



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

CSU33031 Computer Networks

Assignment #1: Protocols

Hannah Foley

Student ID: 20332137

October 28, 2022

1. Introduction	3
2. Theory of Topic	3
2.1 Internet Protocol	3
2.2 UDP	3
2.3 Ports and Sockets	3
2.4 Packets	3
2.4 Header and Payload	4
2.4 Network Traffic and Wireshark	4
3. Implementation	4
3.1 Topology	4
3.2 Node	5
3.3 Client	5
3.4 Ingress	7
3.5 Worker	9
3.6 Packets	10
3.6.1 Ack	10
3.6.2 FileInfoContent	11
3.6.3 RecFileInfo	11
3.7 Header	12
3.7 Payload	14
3.8 Traffic	15
4. Discussion	15
4.1 Summary	15
4.2 Reflection	16
5. References	16

1. Introduction

The aim of this assignment is to build a protocol that retrieves files. The protocol involves a number of actors. There is the client, who issues requests for a file and receives files. Then there is the ingress who receives the client's requests and forwards on the requests and who also forwards on the actual files. Lastly, there is the worker who retrieves the file and sends it onto the ingress.

We were required to design a protocol to carry out this functionality of forwarding files. My protocol is mostly built from the java example provided on Blackboard. I also made use of the `DatagramPacket`, `DatagramSocket` and `InetSocketAddress` java classes in my implementation.

2. Theory of Topic

2.1 *Internet Protocol*

Internet protocols are a set of rules that control the exchange of data over the internet. It does this by creating packets. A protocol defines the rules for how packets are organised and sent.

Internet protocol works using IP addresses. IP addresses are unique and universal addresses. They consist of a 32 bit number. The Client, Ingress and Workers each have their own IP addresses. These IP addresses are stored in the header of the packets and are used to identify where the packets are sent to.

2.2 *UDP*

One of the requirements for this assignment was to use UDP sockets and datagrams. UDP is one of the simplest protocols as it does not make use of any flow control and does not require a connection to be established before packets are sent.

2.3 *Ports and Sockets*

Datagram packets are sent and received using sockets. `InetSocketAddress`s are made up using ports and IP Addresses. The IP address corresponds to the system and the port number corresponds to the program where the data needs to be sent (Datta, 2021)

2.4 *Packets*

Datagram packets are containers of information passed between nodes. In my implementation, I used the java class `DatagramPacket` to create my packets. (Oracle)

2.4 Header and Payload

UDP headers typically have a fixed header of 8 bytes. There are 2 bytes each for the source and destination ports, 2 for the length and 2 for the checksum. This is followed by the data itself. I have described in my outline of my implementation how I added an extra byte to this header (Section 3.7)

2.4 Network Traffic and Wireshark

I captured the network traffic using tcpdump and viewed the resulting .pcap file in Wireshark. The network traffic contained all the packets that were sent back and forward between the components. Wireshark allowed me to have a visual representation of the packages.

3. Implementation

3.1 Topology

Each of the components was deployed in its own Docker container. The names of these containers were used as the hardcoded hostnames for the InetAddresses. All these containers were connected to the one subnet.

The protocol involves a number of actors. This is described in the diagram below.

