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

CS2031 Telecommunication II
Assignment #1: File Retrieval Protocol

Adriana Hrabowych, 19304296

October 29, 2022

Contents

1 Introduction

The focus of this assignment was to learn about protocol development and the information that is kept in a header to support functionality of a protocol, specifically for UDP datagrams. This is accomplished through creating a file retrieval protocol, where one or multiple clients can contact a server, alternatively called an ingress, who with the help of workers can retrieve files for the client. For this assignment, I took an approach of keeping packets simple so as to not overload the header with too much data, while still trying to maintain basic file retrieval functionality.
In the following report I will discuss my solution in greater detail, from a broad explanation of theory to a description of the network and packets I implemented.

2 Theory of Topic

Before beginning the assignment, through reviewing lecture material and independent sources, I researched the basic concept of both protocols, creating networks, and Java programming using sockets and packets.

I based my system model on the sample topology given in the assignment description, with only a few small differences, mainly in the client. After I had an idea of what components I wanted to include, I then focused on what the minimum data I actually needed to transfer within packets was in order for the system to work. In the end, my solution ended up with 3 unique components and 4 types of UDP packets.

2.1 QUIC and UDP

Using UDP packets was a requirement and through research, I decided that the best course of action was to loosely base my solution on the QUIC protocol.

The QUIC protocol was recommended as something to research when working on the assignment. QUIC stands for Quick UDP Internet Connection and was created essentially as a replacement to TCP for transfer protocols. The main difference between the two is that QUIC is faster at establishing connections, and can maintain connections over network changes. Both of these differences are because instead of recording connections using the ip/addresses of each endpoint, it's done by creating a unique connection ID.

UDP stands for User Datagram Protocol, it's a very lightweight protocol that allows users to send packets without establishing a connection nor verification. A UDP is made up of two parts, the header and the data that's being sent. The header is made up of the source port, destination port, length of the packet in bytes, and the checksum. Though my UDP packets do not include a checksum as they are optional in IPv4 which is what my solution runs on.

2.2 Components

As mentioned previously in the report, my solution has three unique components that act in the protocol. The functionalities of which are as follows,

Server: The server is the central hub of communication in the network, other components can only interact with the server and not each other so all communication must go through it. A server can have multiple clients and workers connected to it at once. The server handles file requests from a client and decides on which active worker to transfer the request to. It then receives files from a worker, sends an acknowledgement back to the worker, and then transmits the packet back to the client which requested it. Within the server is a list that keeps track of all the active workers connected to itself, the list is updated with every connect and disconnect request the server receives from workers. The server also responds with each c/d request with an acknowledgement.

Client: The client sends requests for specific files to the server, and upon receiving the correct file in response it then sends an acknowledgement back to the server.

Worker: The workers are meant to be able to come on and off line easily, and as such must properly inform the server whether or not they are currently active. Whenever a new worker is started, it automatically sends a connect request to the server, and becomes available for file retrieval once an acknowledgement is received in return. When a worker is to be shutdown, it sends a disconnect request to the server and only closes when it receives an acknowledgement in return. When a worker is active it can receive file requests from the server and create a new packet containing the file data to send back to the server, it then waits for an acknowledgement before becoming active again.

2.3 Packet Descriptions

As mentioned previously in the report, there are four unique types of UDP packets components can send between each other. Each one has the traditional UDP header and additional header information depending on its type. The types are as follows,

FileRequest: A request for a file, header includes the name of the requested file.

FileInfo: A file, header includes name of the file and size of the file data, data is the text of the file.

ConnectRequest: A request to change connection status with server, header includes the type of request (either connect or disconnect).

ACK: An acknowledgement, header includes information regarding the type of acknowledgement in string format.

2.4 Models

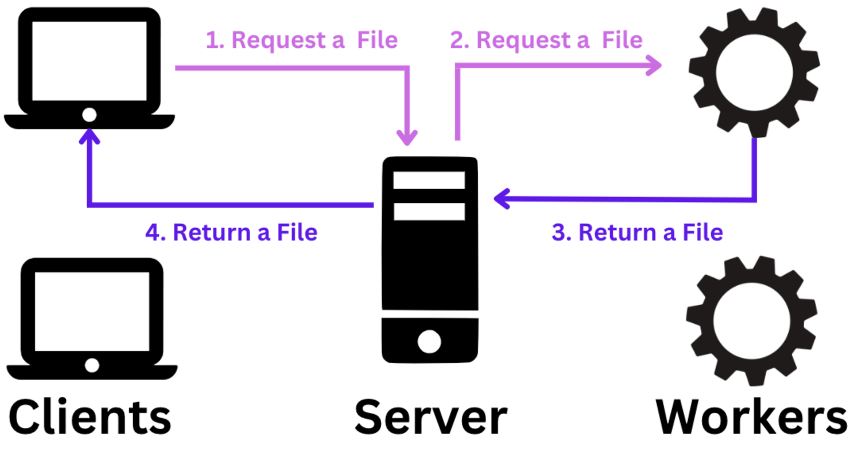


Figure 1: A diagram showcasing the fundamental design of the solution.

