requesting error packets from the server!")
    packets = str(forward_error).encode()
    header_info = len(packets)
    status = calculate_status(packets)
    new_header = struct.pack(
        "!IIII", source, destination, header_info, status, 0)
    error_id_packet = udp_header + packets
    sock.sendto(error_id_packet, ADDR)

    print("[CLIENT]: Receiving errors!")
    header, ADDR = stream_sock.recvfrom(client_buffer)
else :
    print("[CLIENT]: No error packets found!")
    while packets:
        packets = protocol.decrypt(packets)
        print(
            f"#{new_header[4]}->{new_header [2] == calculate_status(packets)}")
        sock.settimeout(2)
        header, ADDR = stream_sock.recvfrom (client_buffer)

```

Listing 3: client.py – (forward error correction protocol)

In the example above, I used forward error correction protocol as you can see it was implemented using if statements to check if they are any error packets found as it works by counting the number of packets coming from the server, calculate the number of packets that are missing, and if that calculation returns with the missing packets, we request it from the ingress. To see the capability of forward error correction, the server must send them back again. In addition, I printed out the information of every packet as well as the statuses which was implemented in the above code. I will explain it more in the user interaction section.

### 3.2 Server.py

The Server class was given to us on Blackboard for the Docker Walkthrough, which I implemented more functionalities and topologies to suit my protocol and my implementation. I used python built-in libraries to help me with this Assignment.

In the code below, as we have the checksum calculated from the code that was implemented in both client and server, we're attempting to generate a UDP header with the four necessary fields. The UDP header is being created using the struct module. We inform the UDP header of the type of data it contains and the number of fields we plan to use using the first parameter of the struct.pack() function. "IIII" I stands for an unsigned integer with a standardized size of 5, and "I" itself denotes that we will be packing 4 fields. After that I just combined the UDP header with encoded data.

```
analyzer = argparse.ArgumentParser()
analyzer.add_argument('--port', '-p', help="Port no")
analyzer.add_argument('--ipAddress', '-i', help="Ip no")
i = analyzer.parse_args()
PORT = int(i.port)
HOST = i.ipAddress
data_length = len(chunk)
status_checksum = calculate_status(chunk)
new_header = struct.pack( "IIII", source, destination, data_length, status_checksum, index)
packet = udp_header + data

sock.sendto(packet, ADDR)

print("[SERVER]: All packets were sent! \n\n")
```

Listing 4: Server.py (UDP header data)

In the code below you can see my implementation to the workers directory in which connects ingress to workers and vice versa as it shows in the command file when program executes that server chooses which worker to retrieve the file from. I implemented that using for loop and I will show the command line in the User Interaction section.

```
def server():
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
    ADDR = (HOST, PORT)
    server_buffer = 512
    key = "*asdasdasdhssss*"
    source = PORT
    destination = PORT
    sock.bind(ADDR)
    aes = AESCipher(key)
    full_packet, ADDR = sock.recvfrom(server_buffer)
    new_header, info = full_packet[:header_size], full_packet[header_size:]
    file_name = info
    file_name = aes.decrypt(file_name).decode()
    print(f"[SERVER]: Search for {file_name} Request in Available workers")
    directory = ""
    for i in workers:
        if file_name in workers[i]:
            print(f"[SERVER]: File located in: {i}")
            real_path = f"./workers/{i}/{file_name}"
```

Listing 5: Server.py (workers directory)

### 3.3 Worker.py, Worker2.py, Worker3.py

These 3 classes have the same implementation as when the server picks which worker to retrieve the file to the client, the selected worker will then start searching for the file using glob library and retrieve it to server which then server sends it to client and the codes below was implemented using if statements to distinguish between the workers

```
import glob
transfer = []
workers = []
for path in glob.glob(f):
    if len(path.split("-")) != 3:
        continue
    workers[m].append(f)
try:
    # send to a multicast group
    if option == 1:
        print(f"[SERVER]: Forwarding, {filename} Request To Worker: 1\n")
        multicast_group = ('172.20.0.4', 9999)
        sock.sendto(filename.encode(FORMAT), multicast_group)
        file = open("./code/worker1/" + filename, 'wb')
        file = open("./code/client/" + filename, 'wb')

    elif option == 2:
        multicast_group = ('172.20.0.5', 9999)
        sock.sendto(filename.encode(FORMAT), multicast_group)
        file = open("./code/worker2/" + filename, 'wb')
        file = open("./code/client/" + filename, 'wb')

    elif option == 3:
        multicast_group = ('172.20.0.6', 9999)
        sock.sendto(filename.encode(FORMAT), multicast_group)
        file = open("./code/worker3/" + filename, 'wb')
        file = open("./code/client/" + filename, 'wb')

    else:

        print(f"[ERROR]: This worker Ip Address is not Authorised!!!")
        print(f"[ERROR]: Enter a valid ip !!!\n")
```

Listing 6: Server.py which shows the 3 workers that the server will request the files from