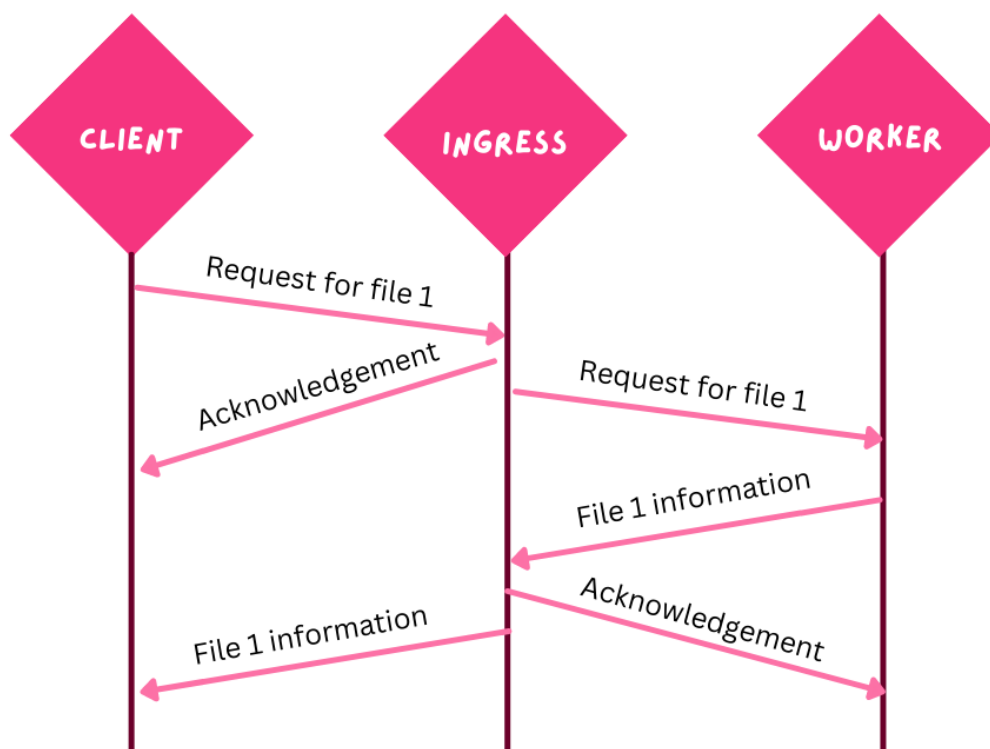


Figure 1: There are three node types; Client, Ingress and Worker. The Client sends requests for files onto the Ingress. The Ingress forwards this request to the Worker. The Worker retrieves this file and sends it to the Ingress. The Ingress forwards this file onto the Client who originally requested it. Note this figure only contains one Worker and Client for simplicity but it is possible to have more.

3.2 Node

The Node.java class is the base of all the node types. This class was provided for us on Blackboard. Each of the Client, Ingress and Worker extend and build on top of the Node class, adding their own unique functionalities.

There is one function that is not overridden in the extension classes, and this is the *run()* function. This function consists of an endless loop that waits and listens for packets. Once a packet is received, the *onReceipt(packet)* function is called. There are different implementations of *onReceipt(packet)* in each node extension class.

```

/*
 * Listen for incoming packets and inform receivers
 */
public void run() {
    try {
        latch.await();
        // Endless loop: attempt to receive packet, notify receivers, etc
        while(true) {
            DatagramPacket packet = new DatagramPacket(new byte[PACKETSIZE], PACKETSIZE);
            socket.receive(packet);

            onReceipt(packet);
        }
    } catch (Exception e) {if (!(e instanceof SocketException)) e.printStackTrace();}
}

```

Figure 2: run function, common to all node types. This function consists of an endless loop that waits and listens for packets. Once a packet is received, the onReceipt function is called.

3.3 Client

The first class that extends Node.java is Client.java. This class is built from the Client.java class provided on Blackboard. Client is the class that the user interacts with to pick which file they would like to issue a request for. Upon being run, the user is asked which file they would like to request, file 1 or file 2. Based on the user's choice, a file request packet is created and sent onto the Ingress.

```

root@4b950a26a58a: /compnets
Enter '1' to request file 1
Enter '2' to request file 2
1
File size: 29
Requesting packet w/ name: file1.txt & length: 29
Packet sent
ACK:OK - Received this
Program completed
Filename: file1.txt - Size: 29
root@4b950a26a58a: /compnets#

```

Figure 3: Terminal that is running the *Client.java* class. The first two lines are asking the user which file they would like to request

The destination address for the Client node is always set as the Ingress address. This is because the Client will only send and receive packets from the Ingress.

When the Client receives a packet, the type of packet is extracted. The type is used in a switch statement to determine what happens next with the packet.