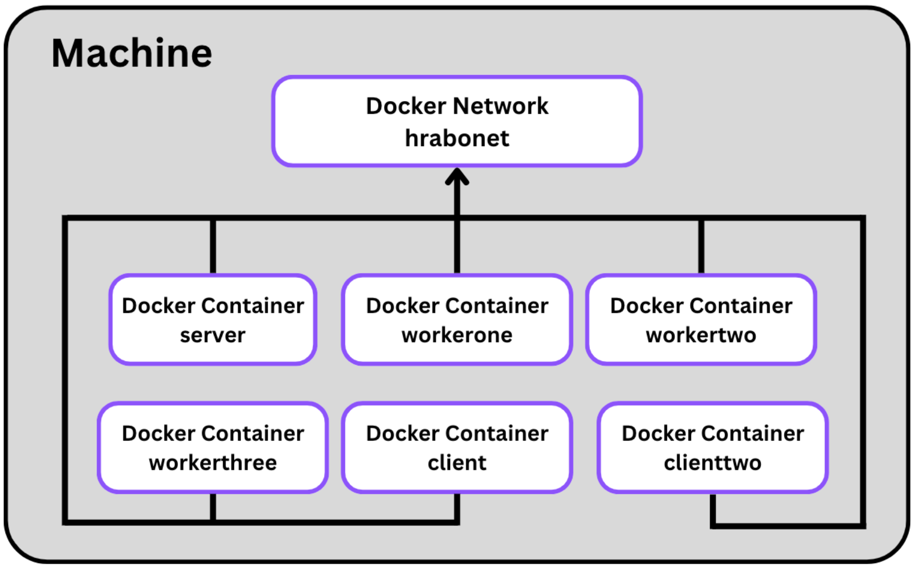

Machine
Docker Network hrabonet
Docker Container server
Docker Container workerone
Docker Container workertwo
Docker Container workerthree
Docker Container client
Docker Container clienttwo

Figure 2: A network diagram of the solution in Docker.

Figure one highlights the base design of my solution, with multiple clients and workers interacting with a single server in order to request and receive files. Figure two describes how each of the different components were connected using Docker, each component was ran within its own container which all communicated through a Docker bridge network called hrabonet. The IP addresses of each container were not included as depending on the order in which the containers are started each time, the IP addresses can vary.

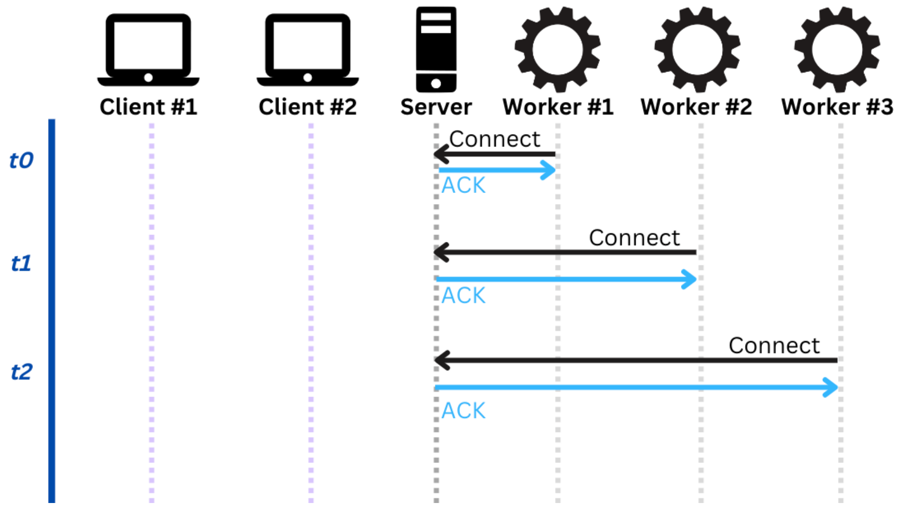


Figure 3: A flow diagram that shows each of the active workers connecting to the server upon startup.