```

def worker2():
    address = (HOST, PORT)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(server_address)
    group = socket.inet_aton(multicast_group)
    mreq = struct.pack('4sL', group, socket.INADDR_ANY)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
    while x:
        if not sock.sendto(x, address):
            continue
        print(f"[WORKER2]: Sending Traffic To The [SERVER]....")
        x = file.read(SIZE)
        sock.sendto('ack'.encode(), address)
        print(f"[RESPONSE]: Sending Acknowledgement To", address)

        print(f"[STATUS]: SUCCESS")

    exit()

def worker3():
    address = (HOST, PORT)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(server_address)
    group = socket.inet_aton(multicast_group)
    mreq = struct.pack('4sL', group, socket.INADDR_ANY)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
    while x:
        if not sock.sendto(x, address):
            continue
        print(f"[WORKER3]: Sending Traffic To The [SERVER]....")
        x = file.read(SIZE)
        sock.sendto('ack'.encode(), address)
        print(f"[RESPONSE]: Sending Acknowledgement To", address)

        print(f"[STATUS]: SUCCESS")

    exit()

def worker1():
    address = (HOST, PORT)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(server_address)
    group = socket.inet_aton(multicast_group)
    mreq = struct.pack('4sL', group, socket.INADDR_ANY)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

    while True:
        print('-----')
        print(
            'UDP PROTOCOL FOR SENDING / RETRIEVING\n  FILES FROM CLIENTS AND SERVER')
        print('-----')
        print(f"[WORKER1]: Waiting to Receive Message\n")
        x, address = sock.recvfrom(SIZE)
        filename = x.decode().strip()
        print(f"[WORKER1]: Uploading {filename} .... ")
        file = open(filename, "rb")
        x = file.read(SIZE)

```

Listing 7: worker.py, worker2.py and worker3.py

As you can see above. They have the same implementation as I explained before and their main job is to wait for a request from the server and to retrieve the file to the server after matching it using glob library.

### 3.4 multiplexEncrypt.py

Lastly this class in which I implemented 2 more functionalities in addition to forward error correction. I added AES encryption and Multiplexing. I found some resources and code online that I will be referencing it as I asked the lecturer if I could use online material and he said as it works with your program, you should only reference it.

```
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
import selectors
import socket

class AESCipher():
    def __init__(self, key):
        self.block_size = AES.block_size
        self.key = hashlib.sha256(key.encode()).digest()

    def encrypt(self, aes_cipher):
        aes_feedback = Random.new().read(self.block_size)
        cipher = AES.new(self.key, AES.MODE_CFB, aes_feedback)
        return aes_feedback + cipher.encrypt(aes_cipher)

    def decrypt(self, encrypted_text):
        aes_feedback = encrypted_text[:self.block_size]
        aes = AES.new(self.key, AES.MODE_CFB, aes_feedback)
        return aes.decrypt(encrypted_text[self.block_size:])
    def cipher_block(self, plain_text):
        feedback = self.block_size - len(plain_text) % self.block_size
        cipher_block = chr(feedback)
        padding_str = feedback * cipher_block
        new_plain_text = plain_text + padding_str
        return new_plain_text

sel = selectors.DefaultSelector()

def accept(socket):
    multi, address = socket.accept()
    print('accepted', multi, 'from', address)
    multi.setblocking(False)
    sel.register(multi, selectors.EVENT_READ, transfer())

def transfer(multi):
    info = multi.recv(512)
    assert isinstance(info, object)
    if not info:
        return
    print('echoing', repr(info), 'to', multi)
    multi.send(info)