Ack Packet (Acknowledgement)	The Ingress has received what the Client has sent Prints: "Received Ack packet"
GetFileInfo Packet (request for file)	The Client should never receive a request for a file Prints: "Error: Client has received request to get file"
RecFileInfo Packet (receipt of file)	The Client received the file requested. From here the <i>receivedFile(packet)</i> function is called Prints: "Received file"

Table 1: Describes what the Client does with each packet type it receives

The *receivedFile(DatagramPacket packet)* function prints the file content to the console.

```

root@4b950a26a58a: /compr
root@4b950a26a58a:/compnets# java -cp . Client
Enter '1' to request file 1
Enter '2' to request file 2
1
File size: 37
Requesting packet w/ name: file1.txt & length: 37
Making packet now with type 2
Making packet now with filename:1
Packet sent
Packet received
Received Ack packet
Packet received
Received File
Client has received the file - Thank you!!
File contents are as follows:
Hello World, lots of love from file 1

```

Figure 4: Terminal that is running the Client.java class. The content of file one is "Hello World, lots of love from file 1".

3.4 Ingress

The next class that extends Node.java is Ingress.java. The Ingress acts as a middleman. It routes traffic between nodes. Upon being run, the Ingress class waits to receive a packet.

The main functionalities of the Ingress class are to:

- Take the request for a file received from the Client and forward it onto the relevant Worker
- Receive file from the Worker and forward it onto the Client.

When the Ingress receives a packet, the type of packet is extracted. The type is used in a switch statement to determine what happens next with the packet.

Ack Packet (Acknowledgement)	The Worker or Client has received what the Ingress has sent Prints: "Received Ack packet"
GetFileInfo Packet (request for file)	The Ingress has received a request for a file from the Client. Here the <i>getFile(packet)</i> function is called. Prints: "Received request to get file"
RecFileInfo Packet (receipt of file)	The Ingress received the file requested. From here the <i>sendFile(packet)</i> function is called. Prints: "Received file"

Table 2: Describes what the Ingress does with each packet type it receives

The `sendFile(DatagramPacket packet)` function forwards the file to the Client.

```
public synchronized void sendFile(DatagramPacket packet) throws IOException, InterruptedException {
    DatagramPacket packetCopy = packet;

    packetCopy.setSocketAddress(clientAddress);
    socket.send(packetCopy);
    System.out.println("Ingress has sent file onto client ");
}
```

Figure 5: SendFile function. This creates a copy of the packet and forwards it to the Client

The `getFile(DatagramPacket packet)` function chooses a Worker to send the request for the file onto. The Ingress chooses which Worker to forward the request onto based on the file requested by the Client.

File 1	Worker 1
File 2	Worker 2

Table 3: Mapping of requested files to workers

```
public synchronized void getFile(DatagramPacket packet) throws IOException, InterruptedException {
    PacketContent content = PacketContent.fromDatagramPacket(packet);
    int fileno = content.getFilenum();
    InetSocketAddress workerToSendTo = workerAddress;
    if(fileno == 1){
        workerToSendTo = workerAddress;
        System.out.println("SENDING TO WORKER 1");
    }
    else if (fileno == 2){
        workerToSendTo = worker2Address;
        System.out.println("SENDING TO WORKER 2");
    }
    else{
        System.out.println("ERROR: DONT KNOW WHICH WORKER TO SEND TO");
    }

    System.out.println("Sending to worker " + workerToSendTo.getHostAddress());
    //...

    DatagramPacket packetCopy = packet;
    packetCopy.setSocketAddress(workerToSendTo);
    socket.send(packetCopy);
    System.out.println("Ingress has sent request for file onto worker ");
}
```

Figure 6: getFile function. This chooses a worker to send to based on the file being requested. Then a copy of the request from the Client is forwarded onto the relevant Worker.


```

root@08b293ce3827:/compnets# java -cp . Ingress
Waiting for contact
Received a packet
Making packet now with type 1
Making packet now with filename:0
Received request to get file
SENDING TO WORKER 1
Sending to worker part2worker
port: 50092
WORKER ADDRESS: part2worker/172.21.0.3
INGRESS ADDRESS: part2ingress/172.21.0.4
CLIENT ADDRESS: part2client/172.21.0.2
RESOLVED?: false
Ingress has sent request for file onto worker
Received a packet
Making packet now with type 1
Making packet now with filename:0
Received File
Ingress has sent file onto client
|

```

Figure 7: Terminal that is running the Ingress.java class

3.5 Worker

The final class that extends Node.java is Worker.java. This class is built from the Server.java class provided on Blackboard. The Worker receives requests for files from the Ingress and sends out packets with the file contents back. The destination address for the Worker node is always set as the Ingress address. This is because the Worker will only send and receive packets from the Ingress. Upon being run, the Worker class waits to receive a packet from the Ingress.