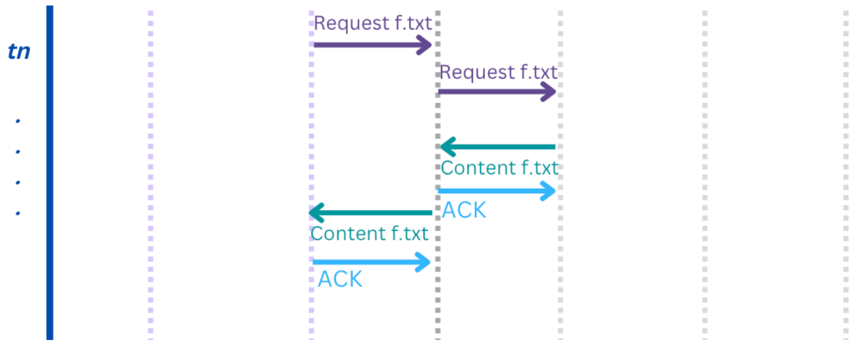


Figure 4: Continuing from figure 3, the flow diagram shows the process of a singular client requesting and receiving a file.

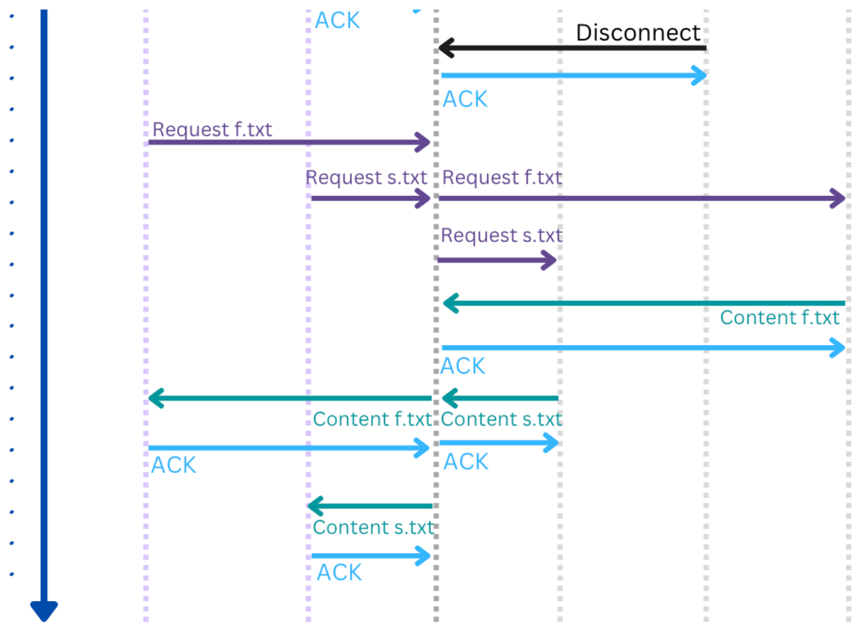


Figure 5: Continuing from figure 3 and 4, the flow diagram shows a more complicated process of multiple clients requesting files while one of the workers has disconnected and is therefore unusable.

3 Implemenation

This section discusses the details of the implementation of the prior theory. The solution was written using Java datagram packets and ran on Docker containers, XQuartz and wireshark were used to monitor the packets being sent. I used the sample Java code provided on blackboard as a base point for my solution.

3.1 Server

The server is an extension of the Node class given in the sample code, and is based upon the class of the same name also in the sample code. It's functionalities are exactly as described previously in the theory section, it can handle receiving all the different packet types: file requests, file contents, connect requests, and acknowledgements. Upon startup, the server sends no packets and immediately starts listening for incoming packets, it only acts when it receives a packet.

```
static final int DEFAULT_PORT = 50001;
static int WORKER_TO_PICK = 0;
private HashMap<InetSocketAddress , String> requests = new HashMap<>();
private ArrayList<InetSocketAddress> workers =
        new ArrayList<InetSocketAddress >();
```

Listing 1: The server contains a hashmap called *requests*, which it uses to store incoming requests by mapping the address of the requester and the name of the file requested. It also has an arraylist called *workers*, which stores the addresses of all the workers currently connected to itself.

```
if(WORKER_TO_PICK > workers.size() - 1)
        WORKER_TO_PICK = 0;
netSocketAddress workerdst = workers.get(WORKER_TO_PICK);
WORKER_TO_PICK++;
```

```
if(WORKER_TO_PICK > workers.size() - 1)
        WORKER_TO_PICK = 0;

packet.setSocketAddress(workerdst);
socket.send(packet);
```

Listing 2: Upon receiving a file request, the server uses static int *WORKERTOPICK* to choose which worker the request should be sent to from the arraylist of active workers. The array is traversed incrementally and then loops back to the start in order to keep one worker from taking every single request.

