



---

# CSU33031 Computer Networks II

## Assignment #1: Routing Protocols

---

Thomas Moroney, 20332993

November 2, 2022

### Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theory of Topic</b>	<b>2</b>
2.1	UDP . . . . .	2
2.2	Flow Control . . . . .	2
2.3	Design of Solution . . . . .	3
<b>3</b>	<b>Implementation</b>	<b>5</b>
3.1	Client Implementation . . . . .	5
3.2	Ingress Implementation . . . . .	7
3.3	Worker Implementation . . . . .	9
3.4	User Interaction . . . . .	11
3.5	Packet Encoding . . . . .	11
<b>4</b>	<b>Demonstration</b>	<b>12</b>
<b>5</b>	<b>Summary</b>	<b>17</b>
<b>6</b>	<b>Reflection</b>	<b>17</b>

## 1 Introduction

The problem description for this assignment is to create a network protocol that provides a mechanism to retrieve files from a service based on UDP (User Datagram Protocol) datagrams. It defined that the solution should facilitate the transfer of data packets from one node to another with the source and destination address encoded in binary in the header of each packet. The protocol involves a number of actors: one or more client, an ingress node, and one or more workers. In order to facilitate the low-latency transfer of each datagram packet, the header is designed to be as lightweight as possible.

In my solution, I have one client, an ingress, and three workers. The workers have access a directory that contains a number of files. The client can send a request packet containing the name of a file. The ingress will receive the request and then decide which worker to send it on to. Then if the worker has the file, it will

send it on to the Ingress, which will forward the file to the Client.

In the following, I will discuss the mechanism that was fundamental to my solution. This will be followed by the overall design of the network elements of my solution and a description of the data units that are exchanged between the network elements.

## 2 Theory of Topic

In this section, I will explain my understanding of UDP Datagrams and I will discuss the Flow Control protocol I used. This will then lead into the explanation of my design of the solution. The explanation of the design will describe the individual network elements, followed by a description of the packets that are exchanged between these network elements.

### 2.1 UDP

User Datagram Protocol (UDP) is a communications protocol designed for the low-latency transfer of data across the internet. UDP speeds up the transmissions by allowing the transfer of data without waiting for acknowledgment that the packet has been received. Since UDP is connectionless, there is no guarantee that when a packet is sent it will be received by the other party. Being connectionless also means that it does not require prior communication to set up communication channels or data paths (allows the communication between two network endpoints without a prior arrangement). The User Datagram Protocol (UDP) is all about efficiency, so it will fire each packet off as fast as possible without waiting for recognition they are being received. It is the job of the Flow Control protocol that is chosen to make sure that each packet arrives. UDP is used to transfer Datagram packets which are self-contained with source and destination addresses written in the header.

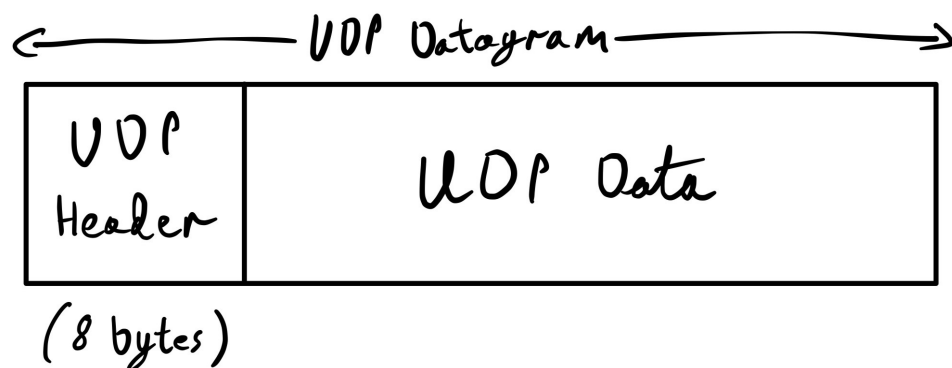


Figure 1: Figure that shows the layout of a UDP Datagram packet.

### 2.2 Flow Control

Flow control is the method for the management of data flow between nodes. The Flow Control protocol chosen for this assignment is the "Simplest" protocol. The Simplest protocol does not have any error control and it doesn't actually check if the file has been received by the other party because it is designed for use in noiseless channels. In noiseless channels there is guaranteed to be no interference, so we can safely assume that every packet reaches its destination. Theoretically, if a packet were to go missing and not reach its destination, then the program would not stop and wait or even send it again, it would simply move on to sending the next packets.

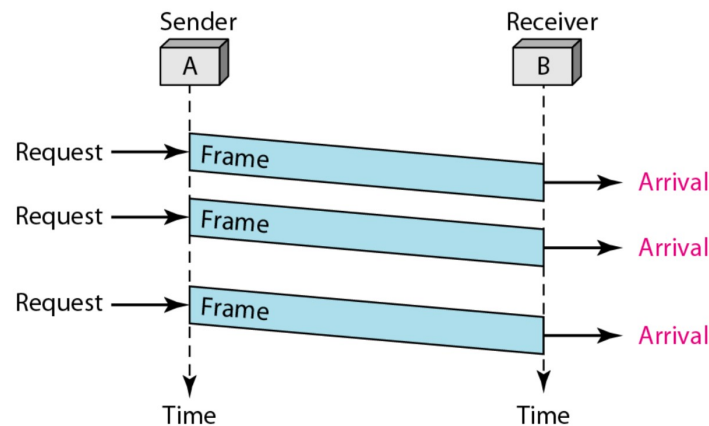


Figure 2: Figure that demonstrates the Simplest Flow Control protocol.