sock = socket.socket()
sock.listen(100)
```

Listing 8: multiplexEncrypt

In the code above you can see via the implemented functions for encryption , the first function in which it encrypts the files that is being transferred for more speed and safety and then the second function which I worked on after the 2<sup>nd</sup> video in which decrypts the files sent and 3<sup>rd</sup> function (cipher\_block) which shows the name of the file in the traffic capture in Wireshark as when I submitted the 2<sup>nd</sup> video the files were encrypted in Wireshark and I mentioned that I will work on decrypting the files in which I will show the difference in User Interaction section. Then we go into the multiplexing as the reason I used multiplexing because it sends multiple of signals of information over a communication link at the same time in the form if a single signal and you can it is implemented in functions “accept and transfer). I implemented Encryption using a Cryptodome library and I implemented multiplexing using selectors and socket libraries.

### 3.5 User Interaction

I designed my program with 5 containers, one for client, one for server and three for workers as you can see in figure 9. My program works by starting the server then client then workers. Once the user starts them, the program will be running, and the server will give you the names of the files that are in the workers as seen in figure 10. The server is waiting for the client to send a request for which file client wants. Then the server will be asking the client how many files you want to transfer as if 2 files them 2 workers will be working and same goes to 3 and 1 and then server will ask the client about the name of the file which is shown in figure 11. After stating the name , specific worker that has that file name will be retrieving the file and sending it to server as shown in Figure 12 which then will be sent to client and message from server saying that no errors received as part of forward error correction and that the file send successfully to the client in which client will show a message saying no error packets found which is forward error correction functionality that I implemented and file name downloaded which will be shown in Figure 13. During sending the files, as the file is being sent in chunks, it will show me the checksum for every chunk in both server and client terminals as shown in figure 14. I will talk about the traffic in discussion as I used security “encryption “functionality in which I noticed some differences which will be talked about in discussion section.

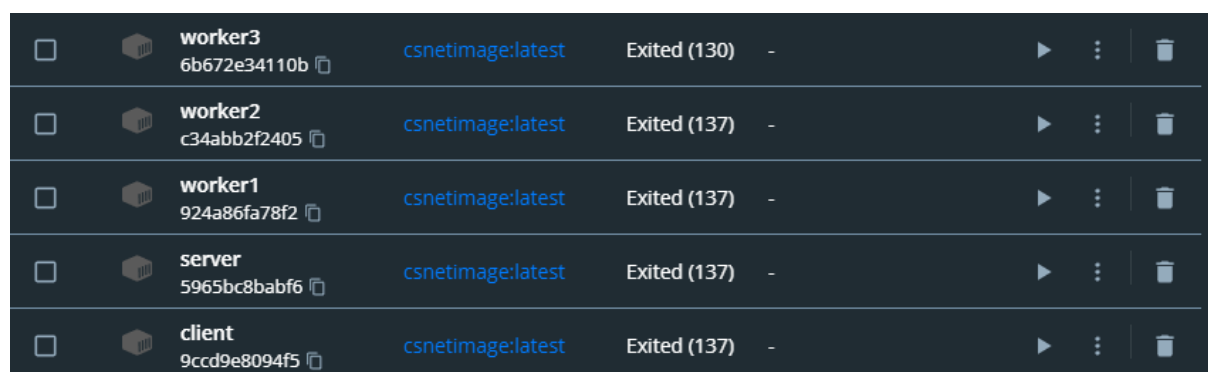
























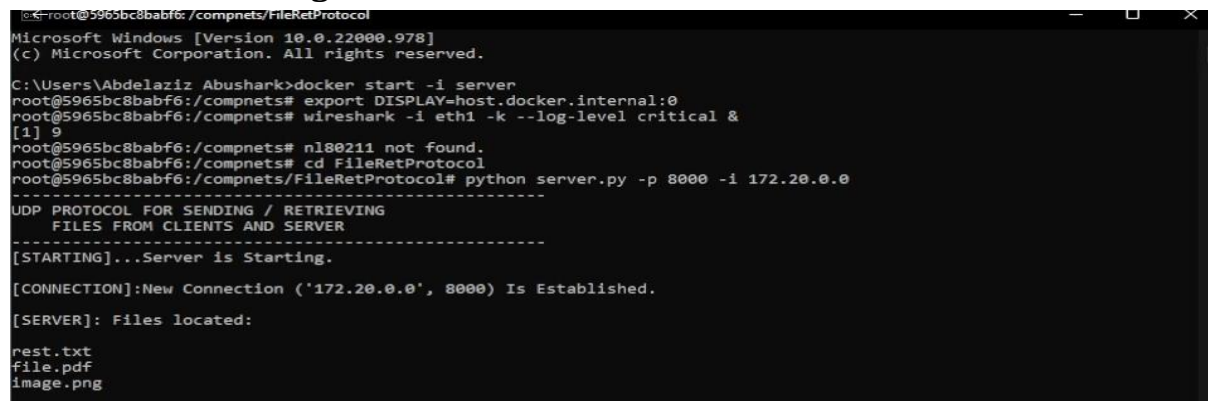
<input type="checkbox"/>	 <b>worker3</b> 6b672e34110b 	<a href="#">csnetimage:latest</a>	Exited (130)	-			
<input type="checkbox"/>	 <b>worker2</b> c34abb2f2405 	<a href="#">csnetimage:latest</a>	Exited (137)	-			
<input type="checkbox"/>	 <b>worker1</b> 924a86fa78f2 	<a href="#">csnetimage:latest</a>	Exited (137)	-			
<input type="checkbox"/>	 <b>server</b> 5965bc8babf6 	<a href="#">csnetimage:latest</a>	Exited (137)	-			
<input type="checkbox"/>	 <b>client</b> 9ccd9e8094f5 	<a href="#">csnetimage:latest</a>	Exited (137)	-			

Figure 9: shows the number of containers I have

### 3.5.1 Terminal Usage



```

root@5965bc8babf6:/compnets/FileRetProtocol
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

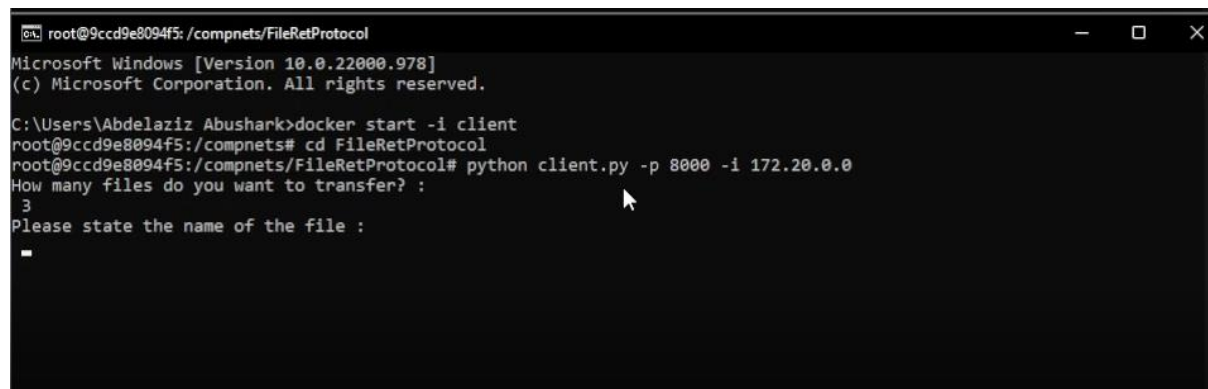
C:\Users\Abdelaziz Abushark>docker start -i server
root@5965bc8babf6:/compnets# export DISPLAY=host.docker.internal:0
root@5965bc8babf6:/compnets# wireshark -i eth1 -k --log-level critical &
[1] 9
root@5965bc8babf6:/compnets# nl80211 not found.
root@5965bc8babf6:/compnets# cd FileRetProtocol
root@5965bc8babf6:/compnets/FileRetProtocol# python server.py -p 8000 -i 172.20.0.0
-----
UDP PROTOCOL FOR SENDING / RETRIEVING
FILES FROM CLIENTS AND SERVER
-----
[STARTING]...Server is Starting.