3.2 Client

The client is an extension of the Node class given in the sample code, and is based upon the class of the same name also in the sample code. It's functionalities are exactly as described previously in the theory section, it can only handle receiving file content packets. Upon startup, the client immediately sends a single file request packet to the server and waits for a response.

```
static final int DEFAULT_SRC_PORT = 50000;
static final int DEFAULT_DST_PORT = 50001;
static final String DEFAULT_DST_NODE = "server";
InetSocketAddress dstAddress;
```

Listing 3: The client has the port for the server hardcoded in, as the server is always thought to be up and running.

```
public synchronized void start() throws Exception {
        String fname;
        FileRequestContent frequest;

        DatagramPacket packet= null;
        fname= "message.txt";
        frequest = new FileRequestContent(fname);

        System.out.println("Sending request for file message.txt");

        packet= frequest.toDatagramPacket();
        packet.setSocketAddress(dstAddress);
        socket.send(packet);
        System.out.println("First request sent");
        this.wait();
}
```

Listing 4: The start method for the client, which includes sending a file request to the server and then waiting.

3.3 Worker

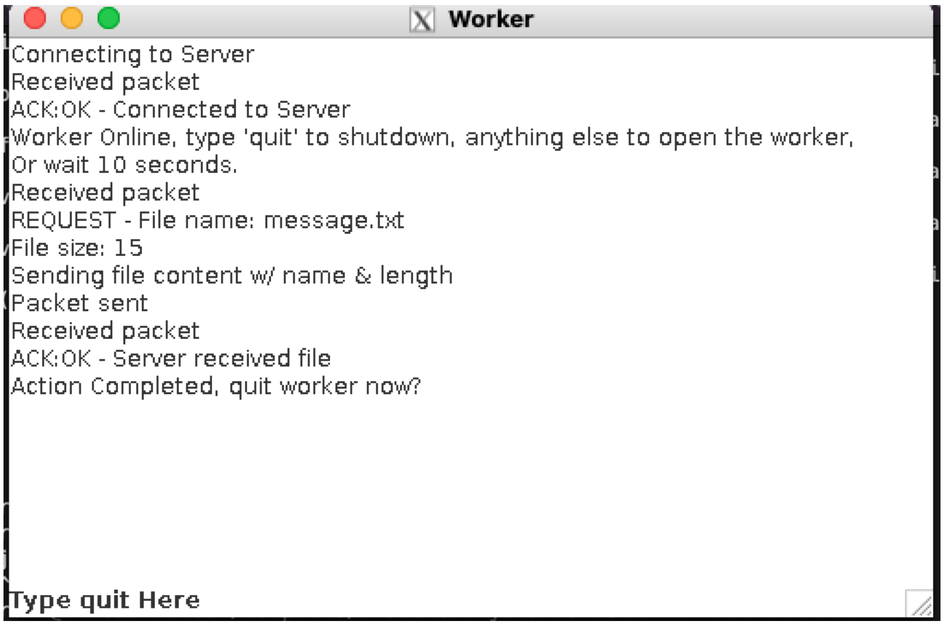
The worker is an extension of the Node class given in the sample code, and was based upon the client class from the sample code. It's functionalities are exactly as described previously in the theory section, it can handle receiving file requests and acknowledgements. Upon startup, a worker will immediately send a connection request to the server and then waits to receive any packets from the server. The worker makes use of a terminal class in order to allow for user interaction. Within ten seconds of a worker receiving a packet, the user can tell that worker to shutdown. Upon shutting down, the worker will send a disconnect request to the server to inform the server that it will no longer be available. The time limit of this ability is due to the limitations of the terminal class, a worker cannot be ready to receive text in the terminal and be ready to receive packets at the same time.

```java
static final int DEFAULT_SRC_PORT = 50000;
static final int DEFAULT_DST_PORT = 50001;
static final String DEFAULT_DST_NODE = "server";
InetSocketAddress dstAddress;
```