## 2.3 Design of Solution

Firstly, I used docker containers to store each of the actors. I decided to create two docker networks called csnet and csnet2. The client and ingress are connected via csnet, while the workers and ingress are connected via csnet2. Since the client and ingress share a network they can send packets back and forth between them, and the same goes for the workers and the ingress. So therefore the client and workers can communicate with each other through the ingress. I decided to make three different types of packets for use in my design. The request packet contains the name of a file to be retrieved from the workers. The client will send a request packet to the Ingress, which will then forward it to a worker. When the worker receives the request packet, it will check if the name contained in the packet matches any of the files it has access to. If it finds a file that matches, it will send back a FileInfo packet to the client through the ingress. This FileInfo packet contains the name and the size of the file. If the file does not exist, the worker will send a packet back to the ingress to let it know that the file was not found. The ingress will then forward this packet to the client.

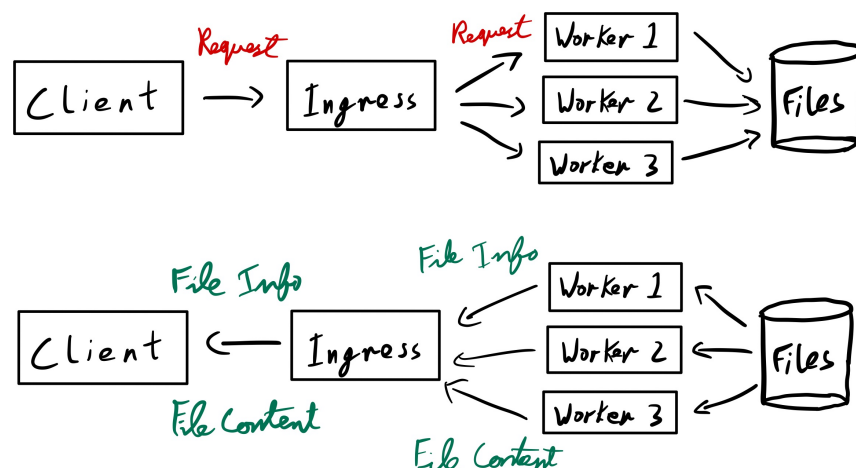


Figure 3: The figure above is topology of my network. The network is using UDP and the Simplest Flow Control protocol. The client sends a request packet to the Ingress, which then forwards this packet on to a worker. The chosen worker returns the requested file to the Ingress, which forwards this packet on to the Client. First the worker sends an info packet, and then a content packet.

Once the worker has sent the FileInfo packet, it will then start sending FileContent packets. The worker will convert the content of the file into a number of byte arrays. Each of these byte arrays is then turned into a FileContent packet and sent to the Ingress. The amount of byte arrays created, and therefore packets created, depends on the size of the file and the specified size of each packet. When the client receives the FileContent packet from the ingress, it will extract the byte array from the FileContent packet and write the contents to a new file created in the client's directory. Once the worker has sent all the packets, the client should have a complete copy of the file in its possession.

On receipt of a packet from a worker, the ingress will check the packet type. If it has received a FileInfo or FileContent packet, then the packet will be forwarded to the client. If it has received a Request packet, then the ingress knows it is from the client, and will forward the packet to the worker. I chose to use the Simplest Protocol for flow control, which means that the worker does not wait to see if each packet has been received before sending more packets. Instead the worker sends one packet after another without stopping. When the client receives a packet, it starts by sorting out which type of packet it is. The first packet the client will receive is the FileInfo packet which tells the client how many more packets to expect, as it contains the size of the file.

My system has support for large files and is able to transmit text files and png or jpg images.

## 3 Implementation

### 3.1 Client Implementation

```
public synchronized void onReceipt(DatagramPacket packet) {
    try {
        System.out.println("Packet Recieved");
        PacketContent content = PacketContent.fromDatagramPacket(packet);
        if (content.getType() == PacketContent.FILEINFO) {
            fileSize = ((FileInfoContent)content).getFileSize();
            fileName = ((FileInfoContent)content).getFileName();
            System.out.println("File name: " + fileName);
            System.out.println("File size: " + fileSize);
            path = "Client_files/" + fileName;
            byteArray = new byte[fileSize];

            if (fileName.equals("notfound")) {
                System.out.println("No worker had the requested file.");
                notify();
            }
            else {
                file = new File(path);
                System.out.println("File path is: " + path);
                boolean value = file.createNewFile(); // false if file already exists
                if (!value){
                    System.out.println("The file already exists");
                }
                output = new FileWriter(path);
                packetNum = 0;
            }
        }
    }
}
```