[CONNECTION]:New Connection ('172.20.0.0', 8000) Is Established.

[SERVER]: Files located:
rest.txt
file.pdf
image.png

```

Figure 10: shows the server class running on terminal

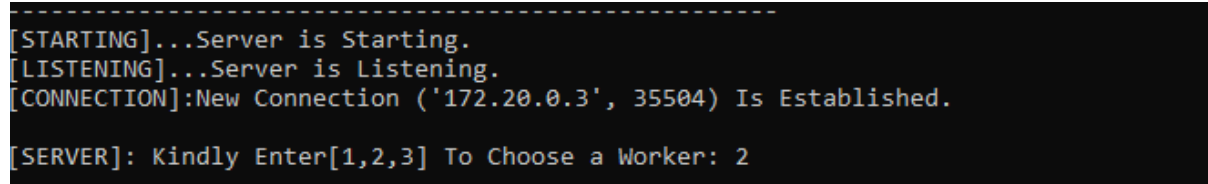


```

root@9ccd9e8094f5:/compnets/FileRetProtocol
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Abdelaziz Abushark>docker start -i client
root@9ccd9e8094f5:/compnets# cd FileRetProtocol
root@9ccd9e8094f5:/compnets/FileRetProtocol# python client.py -p 8000 -i 172.20.0.0
How many files do you want to transfer? :
3
Please state the name of the file :
-

```



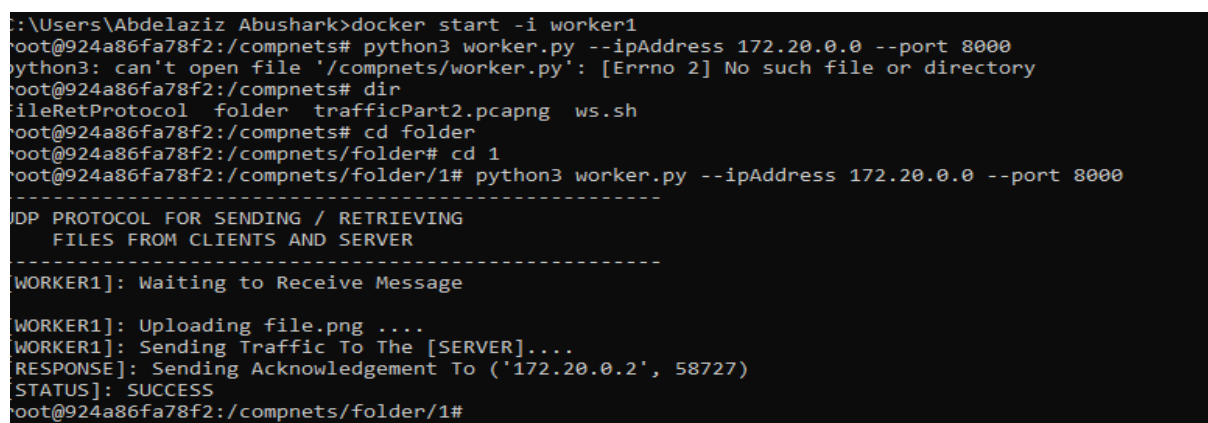
```

-----
[STARTING]...Server is Starting.
[LISTENING]...Server is Listening.
[CONNECTION]:New Connection ('172.20.0.3', 35504) Is Established.

[SERVER]: Kindly Enter[1,2,3] To Choose a Worker: 2

```

Figure 11: shows how the client is requesting a file from the server and the choice of the worker



```

C:\Users\Abdelaziz Abushark>docker start -i worker1
root@924a86fa78f2:/compnets# python3 worker.py --ipAddress 172.20.0.0 --port 8000
python3: can't open file '/compnets/worker.py': [Errno 2] No such file or directory
root@924a86fa78f2:/compnets# dir
fileRetProtocol folder trafficPart2.pcapng ws.sh
root@924a86fa78f2:/compnets# cd folder
root@924a86fa78f2:/compnets/folder# cd 1
root@924a86fa78f2:/compnets/folder/1# python3 worker.py --ipAddress 172.20.0.0 --port 8000
-----
UDP PROTOCOL FOR SENDING / RETRIEVING
FILES FROM CLIENTS AND SERVER
-----
[WORKER1]: Waiting to Receive Message

[WORKER1]: Uploading file.png ....
[WORKER1]: Sending Traffic To The [SERVER]....
[RESPONSE]: Sending Acknowledgement To ('172.20.0.2', 58727)
[STATUS]: SUCCESS
root@924a86fa78f2:/compnets/folder/1#

```

Figure 12: shows how the worker completes the file retrieval process and send it back to the server

```
[SERVER]: No errors received!
[STATUS]: Ok!
[SERVER]: rest.txt send successfully!
```