Listing 5: The worker has the port for the server hardcoded in, as the server is always thought to be up and running.

```java
public synchronized void start() throws Exception {
  workerTerminal.println("Connecting to Server");

  DatagramPacket connect= new ConnectContent("CONNECT").toDatagramPacket();
  connect.setSocketAddress(dstAddress);
  socket.send(connect);
  this.wait();

  workerTerminal.println("Worker Online, type 'quit' to shutdown,
        anything else to open the worker,");
  workerTerminal.println("Or wait 10 seconds.");
  while (true)
  {
        String userInput = workerTerminal.read("Type quit Here");
        if(userInput != null && userInput.equals("quit"))
        {
           workerTerminal.println("Sending Disconnect request to server");
           DatagramPacket disconnect;
           disconnect= new ConnectContent("DISCONNECT").toDatagramPacket();
           disconnect.setSocketAddress(dstAddress);
           socket.send(disconnect);
           System.exit(0);
        }
        this.wait();
        workerTerminal.println("Action Completed, quit worker now?");
  }
}
```

Listing 6: The start method for a worker. Upon starting, a connect request is sent to the server. Once an acknowledgement is received from the server the worker then starts the infinite loop. Each time a packet is received and the this.wait() is broken, the terminal will be able to accept responses for ten seconds.



```
Connecting to Server
Received packet
ACK:OK - Connected to Server
Worker Online, type 'quit' to shutdown, anything else to open the worker,
Or wait 10 seconds.
Received packet
REQUEST - File name: message.txt
File size: 15
Sending file content w/ name & length
Packet sent
Received packet
ACK:OK - Server received file
Action Completed, quit worker now?
```

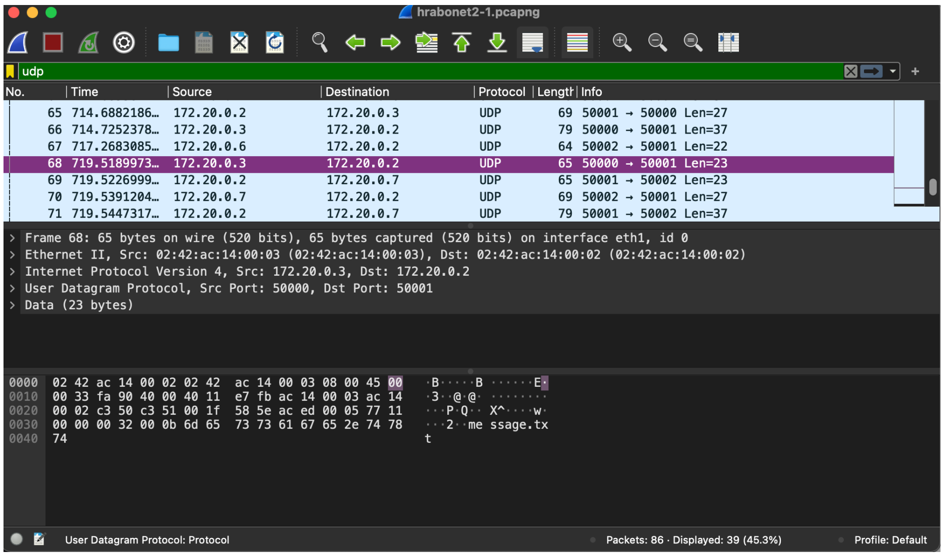
Figure 6: Example of the terminal for a worker. The worker is connected to the server and is sent one file request which it then completes and responds with a file content packet.

3.4 Packet Encoding

As mentioned previously, there are four types of packets in my solution, all of which inherit from the packetContent class given in the sample. The acknowledgement packet type is the exact same as the acknowledgement packet in the sample given. The same is true for the file info content packet type. The two new packet types are the file request and connect request type.

```
FileRequestContent(String filename) {
        type= FILEREQUEST;
        this.filename = filename;
}
```

Listing 7: The constructor for a file request packet, it's only data is the name of the file being requested.