Listing 1: Client onReceipt() method shows how the client handles incoming packets. When the Client receives a packet, it uses the onReceipt() method to sort the packet by type and read it's contents. If a FileInfo packet is received, it will read the file name contained in the packet and check if the file already exists on the client-side. If it does not already exist, it will create a new file with the same name as in the packet. A byte array with the same size as the file will also be created.

```
else if (content.getType() == PacketContent.BYTEARRAY) {
    System.out.println("Received packet number: " + (packetNum+1));
    FileContent fileContent = (FileContent)content;
    filetype = fileName.substring(fileName.lastIndexOf(".")+1); // get filetype
    if (filetype.equals("txt")) {
        String s = new String(fileContent.byteArray, StandardCharsets.UTF_8);
        s = s.substring(PADDING, s.length());
        output.write(s); // writes contents of byte array to new file

        fileSizeRead = fileSizeRead + (PACKETSIZE - PADDING);
        if (fileSizeRead >= fileSize) {
            output.close();
            notify();
        }
    }
}
else if (filetype.equals("png") || filetype.equals("jpg")) {
    byte[] imgArray = Arrays.copyOfRange(fileContent.byteArray, PADDING, PACKETSIZE);
    // First few bytes in each packet are reserved for the header

    if (fileSizeRead + imgArray.length >= fileSize) { // received all packets
        int lastPacketSize = (int) (fileSize-fileSizeRead);
        System.arraycopy(imgArray, 0, byteArray, fileSizeRead, lastPacketSize);
        ByteArrayInputStream bis = new ByteArrayInputStream(byteArray);
        BufferedImage bImage = ImageIO.read(bis);
        ImageIO.write(bImage, filetype, new File(path) );
        System.out.println("Image created");
        notify();
    }
    else {
        System.arraycopy(imgArray, 0, byteArray, fileSizeRead, imgArray.length);
    }
    fileSizeRead = fileSizeRead + imgArray.length;
}
packetNum++;
}
```

Listing 2: If the Client receives a file that is of type FileContent (called BYTEARRAY), then the onReceipt method will read the byte array contained in the packet and convert it to a string. The padding reserved for the header is removed from the string and then the string is written to the new file that was created earlier (when the FileInfo packet was received).

```
public synchronized void start() throws Exception {
    Scanner input = new Scanner(System.in);
    while (true) {
        System.out.print("What file are you requesting? ");
        String filename = input.nextLine();
        if (filename.equals("exit")) {
            input.close();
            return;
        }
        else {
            System.out.println("Requested " + filename);
        }

        DatagramPacket request;
        request = new ReqPacketContent(filename).toDatagramPacket();
        request.setSocketAddress(ingressAddress);
        socket.send(request);
        this.wait();
    }
}
```

Listing 3: start() method that runs when Client is started and when notify() is called. This method asks the user what file they would like to request. This filename inputted by the user is then converted to a Datagram packet and sent to the Ingress.

### 3.2 Ingress Implementation

```
public void onReceipt(DatagramPacket packet) {
    try {
        System.out.println("Received packet");
        PacketContent content= PacketContent.fromDatagramPacket(packet);
        if (content.getType()==PacketContent.REQPACKET) {
            request = new
                ReqPacketContent(((ReqPacketContent)content).getPacketInfo()).toDatagramPacket();
            InetSocketAddress workerAddress = new InetSocketAddress(WORKER1_NODE, WORKER1_PORT);
            switch (workerNum) {
                case 1:
                    workerAddress = new InetSocketAddress(WORKER1_NODE, WORKER1_PORT);
                    break;
                case 2:
                    workerAddress = new InetSocketAddress(WORKER2_NODE, WORKER2_PORT);
                    break;
                case 3:
                    workerAddress = new InetSocketAddress(WORKER3_NODE, WORKER3_PORT);
                    break;
                default:
                    workerAddress = new InetSocketAddress(WORKER1_NODE, WORKER1_PORT);
            }
            System.out.println("Forwarded request packet to Worker "+workerNum);
            request.setSocketAddress(workerAddress);
            socket.send(request);
            if (workerNum == 3) { // Round robin system for workers
                workerNum = 1;
            }
            else {
                workerNum++;
            }
        }
    }
}
```

---

Listing 4: Shows the Ingress' onReceipt() function that deals with passing packets between the Client and the Worker. When a Request packet is received, it forwards it to one of the workers (depending on the value in workerNum. Every time a request is received, workerNum is incremented so that the next request is sent to a different worker. This way three requests can be handled almost simultaneously.

```
        else if (content.getType()==PacketContent.FILEINFO) {
            if (((FileInfoContent)content).getFileName().equals("notfound")) {
                System.out.println("File not found in Worker "+workerNum+", sent 'notfound' packet
                    to Client");
            } // filename is "notfound" so the Client knows that the file does not exist
            else {
                System.out.println("Forwarded FileInfo packet to Client");
            }
            clientAddress = new InetSocketAddress(CLIENT_NODE, CLIENT_PORT);
            packet.setSocketAddress(clientAddress);
            socket.send(packet);
        }
        else if (content.getType()==PacketContent.BYTEARRAY) {
            System.out.println("Forwarded FileContent packet to Client");
            clientAddress = new InetSocketAddress(CLIENT_NODE, CLIENT_PORT);
            packet.setSocketAddress(clientAddress);
            socket.send(packet);
        }
    }
```

Listing 5: If the Ingress receives either a FileInfo or FileContent packet (BYTEARRAY), it will forward the packet straight to the Client.