```
[CLIENT]: No error packets found!
[STATUS]: Ok!
[CLIENT]: rest.txt downloaded!
```

Figure 13: shows the messages from client and server after the retrieval of the file from workers

```
[SEND TRAFFIC]:No #33 |143318519| Status: True
[SEND TRAFFIC]:No #34 |4100700612| Status: True
[SEND TRAFFIC]:No #35 |4026638517| Status: True
[SEND TRAFFIC]:No #36 |995568963| Status: True
[SEND TRAFFIC]:No #37 |3300847930| Status: True
[SEND TRAFFIC]:No #38 |4004766856| Status: True
[SEND TRAFFIC]:No #39 |1846754570| Status: True
[SEND TRAFFIC]:No #40 |613398951| Status: True
[SEND TRAFFIC]:No #41 |1096209765| Status: True
[SEND TRAFFIC]:No #42 |3582186426| Status: True
[SEND TRAFFIC]:No #43 |1073150056| Status: True
[SEND TRAFFIC]:No #44 |3409136503| Status: True
[SEND TRAFFIC]:No #45 |3785925475| Status: True
[SEND TRAFFIC]:No #46 |4218482432| Status: True
[SEND TRAFFIC]:No #47 |325949960| Status: True
[SEND TRAFFIC]:No #48 |435677946| Status: True
```

```
[CLIENT]: No error packets found!
[STATUS]: Ok!
[CLIENT]: rest.txt downloaded!
```

```
[SEND TRAFFIC]:No. #34, Checksum: 4100700612
[SEND TRAFFIC]:No. #35, Checksum: 4026638517
[SEND TRAFFIC]:No. #36, Checksum: 995568963
[SEND TRAFFIC]:No. #37, Checksum: 3300847930
[SEND TRAFFIC]:No. #38, Checksum: 4004766856
[SEND TRAFFIC]:No. #39, Checksum: 1846754570
[SEND TRAFFIC]:No. #40, Checksum: 613398951
[SEND TRAFFIC]:No. #41, Checksum: 1096209765
[SEND TRAFFIC]:No. #42, Checksum: 3582186426
[SEND TRAFFIC]:No. #43, Checksum: 1073150056
[SEND TRAFFIC]:No. #44, Checksum: 3409136503
[SEND TRAFFIC]:No. #45, Checksum: 3785925475
[SEND TRAFFIC]:No. #46, Checksum: 4218482432
[SEND TRAFFIC]:No. #47, Checksum: 325949960
[SEND TRAFFIC]:No. #48, Checksum: 435677946
[SERVER]: No errors received!
[STATUS]: Ok!
[SERVER]: rest.txt send successfully!
```

Figure 14: shows the checksum and the status for every chunk sent from server to client



### 3.6 Packet Encoding

The Header and the Data that we are transmitting make up our packet. A Header identifying the source entity of the string is combined with the data. The frame is transformed to a binary then sent to the server over the socket. For instance, the data which consists of the files that were being sent at a specific time in the exchange between server and client. Given that the object supplying the information is a client entity, the header in this instance is Client. The data is combined with the header, translated to binary format, and delivered to the server after that. As soon as the requests reaches the worker, the worker decodes it again, splits the header from the data, and reads the contents of the header. The data is stored because the Header in this instance is "Client." Second, when the Workers and Server communicate, the Server "pickles" the list of all the requests that has received from the Client by concatenating the Header "Server" at the top to identify the sender entity. The message is subsequently transmitted to the Worker via the socket. It is then transformed to binary format when it gets to the Worker, the header is separated from the data, and when the dashboard has determined that the data was truly delivered by the Worker by reading the contents of the header, it has unpickled the list that was sent. The data is presented on the command line in a user-friendly way after being unpickled. This was all done using functions I implemented in server.py and client.py. In addition, -middle attacks are extremely probable, and they might lead to the disclosure of secret information that was intended to be kept private. I used a built-in socket library which provided me with more header information such as sequence number and message within the packet.