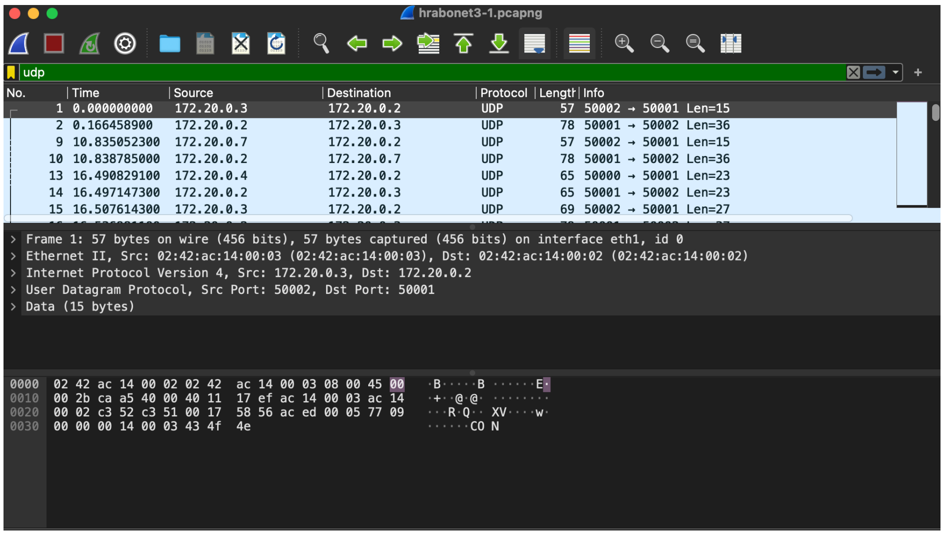


```
No.     Time          Source        Destination   Protocol Length Info
     65 714.6882186…  172.20.0.2    172.20.0.3    UDP      69 50001 → 50000 Len=27
     66 714.7252378…  172.20.0.3    172.20.0.2    UDP      79 50000 → 50001 Len=37
     67 717.2683085…  172.20.0.6    172.20.0.2    UDP      64 50002 → 50001 Len=22
     68 719.5189973…  172.20.0.3    172.20.0.2    UDP      65 50000 → 50001 Len=23
     69 719.5226999…  172.20.0.2    172.20.0.7    UDP      65 50001 → 50002 Len=23
     70 719.5391204…  172.20.0.7    172.20.0.2    UDP      69 50002 → 50001 Len=27
     71 719.5447317…  172.20.0.2    172.20.0.7    UDP      79 50001 → 50002 Len=37

> Frame 68: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on interface eth1, id 0
> Ethernet II, Src: 02:42:ac:14:00:03 (02:42:ac:14:00:03), Dst: 02:42:ac:14:00:02 (02:42:ac:14:00:02)
> Internet Protocol Version 4, Src: 172.20.0.3, Dst: 172.20.0.2
> User Datagram Protocol, Src Port: 50000, Dst Port: 50001
> Data (23 bytes)

0000  02 42 ac 14 00 02 02 42  ac 14 00 03 08 00 45 00   ·B·····B ······E·
0010  00 33 fa 90 40 00 40 11  e7 fb ac 14 00 03 ac 14   ·3··@·@· ········
0020  00 02 c3 50 c3 51 00 1f  58 5e ac ed 00 05 77 11   ···P·Q·· X^····w·
0030  00 00 00 32 00 0b 6d 65  73 73 61 67 65 2e 74 78   ···2··me ssage.tx
0040  74                                                 t

User Datagram Protocol: Protocol        Packets: 86 · Displayed: 39 (45.3%)        Profile: Default
```

Figure 7: Example of a file request packet as it's recorded as a pcap file on wireshark. Note the filename being encoded within the packet.

```
ConnectContent(String info) {
        type= CONNECTREQUEST;
        this.info = info;
}
```

Listing 8: The constructor for a connect request packet, it's only data is the information on whether it's a connect or disconnect type, CON or DIS.



```
No.     Time          Source        Destination   Protocol Length Info
      1 0.000000000   172.20.0.3    172.20.0.2    UDP      57 50002 → 50001 Len=15
      2 0.166458900   172.20.0.2    172.20.0.3    UDP      78 50001 → 50002 Len=36
      9 10.835052300  172.20.0.7    172.20.0.2    UDP      57 50002 → 50001 Len=15
     10 10.838785000  172.20.0.2    172.20.0.7    UDP      78 50001 → 50002 Len=36
     13 16.490829100  172.20.0.4    172.20.0.2    UDP      65 50000 → 50001 Len=23
     14 16.497147300  172.20.0.2    172.20.0.3    UDP      65 50001 → 50002 Len=23
     15 16.507614300  172.20.0.3    172.20.0.2    UDP      69 50002 → 50001 Len=27

> Frame 1: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface eth1, id 0
> Ethernet II, Src: 02:42:ac:14:00:03 (02:42:ac:14:00:03), Dst: 02:42:ac:14:00:02 (02:42:ac:14:00:02)
> Internet Protocol Version 4, Src: 172.20.0.3, Dst: 172.20.0.2
> User Datagram Protocol, Src Port: 50002, Dst Port: 50001
> Data (15 bytes)

0000  02 42 ac 14 00 02 02 42  ac 14 00 03 08 00 45 00   ·B·····B ······E·
0010  00 2b ca a5 40 00 40 11  17 ef ac 14 00 03 ac 14   ·+··@·@· ········
0020  00 02 c3 52 c3 51 00 17  58 56 ac ed 00 05 77 09   ···R·Q·· XV····w·
0030  00 00 00 14 00 03 43 4f  4e                        ······CO N
```