### 3.3 Worker Implementation

```
public void sendInfoPacket(FileInfoContent fileInfo) {
    try {
        DatagramPacket infoPacket= fileInfo.toDatagramPacket();
        infoPacket.setSocketAddress(ingressAddress);
        socket.send(infoPacket); // Send packet with file name and length
        System.out.println("Sent Info-Packet w/ name & length of file");
    }
    catch(Exception e) {e.printStackTrace();}
}
```

Listing 6: I decided to make a separate function in the Worker that handles the sending of FileInfo packets due to the code being reused so much in the Worker class.

```
public void onReceipt(DatagramPacket packet) {
    try {
        PacketContent content= PacketContent.fromDatagramPacket(packet);

        if (content.getType()==PacketContent.REQPACKET) {
            System.out.println("Received request packet");

            String fname;
            String filetype;
            File file= null;
            FileInputStream fin= null;
            FileInfoContent fileInfo;
            double totalPackets;
            int packetLength;
            int lastPacketLength;
            byte[] buffer= null;
            DatagramPacket infoPacket = null;
            DatagramPacket contentPacket = null;

            fname= ((ReqPacketContent)content).getPacketInfo(); // name of file requested
            file= new File("Worker_files/" + fname);

            String input = null;
            if (file.exists()) {
                System.out.println("Found " + fname);
                System.out.print("Do you want to send this file? (yes/no): ");
                Scanner scanner = new Scanner(System.in);
                input = scanner.nextLine();
                if (input.equals("no")) {
                    System.out.println("File not sent.");
                }
            }
            else {
                System.out.println("Requested file was not found!");
            }
        }
    }
}
```

Listing 7: The Worker's onReceipt() method deals with packets coming from Ingress. First it confirms that the incoming packet is a Request packet (REQPACKET). The Worker then checks if the requested file exists in it's directory. If the file exists, the user will be prompted for confirmation that they want this file to be returned to the ingress.

```
if (file.exists() && input.equals("yes")) { // SEND FILE REQUESTED
    System.out.println("Sending " + fname);
    System.out.println("File size: " + file.length());

    packetLength = PACKETSIZE-PADDING; // PADDING is the space reserved for the header
    filetype = fname.substring(fname.lastIndexOf(".") + 1); // get filetype
    if (filetype.equals("txt")) {
        fileInfo = new FileInfoContent(fname, (int) file.length());
        sendInfoPacket(fileInfo); // Send FileInfo packet

        buffer = new byte[(int) file.length()];
        fin = new FileInputStream(file);
        totalPackets = Math.ceil(buffer.length / packetLength);

        BufferedInputStream bis = new BufferedInputStream(fin);
        for (int i = 0; i <= totalPackets; i++) {
            byte[] byteArray = new byte[PACKETSIZE];
            if (i == totalPackets) {
                lastPacketLength = (int) (buffer.length - ((totalPackets - 1) * packetLength));
                byteArray = new byte[lastPacketLength];
            }
            int bytesReadIn = bis.read(byteArray, 0, byteArray.length - PADDING);
            System.out.println("Sent FileContent Packet Number: " + (i + 1));

            FileContent fileContent = new FileContent(byteArray);
            contentPacket = fileContent.toDatagramPacket();
            contentPacket.setSocketAddress(ingressAddress);
            socket.send(contentPacket);
        }
        fin.close();
    }
}
```

Listing 8: If the user inputs "yes" then the worker will check if the file is a text or image file. The Worker will then create a FileInfo packet with the name and size of the file, and send it to the Ingress, and then on to the Client. This size value is necessary for the Client to create a byte array that is the same size as the entire file. After sending the FileInfo packet, the worker will proceed to read the file into a BufferedInputStream, which can then be packetized and sent off to the Ingress.

```

else if (filetype.equals("png") || filetype.equals("jpg")) {
    BufferedImage bImage = ImageIO.read(file);
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ImageIO.write(bImage, filetype, bos );
    byte [] byteArray = bos.toByteArray();

    fileInfo= new FileInfoContent(fname, (int) byteArray.length);
    sendInfoPacket(fileInfo);

    totalPackets = Math.ceil(byteArray.length/packetLength);
    System.out.println("Total Packets: " + totalPackets+1);
    for (int i = 0; i <= totalPackets; i++) {
        byte[] tmp = new byte[PACKETSIZE];
        if(i==totalPackets){
            lastPacketLength = (int) (byteArray.length % packetLength);
            tmp = new byte[lastPacketLength+PADDING];
            System.arraycopy(byteArray, (i * packetLength), tmp, 7, lastPacketLength);
        }
        else {
            System.arraycopy(byteArray, (i * packetLength), tmp, 7, packetLength);
        }
        FileContent fileContent = new FileContent(tmp); // Creates new filecontent
            using array of bytes from buffer
        contentPacket = fileContent.toDatagramPacket();
        contentPacket.setSocketAddress(ingressAddress);
        socket.send(contentPacket);
        System.out.println("Sent packet number: " + (i+1)); // there is no packet zero
    }
}
else { // Send empty packet to let ingress know that no file was found.
    fileInfo= new FileInfoContent("notfound", 0);
    sendInfoPacket(fileInfo);
}