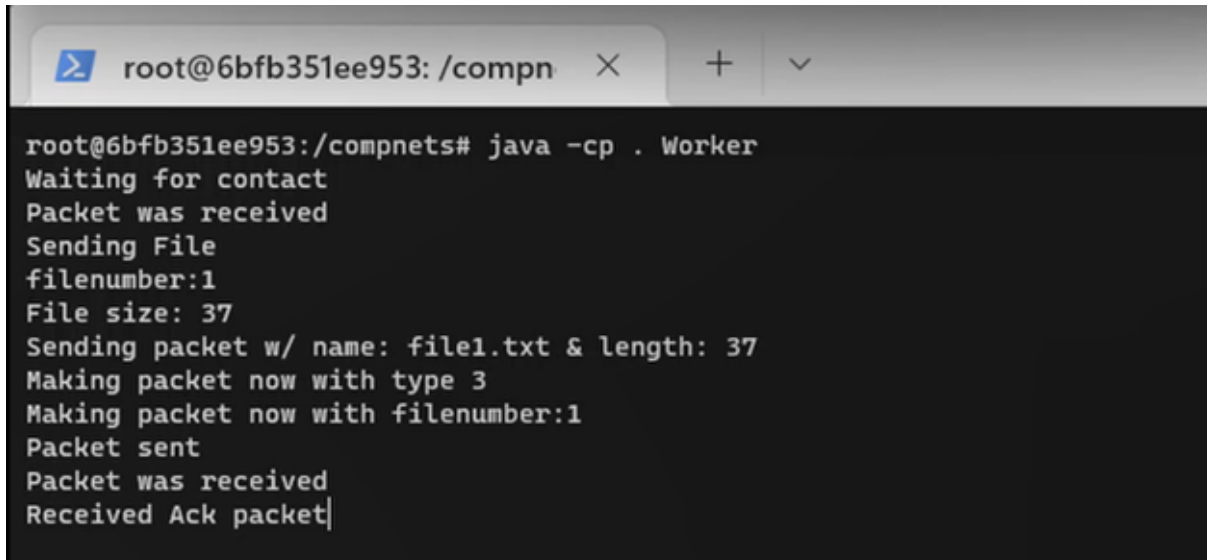
When the Worker receives a packet, the type of packet is extracted. The type is used in a switch statement to determine what happens next with the packet.

Ack Packet (Acknowledgement)	The Ingress has received what the Worker has sent Prints: "Received Ack packet"
GetFileInfo Packet (request for file)	The Worker has received a request for a file from the Ingress. Here the <i>sendFile(packet)</i> function is called. Prints: "Sending File"
RecFileInfo Packet (receipt of file)	The Worker should never receive a packet containing file info Prints: "Error: Wrong packet reached worker"

Table 4: Describes what the Worker does with each packet type it receives

The `sendFile(DatagramPacket packet)` function creates a new instance of `RecFileInfo Packet`. This packet is sent onto the Ingress.

To implement multiple workers. I made a second Worker class, `Worker2.java` which extends from `Worker`. It has all the same functionalities except it is made using a different port number.



```

root@6bfb351ee953: /compn
root@6bfb351ee953:/compnets# java -cp . Worker
Waiting for contact
Packet was received
Sending File
filename:1
File size: 37
Sending packet w/ name: file1.txt & length: 37
Making packet now with type 3
Making packet now with filename:1
Packet sent
Packet was received
Received Ack packet

```

Figure 8: Terminal that is running the `Worker.java` class.