Figure 8: Example of a connect request packet as it's recorded as a pcap file on wireshark. Note the connection type being encoded within the packet, as this packet is a connect and not a disconnect request, CON is encoded.

4 Discussion

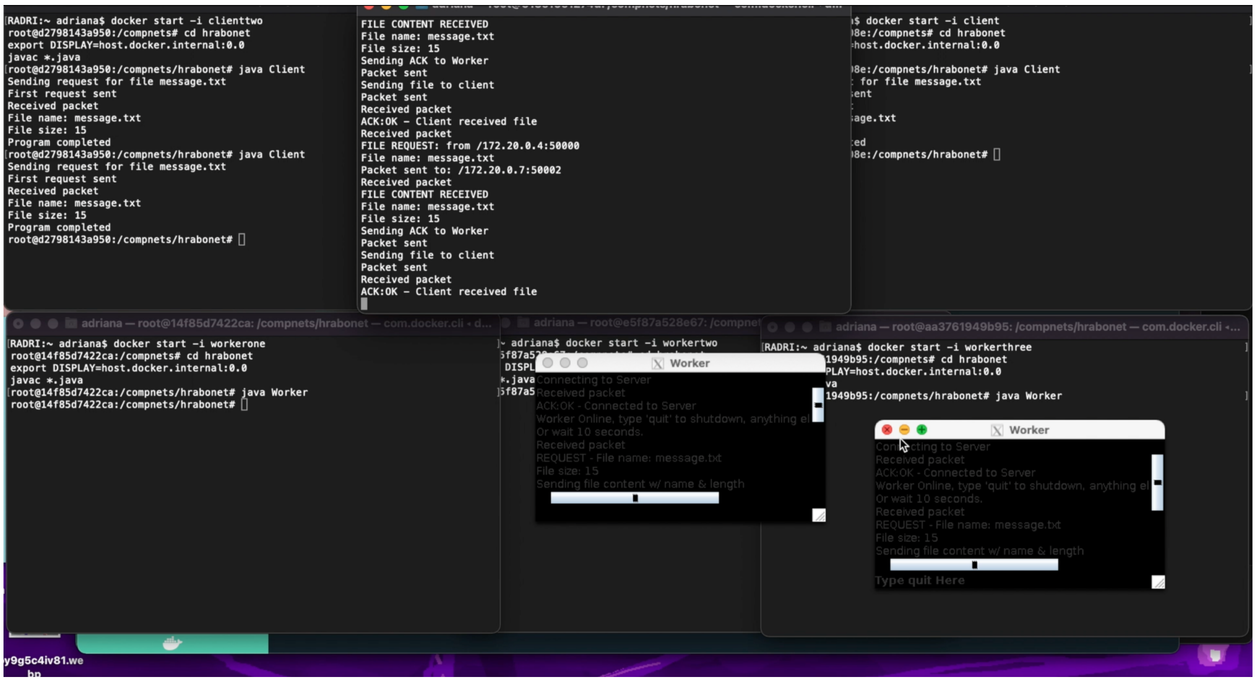
In this section I will discuss the various decisions that were made in the process of creating the implementation, and why the conclusions were made.

Using docker instead of localhost, kubernetes, or other virtual infrastructure softwares was decided due to the fact that it's what seemed to have a lot of easy to read online documentation to kickstart the process. Also, as I use a mac for most of my assignments, docker seemed like the most lightweight option that would work for my computer.

Being able to have multiple workers and clients connected to a single server was done partly due to the fact that having multiple workers is a requirement but also that it makes my solution more advanced. It also lends itself to going along with real world logic, as it wouldn't make sense for every client to have its own server.

Having four different types of packets may seem a bit overkill, but each one is necessary for the protocol to work. Acknowledgements increase security, and they make error checking easier as it gives a trail to track on the terminal as packets are sent back and forth. The other three are vital to the actual functionality of the protocol, without them it would be an entirely different solution. It might have been possible to combine the different types and have some sort of system to easily distinguish between them from the encodings alone but it would of made the packets and code harder to understand.