```

Listing 9: This is the procedure for when an image file is requested. Firstly it reads the file using a `BufferedImage`, and then writes the file to a `ByteArrayOutputStream`, which is then converted to a byte array. This byte array is then split up into packets and sent one by one to the Ingress. The "padding" at the beginning of each packet is necessary in order to make space for the header. If the file requested is not found, then the Worker will return a `FileInfo` packet with the filename "notfound".

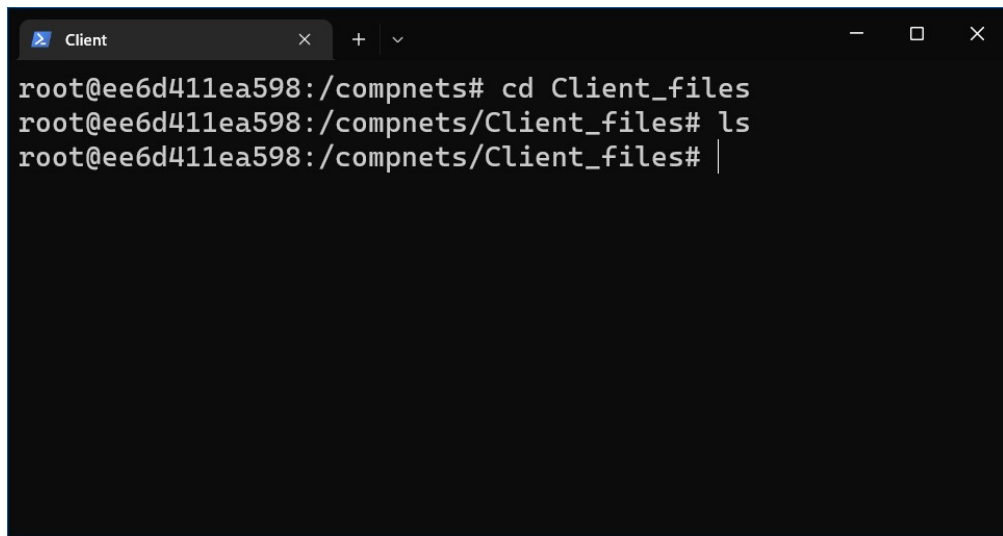
### 3.4 User Interaction

The user interacts with the system using the client and the Worker. The user can enter either an image or text file name that they would like to receive. This request is sent to the Ingress which receives the request and forwards it to a Worker. Once the worker receives the request and finds the file in it's directory, the user is asked if this is indeed the file they want to be returned to them.

### 3.5 Packet Encoding

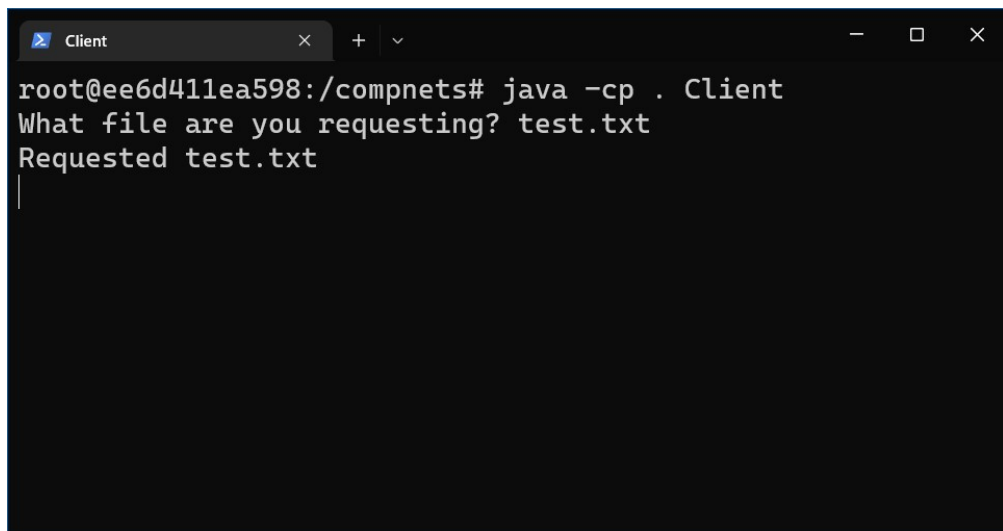
My program uses UDP datagram packets. The Request packet contains the type, port, and file name. These are written to the object output stream. The `FileInfo` packet contains the type, port, file name and file size which is also written to the object output stream. The `FileContent` packet contains the type, port and the file content, which is first converted to a `BufferedInputStream` and then converted to an `ObjectOutputStream`.