3.6 Packets

The `PacketContent.java` class is the base class for the three types of packets. This class was provided for us on Blackboard. The `PacketContent` class is used as a way to extract the content, including the header and the data payload from the `DatagramPackets`. It is also used as a way of making `DatagramPackets` from the files.

The `fromDatagramPacket(DatagramPacket packet)` function extracts the information from a `DatagramPacket`. It does this by reading in a byte array and parsing it. The first element it extracts from the byte array is the type of packet. This will determine how the remainder of the content is parsed. The second element extracted is the relevant file number for the packet. This was something I added myself (Section 3.7). After that, the content is created in the individual packet classes.

The `toDatagramPacket(DatagramPacket packet)` function creates a `DatagramPacket`. It consists of a byte array which contains the type of packet, header and data payload.

3.6.1 Ack

The first class to extend `PacketContent.java` is `AckPacketContent.java`. This is the class for packet content that represents acknowledgements.

3.6.2 FileInfoContent

The next class to extend PacketContent.java is FileInfoContent.java. This is the class for packet content that represents requests for files.

3.6.3 RecFileInfo

The next class to extend PacketContent.java is ReceiveFileInfo.java. This is the class for packet content that represents file information.

This packet type contains one variable that the others don't, being fileMessage. fileMessage contains in a String, the file contents. The contents of the files are only accessed here as only the Workers are supposed to have access to this information.

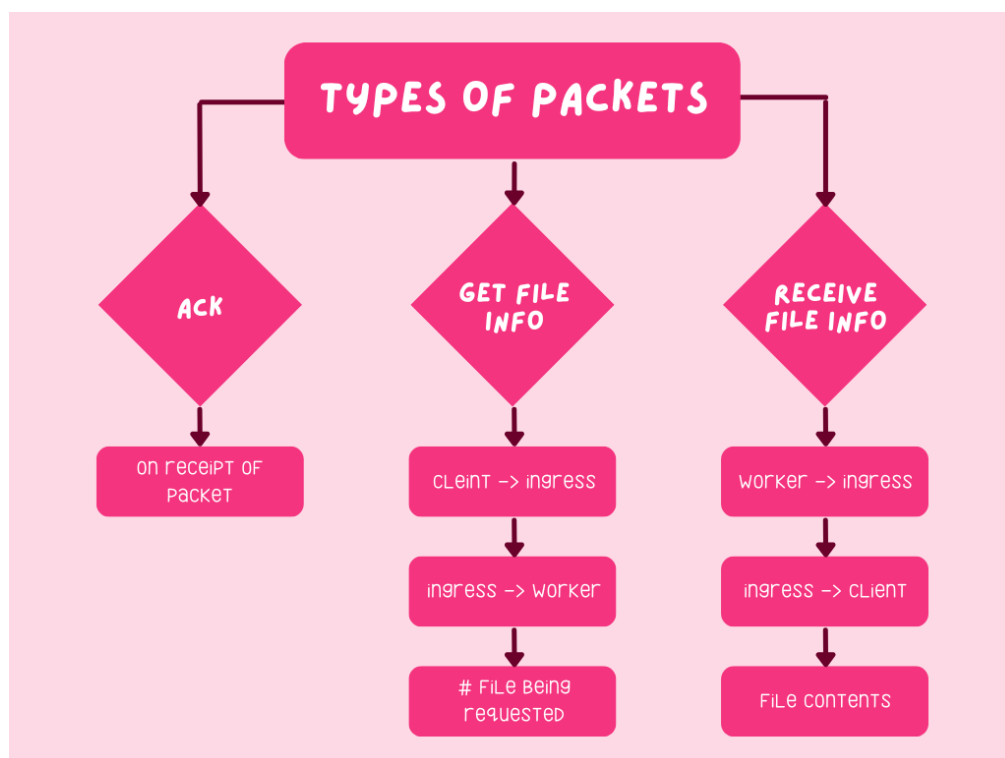


Figure 9: Summary of packet types.