```
new_header = struct.pack(
    "!IIII", source, destination, header_info, status, m)
packet = udp_header + packets
sock.sendto(packet, ADDR)
new_header = struct.pack(
    "!IIII", source, destination, header_info, status, k)
data = aes.encrypt(chunk)
data = data.encode()
packet = udp_header + data
sock.sendto(packet, ADDR)
```

Listing 10: server.py and client.py

### 3.7 Docker and Network capture

In my system, each component in my program, server, client, and workers run within a separate Docker container, which is powered by Docker. The key benefit of this is that any container will have a distinct IP address, enabling accurate simulation of the various network components. In order to examine the protocol's functionality using data packets, Wireshark was used to capture the traffic in which the communications between client, server and workers were visible to us .

## 4 Discussion

In this section I am going to discuss why I decided to use this approach, why did I choose docker as my virtual interface and why I decided to choose these specific functionalities to implement in my program. In addition to raising some thoughts and questions about Wireshark and traffic that was captured by it.

### 4.1 The reason behind using Docker

I used Docker instead of local host as I wanted to learn how to create containers and images via dockers as my I made the components of my protocol "server, client, and workers" to connect with each other using Docker desktop. As you can see in figure 15 once I start the command line writing "docker start - i component" it will start working and program will start executing waiting for user interaction.










<input type="checkbox"/>		<b>worker3</b> 6b672e34110b 	csnetimage:latest	Running	-	21 seconds ag			
<input type="checkbox"/>		<b>worker2</b> c34abb2f2405 	csnetimage:latest	Running	-	20 seconds ag			
<input type="checkbox"/>		<b>worker1</b> 924a86fa78f2 	csnetimage:latest	Running	-	19 seconds ag			
<input type="checkbox"/>		<b>server</b> 5965bc8babf6 	csnetimage:latest	Running	-	19 seconds ag			
<input type="checkbox"/>		<b>client</b> 9ccd9e8094f5 	csnetimage:latest	Running	-	19 seconds ag			

Figure 15: shows the containers running with their own docker image

Communication between the components was happening via Docker Desktop, there is a direct communication between server and workers and server and client but no direct communication between worker to client or vice versa as it all run through the server “ingress”. Figure 16 shows the communication between them 3 components. As When the server is operational, clients can make multiple file requests. Server accepts several files that Client can ask for as input for this example. The data is sent to the server through our newly established UDP header when we are asked for the file names. After finding the file among the three workers, it will but sent to the server who returns it to the client.

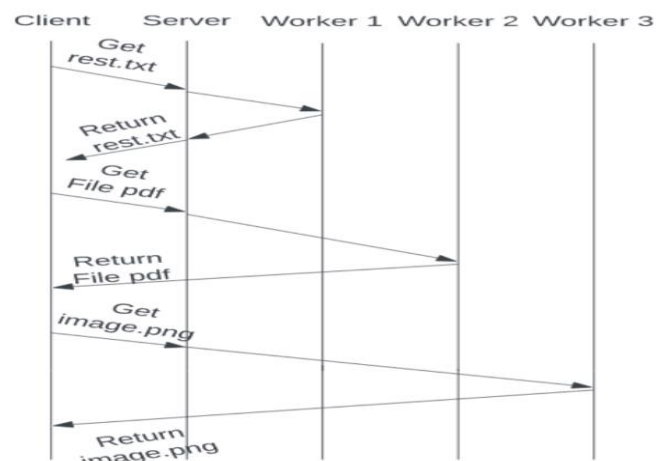


Figure 16: shows the direct and indirect communication between the components

## 4.2 Why I used these Functionalities

Functionalities I used to run this program which I talked in brief in my theory of topic section are forward error correction, multiplexing and encryption.