## 4 Demonstration

A terminal window titled "Client" with standard window controls. The prompt is root@ee6d411ea598:/compnets#. The user enters 'cd Client\_files', the prompt changes to root@ee6d411ea598:/compnets/Client\_files#, and then 'ls' is entered, resulting in a blank line, indicating the directory is empty.

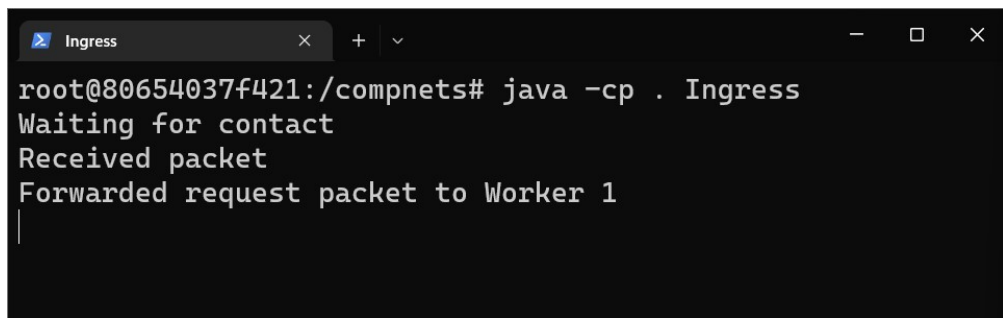
```
root@ee6d411ea598:/compnets# cd Client_files
root@ee6d411ea598:/compnets/Client_files# ls
root@ee6d411ea598:/compnets/Client_files# |
```

Figure 4: We can see here that the Client files directory is currently empty. This is where the Client stores the files it receives.

A terminal window titled "Client" with standard window controls. The prompt is root@ee6d411ea598:/compnets#. The user enters 'java -cp . Client'. The program outputs 'What file are you requesting?' and the user enters 'test.txt'. The program then outputs 'Requested test.txt' and a cursor is visible on the next line.

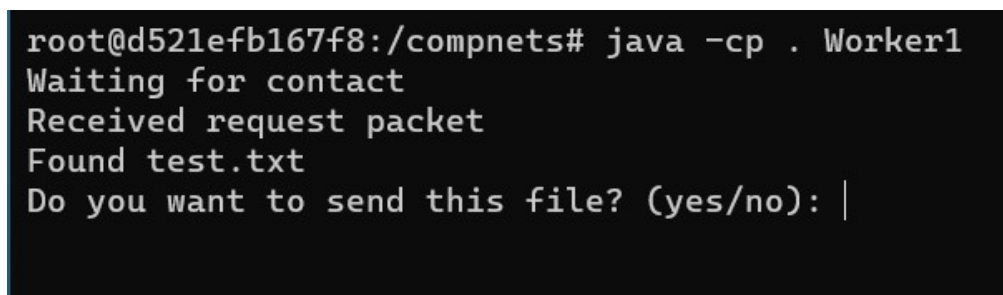
```
root@ee6d411ea598:/compnets# java -cp . Client
What file are you requesting? test.txt
Requested test.txt
|
```

Figure 5: The Client requests a file. Then the request packet is sent to the Ingress.

A terminal window titled 'Ingress' with standard window controls. The prompt is 'root@80654037f421:/compnets#'. The user enters 'java -cp . Ingress'. The output shows 'Waiting for contact', 'Received packet', and 'Forwarded request packet to Worker 1'.

```
root@80654037f421:/compnets# java -cp . Ingress
Waiting for contact
Received packet
Forwarded request packet to Worker 1
|
```

Figure 6: The Ingress receives the request packet and forwards it on to one of the workers. Here it uses Worker 1, but next time it receives a request, the Ingress will send it to Worker 2 instead.

A terminal window showing the prompt 'root@d521efb167f8:/compnets#'. The user enters 'java -cp . Worker1'. The output shows 'Waiting for contact', 'Received request packet', 'Found test.txt', and 'Do you want to send this file? (yes/no):'.

```
root@d521efb167f8:/compnets# java -cp . Worker1
Waiting for contact
Received request packet
Found test.txt
Do you want to send this file? (yes/no): |
```

Figure 7: Worker 1 receives the request, and after finding the file in it's directory, it asks the user if they would like this file to be returned.

```
root@d521efb167f8:/compnets# java -cp . Worker1
Waiting for contact
Received request packet
Found test.txt
Do you want to send this file? (yes/no): yes
Sending test.txt
File size: 80745
Sent Info-Packet w/ name & length of file
Sent FileContent Packet Number: 1
Sent FileContent Packet Number: 2
Sent FileContent Packet Number: 3
Sent FileContent Packet Number: 4
```