3.7 Header

UDP headers typically have a fixed header of 8 bytes. There are 2 bytes each for the source and destination ports, 2 for the length and 2 for the checksum. This is followed by the data itself.

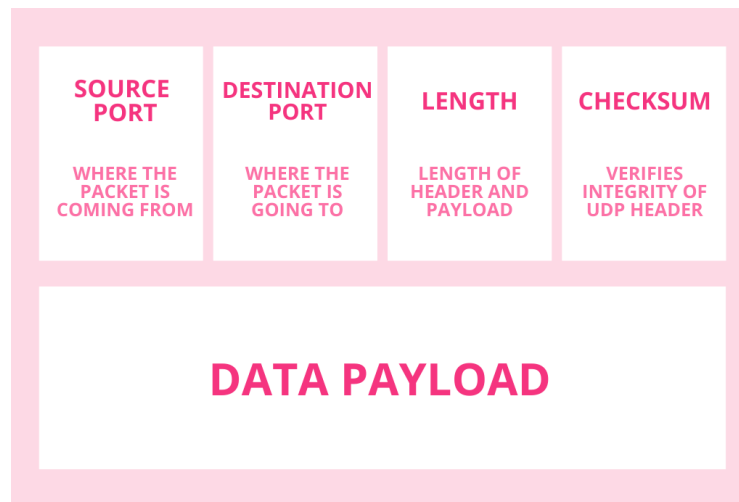


Figure 10: Diagram describing a UDP header

This is a very basic implementation of a header, and I wanted to add some functionality to my header. The extra functionality I added to my header was that it would have a byte in which the relevant file number for the packet could be found.

In order to do this, I made a byte array called header. To this, I wrote in the file number. Then I made the data payload byte array as normal. These two byte arrays were then concatenated together. Then, when making my datagram packet, I used an alternative constructor with an offset. The offset would tell the receiver of the packet where the header would end and the data payload would start.

```
byte[] header = makeHeader();
data= bout.toByteArray(); // convert content to byte array

//concat these byte arrays header + data
ByteArrayOutputStream outputStream = new ByteArrayOutputStream( );
outputStream.write(header);
outputStream.write(data);

byte[] packetByteArray = outputStream.toByteArray( );

packet = new DatagramPacket(packetByteArray, OFFSET, data.length);
oout.close();
bout.close();
```

Figure 11: Code snippet which shows the concatenation of the two byte arrays and the creation of the Datagram Packet using the alternative constructor

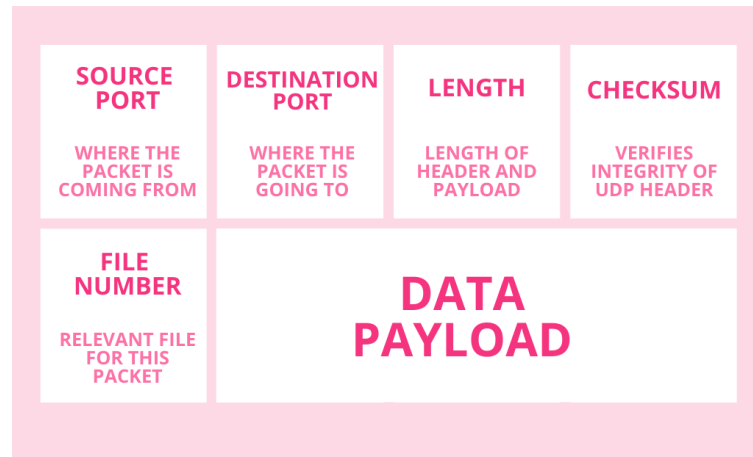


Figure 12: Diagram describing my new implementation of the header. There is a new byte in the header which represents the relevant file number.

```
DatagramPacket(byte[ ] data, int offset, int size):
data : byte array
offset : length into the array that the data payload starts
length : length of message to deliver

It allows you to specify an offset into the buffer at which data will
be stored.
```

Figure 13: Describes the alternative constructor that I used when making the packets

I used Wireshark to ensure the contents of my header was visible.