I used forward error correction so that during the transfer of the file, my program starts counting number of packets coming from the server, calculate the number of packets that are missing, and if that calculation returns with the missing packets, we request it from the ingress. To see the capability of forward error correction, the server must send them back again, this is shown in figure 17.

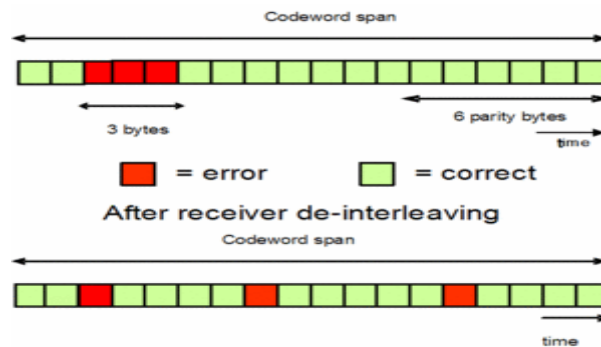


Figure 17: shows the procedure of forward error correction

I decided to use Multiplexing as it was easier for me to send stream of information over a communication link at the same time in the form of a single signal as it allowed me to transfer many signals to a single medium more efficiently in other words it converts multiple inputs into one single output. Figure 18 shows how multiplexing affected my program.

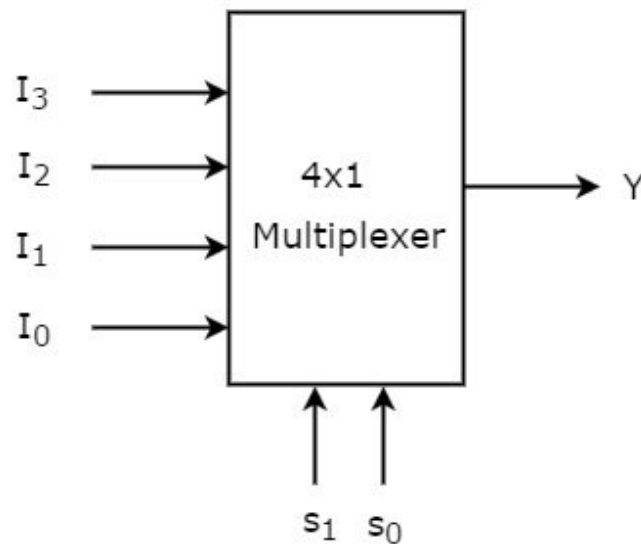


Figure 18: shows how multiplexing works

The most controversial functionality to me and still used was Encryption. I used AES Encryption for more secure and faster transfer. It was a turning point for my program as after I implemented it, my program started transferring files in a faster rate and it also allowed me to transfer large file much quicker, but the problem was when the transfer was done the data was encrypted and the name of the files that were transferred were not showed. I then decided to implement cipher text feedback which allowed me to block cipher to view the name of the file. Figures 19 and 20 will show you the difference between the encrypted data and decrypted data when it comes to transferring file. Figure 18 shows the encrypted data below the Data (111 bytes) as it shows the encryption key as well as I highlighted the name of the file which is also encrypted. Figure 20 shows the transfer after implementing cipher text feedback which allows the program to show the name of the files which I have highlighted.

### 4.3 Wireshark

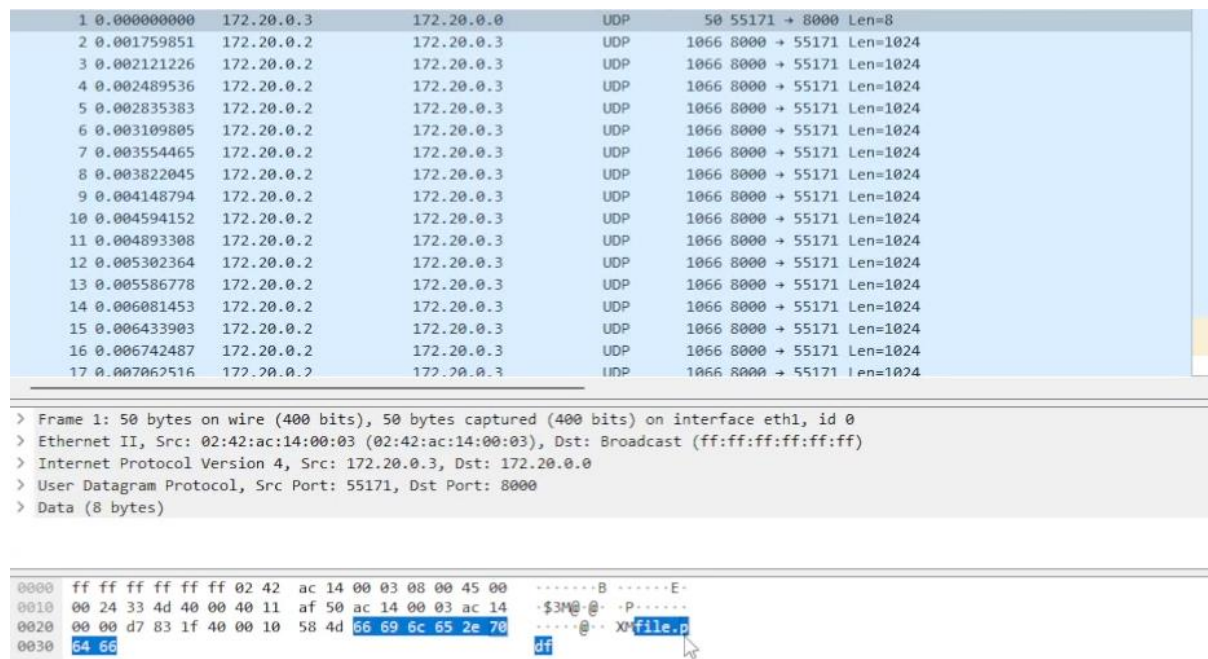
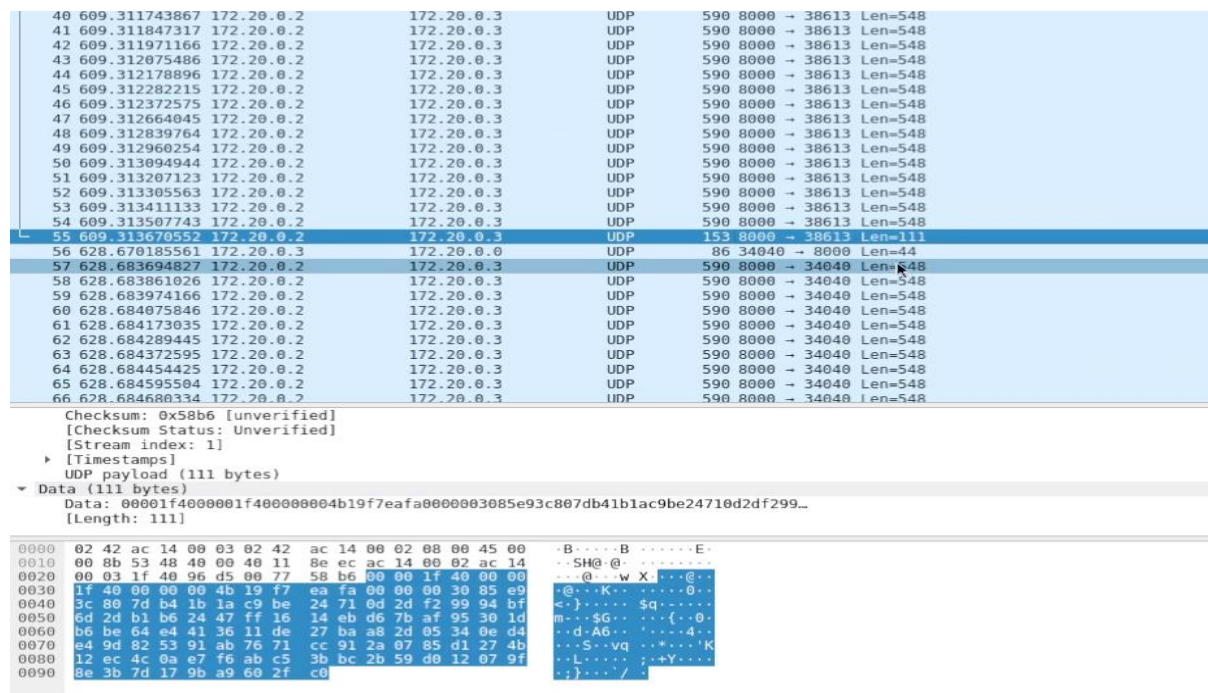


Figure 21 will summarize what I explained to you via a visual image that sums the difference between encrypted file (Ciphertext) and decrypted file (Plaintext).

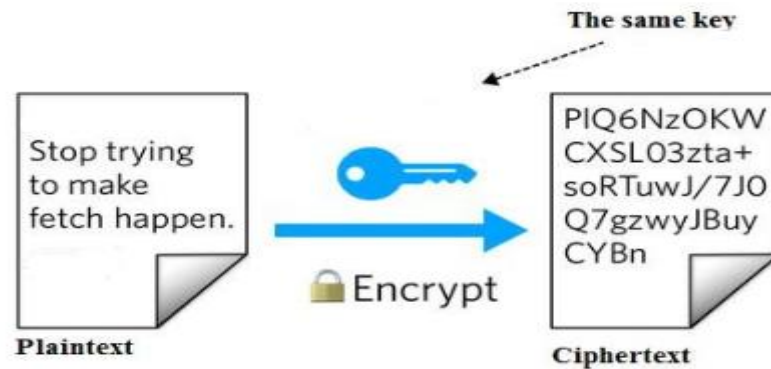


Figure 21: shows the ciphertext before and after

## 4.4 Other Design Decisions

I decided to replicate another User Datagram Protocol (UDP) header to test the functionality of my program in which this allowed me to follow the packets that are being received and it has also helped us to decrease packet loss. In addition, I decided to use Multiplexing as it helped my program to provide such as various file sizes, multiple concurrent streams and Encryption for more security in order to prevent any -middle attacks which are extremely probable, and they might lead to the disclosure of secret information that was intended to be kept private.

## 4.5 Features

- Worker classes run concurrently as when the server forwards the request to the worker, they will run concurrently within the server.
- Program has security as I implemented AES encryption for more security and for quicker files transfer.
- Program can transfer any file type using one or more packets (png, pdf, txt ... etc) as I showed in my video demonstration.
- Protocol takes packet loss into consideration as I explained in Theory of topic section
- To facilitate file transfers between the three components: client, server, and workers protocol encodes header information.