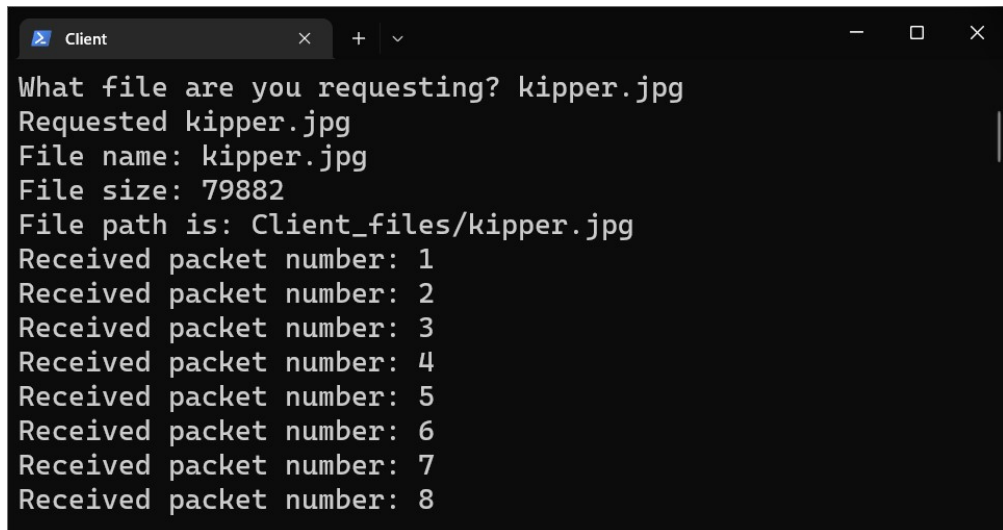
Figure 8: Worker 1 first sends a FileInfo packet with the name and size of the requested file to the Ingress. It then proceeds to send the FileContent packets to the Ingress.

```
Ingress
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
Received packet
Forwarded FileContent packet to Client
```

Figure 9: Ingress receives each packet from the Worker and forwards it to the Client.

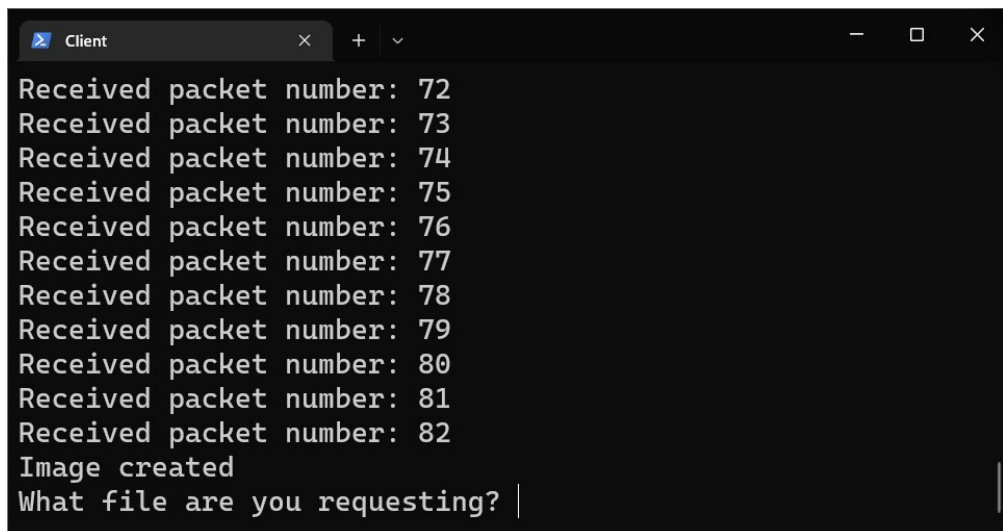
```
Client
Received packet number: 72
Received packet number: 73
Received packet number: 74
Received packet number: 75
Received packet number: 76
Received packet number: 77
Received packet number: 78
Received packet number: 79
Received packet number: 80
Received packet number: 81
Received packet number: 82
Received packet number: 83
What file are you requesting? |
```

Figure 10: The Client receives each packet and writes the data into the local file.

A terminal window titled 'Client' with a dark background and light green text. It shows a sequence of commands and responses for requesting a file named 'kipper.jpg'. The window includes standard OS window controls (minimize, maximize, close) in the top right corner.

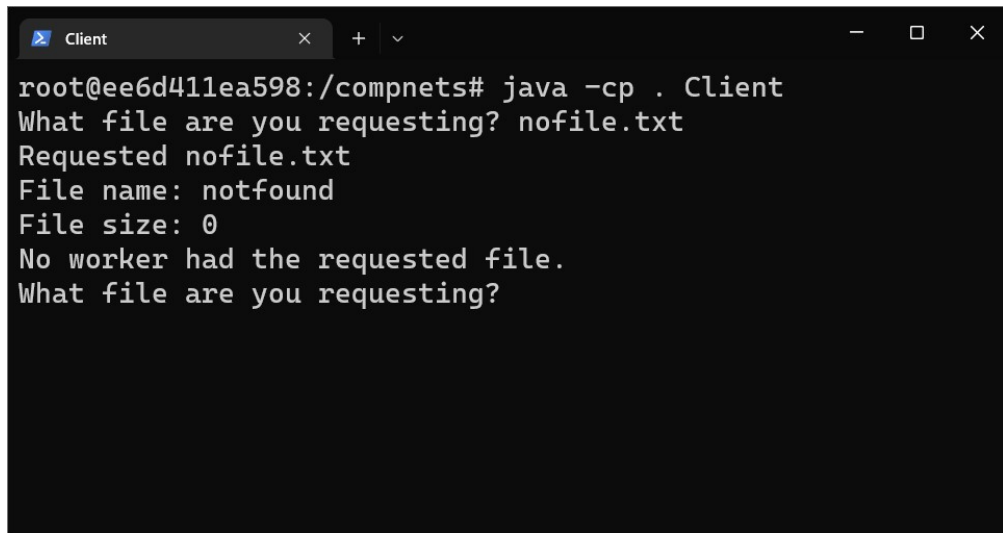
```
Client
What file are you requesting? kipper.jpg
Requested kipper.jpg
File name: kipper.jpg
File size: 79882
File path is: Client_files/kipper.jpg
Received packet number: 1
Received packet number: 2
Received packet number: 3
Received packet number: 4
Received packet number: 5
Received packet number: 6
Received packet number: 7
Received packet number: 8
```