The key here was just to try my hand at creating my own header. I wanted it to be human readable and easy to understand what the number was referring to when seen on the Wireshark display. It is important in header design that there is a balance between the overhead space needed for header and the functionality it provides.

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.21.0.2	172.21.0.3	UDP	74	50091 → 50090 Len=32
2	0.024811	172.21.0.3	172.21.0.2	UDP	84	50090 → 50091 Len=42
3	0.037168	172.21.0.3	172.21.0.4	UDP	74	50090 → 50092 Len=32
4	0.037257	172.21.0.3	172.21.0.4	UDP	74	50090 → 50092 Len=32
5	0.042162	172.21.0.4	172.21.0.3	UDP	67	50092 → 50090 Len=25
5	0.043192	172.21.0.3	172.21.0.4	UDP	84	50090 → 50092 Len=42
7	0.050770	172.21.0.3	172.21.0.2	UDP	67	50090 → 50091 Len=25
8	5.025725	02:42:ac:15:00:03	02:42:ac:15:00:02	ARP	42	Who has 172.21.0.2? Tell 172.21.0.3
9	5.025788	02:42:ac:15:00:02	02:42:ac:15:00:03	ARP	42	Who has 172.21.0.3? Tell 172.21.0.2
9	5.025808	02:42:ac:15:00:03	02:42:ac:15:00:02	ARP	42	172.21.0.3 is at 02:42:ac:15:00:03

> Ethernet II, Src: 02:42:ac:15:00:03 (02:42:ac:15:00:03), Dst: 02:42:ac:15:00:04 (02:42:ac:15:00:04)						
0000	02 42 ac 15 00 04	02 42 ac 15 00 03	08 00 45 00	·B·	·B·	·E·
0010	00 3c e0 19 40 00	00 11 02 66 ac 15 00	03 ac 15	<·@·@·	·f·	·
0020	00 04 c3 aa c3 ac	00 28 58 6b ac ed	00 05 77 1a	·	(Xk·	·w·
0030	00 00 00 02 00 00	00 01 31 00 09 66	69 6c	·	·1·	·fil
0040	65 31 2e 74 78 74	00 00 00 25		e1.txt·	·%	

Figure 14: A screenshot from the network traffic through the Ingress node when file 1 was asked for. Here you can see '1' in the relevant file byte position in the header before the payload.

No.	Time	Source	Destination	Protocol	Length	Info
8	5.037101	02:42:ac:15:00:03	02:42:ac:15:00:05	ARP	42	Who has 172.21.0.5? Tell 172.21.0.3
2	0.022256	172.21.0.3	172.21.0.2	UDP	84	50090 → 50091 Len=42
7	0.045260	172.21.0.3	172.21.0.2	UDP	67	50090 → 50091 Len=25
1	0.000000	172.21.0.2	172.21.0.3	UDP	74	50091 → 50090 Len=32
5	0.040710	172.21.0.5	172.21.0.3	UDP	67	50093 → 50090 Len=25
3	0.031903	172.21.0.3	172.21.0.5	UDP	74	50090 → 50093 Len=32
4	0.031988	172.21.0.3	172.21.0.5	UDP	74	50090 → 50093 Len=32
6	0.041851	172.21.0.3	172.21.0.5	UDP	84	50090 → 50093 Len=42

Data: aced0005771a0000000200000002000132000966696c65322e74787400000037						
0000	02 42 ac 15 00 05	02 42 ac 15 00 03	08 00 45 00	·B·	·B·	·E·
0010	00 3c 5a e4 40 00	00 11 87 9a ac 15 00	03 ac 15	<Z·@·@·	·	·
0020	00 05 c3 aa c3 ad	00 28 58 6c ac ed	00 05 77 1a	·	(Xl·	·w·
0030	00 00 00 02 00 00	00 02 32 00 09 66	69 6c	·	·2·	·fil
0040	65 32 2e 74 78 74	00 00 00 37		e2.txt·	·7	