Having the workers accept user input was something I personally wanted to implement, it wasn't a necessary feature and it did increase the amount of error handling needed as human interaction leads to human error. But again, I felt like this inclusion made my protocol more unique and it made more real world sense for workers to be able to start and end at any time.



```
RADRI:~ adriana$ docker start -i clienttwo
root@d2798143a950:/compnets# cd hrabonet
export DISPLAY=host.docker.internal:0.0
javac *.java
root@d2798143a950:/compnets/hrabonet# java Client
Sending request for file message.txt
First request sent
Received packet
File name: message.txt
File size: 15
Program completed
root@d2798143a950:/compnets/hrabonet# java Client
Sending request for file message.txt
First request sent
Received packet
File name: message.txt
File size: 15
Program completed
root@d2798143a950:/compnets/hrabonet# 
```

```
FILE CONTENT RECEIVED
File name: message.txt
File size: 15
Sending ACK to Worker
Packet sent
Sending file to client
Packet sent
Received packet
ACK:OK - Client received file
Received packet
FILE REQUEST: from /172.20.0.4:50000
File name: message.txt
Packet sent to: /172.20.0.7:50002
Received packet
FILE CONTENT RECEIVED
File name: message.txt
File size: 15
Sending ACK to Worker
Packet sent
Sending file to client
Packet sent
Received packet
ACK:OK - Client received file
```

```
$ docker start -i client
8e:/compnets# cd hrabonet
host.docker.internal:0.0

8e:/compnets/hrabonet# java Client
 for file message.txt
ent

age.txt

ed
8e:/compnets/hrabonet# 
```

```
RADRI:~ adriana$ docker start -i workerone
root@14f85d7422ca:/compnets# cd hrabonet
export DISPLAY=host.docker.internal:0.0
javac *.java
root@14f85d7422ca:/compnets/hrabonet# java Worker
root@14f85d7422ca:/compnets/hrabonet# 
```

```
~ adriana$ docker start -i workertwo
5f87a5
 DISPL
*.java
5f87a5
```

```
Connecting to Server
Received packet
ACK:OK - Connected to Server
Worker Online, type 'quit' to shutdown, anything el
Or wait 10 seconds.
Received packet
REQUEST - File name: message.txt
File size: 15
Sending file content w/ name & length
```

```
RADRI:~ adriana$ docker start -i workerthree
1949b95:/compnets# cd hrabonet
PLAY=host.docker.internal:0.0
va
1949b95:/compnets/hrabonet# java Worker
```

```
Connecting to Server
Received packet
ACK:OK - Connected to Server
Worker Online, type 'quit' to shutdown, anything el
Or wait 10 seconds.
Received packet
REQUEST - File name: message.txt
File size: 15
Sending file content w/ name & length
Type quit Here
```

Figure 9: Screenshot showcasing a working example of the solution with 2 clients, one server, and three workers.

5 Summary

This report has described my research, my idea, and finally my solution for a file retrieval protocol using docker and UDP packets. My final solution included three components communicating over a shared network, sending four different types of packets in order to achieve both the assignment brief and my proposed topology. The solution showcases all that I have learned about protocols and UDP packets, and with that knowledge I have created a sophisticated network program that allows clients to request files and a server and workers to retrieve them.

6 Reflection

At the end of this assignment, I find myself thoroughly proud of my completed work. Though some might find my solution simplistic in some senses, given that I was a complete beginner in using docker and coding for networks, the code I've accomplished here is quite well done and meets all the requirements set for the assignment. Overall I'd say I spent about 25 hours researching, coding, being in labs, and writing this report. If I had more time, or perhaps less assignments, I would have liked to add more user interaction, or at the very least try to perfect the user interaction I've included already. Also, the way I structured my packets potentially could have been more streamlined and if I had to start the assignment over again I think I would have completely changed the way I did my packets. Lastly, one last feature I didn't implement but had thought of, was making it so larger files could be encoded in multiple packets.

Overall, despite the occasional frustration, I learnt a lot about socket programming and packet encoding due to this assignment. I liked having the two smaller submissions for the project before the final as it provides useful feedback going forward, though I do feel as if the marking scheme for the interim submissions was not always as clear. Personally, though I understand the need for them, I feel as though the videos are quite time consuming and elaborate to create and I found myself spending just as much time making them as coding the actual solution.