Figure 11: We can also request an image.

A terminal window titled 'Client' with a dark background and light green text. It shows the final steps of the file request process, including receiving the last packets and creating the image file. The window includes standard OS window controls (minimize, maximize, close) in the top right corner.

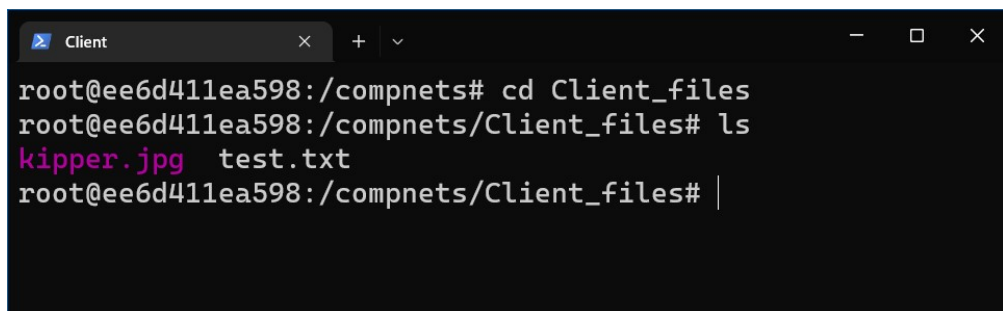
```
Client
Received packet number: 72
Received packet number: 73
Received packet number: 74
Received packet number: 75
Received packet number: 76
Received packet number: 77
Received packet number: 78
Received packet number: 79
Received packet number: 80
Received packet number: 81
Received packet number: 82
Image created
What file are you requesting? |
```

Figure 12: Once the Client has received all of the packets, it can join the parts of the image together and create the image file in it's directory.

A terminal window titled 'Client' with standard window controls. The prompt is 'root@ee6d411ea598:/compnets#'. The user enters 'java -cp . Client'. The program outputs: 'What file are you requesting? nofile.txt', 'Requested nofile.txt', 'File name: notfound', 'File size: 0', 'No worker had the requested file.', and 'What file are you requesting?'.

```
root@ee6d411ea598:/compnets# java -cp . Client
What file are you requesting? nofile.txt
Requested nofile.txt
File name: notfound
File size: 0
No worker had the requested file.
What file are you requesting?
```

Figure 13: If the Client requests a file that does not exist, the workers will return a FileInfo packet with the name "notfound".

A terminal window titled 'Client' with standard window controls. The prompt is 'root@ee6d411ea598:/compnets#'. The user enters 'cd Client\_files'. The prompt changes to 'root@ee6d411ea598:/compnets/Client\_files#'. The user enters 'ls'. The output is 'kipper.jpg test.txt'. The prompt returns to 'root@ee6d411ea598:/compnets/Client\_files# |'.

```
root@ee6d411ea598:/compnets# cd Client_files
root@ee6d411ea598:/compnets/Client_files# ls
kipper.jpg  test.txt
root@ee6d411ea598:/compnets/Client_files# |
```

Figure 14: We can see here that the Client files directory now contains the 2 files that we requested.



## 5 Summary

This report has described my attempt at a solution to address the transfer of files from multiple Workers to a Client through an Ingress. While the implementations of the protocol between the Client, Ingress and Workers is relatively simplistic, it demonstrates the how packets are sent from one node to another over a real network. At the beginning of this report I described the concepts and protocols that I implemented into my design of the system. I then showed some snippets of code that I used and explained it. Lastly, I demonstrated my working program, with screenshots of the terminal as it is executed.

## 6 Reflection

Over the course of completing this assignment, I learned a lot about how the network protocols work such as UDP and about how UDP datagrams are transmitted across networks. I learned how flow control works in case of an interruption in the data flow between nodes. Through my research for this assignment, I slowly started to piece together what I needed for this solution.

I now feel I know how a system like this would work in the real world and the changes I would make if I were implementing on a larger scale. Firstly, I would use a different flow control protocol other than Simplest that uses acknowledgements. This would be useful as if a packet does not make it to its destination for some reason, my system has no idea. I would also remove the ability for the user to stop the Worker from returning the requested file as it doesn't really make sense on a real network.

All in all, excluding the time taken to write the report, the assignment took me about 80 hours to complete.