## 5 Summary

In this report I have described my implementation in Python of the File Retrieval Protocol. My implementation used 7 classes. Server class, Client class, Worker1 class, Worker2 class, Worker3 class, WorkerDirectory class and MultiplexEncrypt class. I used User Datagram protocol as as soon as a UDP packet is sent there is no acknowledgement from the destination. Also, the loss of packets is not retransmitted, followed by Stop and Wait ARQ to check the sequence number of the packets and the header. I used 3 main topologies "functionalities" in my UDP protocol: Forward error correction, Multiplexing and Encryption which I explained in detail in the Discussion, Implementation and Theory of

Topic sections. My knowledge of sockets, the protocol is demonstrated. The implementation's key elements are highlighted on a topological level in this document's description of the implementation. Docker is used to run my solution with every component of my solution having their own container within my Docker Desktop network. The system is composed of 3 components: 3 Workers, 1 Server, 1 Clients. The data that must be sent and the header that describes where the data is coming from make up a packet, which is used for communication. The header also identifies the sender and can be used to process the data in different ways depending on the use case specific to each sender. The traffic was captured using Wireshark. The following elements of the entire system can be summed up:

**Client Class:** requests any type of files (txt, png, pdf ... etc) from the server (ingress).

**Server Class:** selects a worker and forward the request to the worker and then after the retrieval of the file from the workers server will forward it to the client.

**workerDir Class** has the files that will soon be requested by the client.

**Workers (1,2,3) Classes:** will receive a request from the server about specific file, the worker will retrieve the file and send it to the server.

**multiplexEncrypt Class** has the functionalities I implemented for the UDP protocol

## 6 Reflection

This assignment helped me grasp protocols in a very valuable way, and I know that it will be very helpful for the future if I want to go into the networking field. Throughout this assignment, I learned a lot about Python networking. I had a great time putting this protocol into practice, as I now realize. I learned how to lay together a package, how to use basic acknowledgments, and how to consolidate the knowledge I had gained from the lectures. I faced numerous difficulties, such as learning Docker for the first time. I found using docker desktop challenging because I had never done it before. Overall, though, I'm quite pleased with the answer I've come up with and am eagerly anticipating the next assignment.

## 7 References

- [1] William Strunk, Jr. and E. B. White. *The Elements of Style*. Pearson Education, 4th edition, 1999.
- [2] Docker. Docker project page. <https://www.docker.com>
- [3] <https://abdesol.medium.com/udp-protocol-with-a-header-implementation-in-python-b3d8dae9a74b>
- [4] <https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2>
- [5] <https://docs.python.org/3/library/selectors.html>
- [6] <https://pythontic.com/modules/socket/udp-client-server-example>
- [7] <https://pymotw.com/2/socket/udp.html>
- [8] <https://www.youtube.com/watch?v=esLgiMLbRkI>
- [9] [https://linuxhint.com/send\\_receive\\_udp\\_python/](https://linuxhint.com/send_receive_udp_python/)
- [10] Trinity College. CSU33031 Lecture Slides
- [11] <https://docs.python.org/3/library/socket.html>