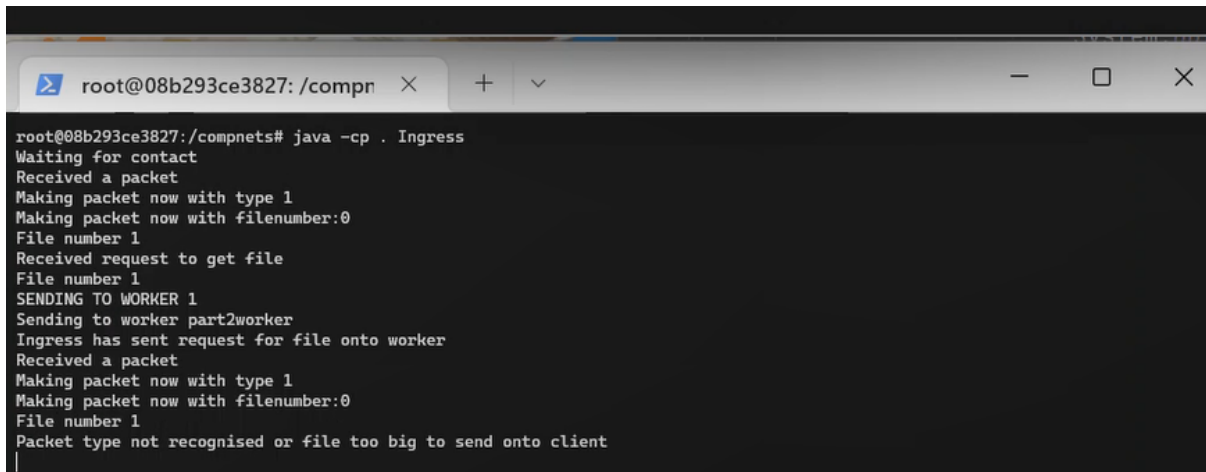
Figure 15: A screenshot from the network traffic through the Ingress node when file 2 was asked for. Here you can see '2' in the relevant file byte position in the header before the payload.

3.7 Payload

The maximum transmission unit (MTU) of a UDP Datagram Packet is 65535 bits. My protocol cannot work with data larger than this. Other protocols such as TCP can split a very large piece of data between different packets. The *Don't Fragment* flag in a TCP header allows for this functionality. This complexity was not necessary for this assignment.

Instead, I have designed my protocol so that if the Ingress receives a file which is bigger than the MTU, it prints to the console that the file is too big. This file is not sent on to the Client in this circumstance.

My reasoning for the Ingress being the node to stop the forwarding here is because the Ingress controls the traffic flow between nodes.



```

root@08b293ce3827:/comprn X + v - □ X
root@08b293ce3827:/comprn# java -cp . Ingress
Waiting for contact
Received a packet
Making packet now with type 1
Making packet now with filename:0
File number 1
Received request to get file
File number 1
SENDING TO WORKER 1
Sending to worker part2worker
Ingress has sent request for file onto worker
Received a packet
Making packet now with type 1
Making packet now with filename:0
File number 1
Packet type not recognised or file too big to send onto client

```

Figure 16: A screenshot from the Ingress terminal when the file being requested is too big. The final line reads; "Packet type not recognised or file too big to send onto client".

3.8 Traffic

As mentioned, I viewed the network traffic using Wireshark.

Wireshark allowed me to view the transport of packets between nodes in the network in a chronological order. The information provided on Wireshark told me which port packets were coming from and which port they were heading to.

With Wireshark I could see all the packets from one conversation between a group of nodes in one place. I mostly made use of this tool when implementing my own custom header,

4. Discussion

4.1 Summary

The purpose of this assignment was to 'learn about protocol development and the information that is kept in a header to support functionality of a protocol'. This assignment allowed me to get a hands-on learning experience of how protocols work.

While built primarily from the Java example provided for us on Blackboard, I have implemented some of my own node and packet types in order to meet the requirements of this assignment. This includes the Worker and Ingress nodes along with the RecFileInfo Packet type.

I also gained skills in using Docker and Wireshark. Docker was used to create containers that mimicked the network. Wireshark was used to view the captures of the network traffic.

4.2 Reflection

Upon completion of this assignment, I think I have a better grasp of the topic. However, this did not come without a lot of time and energy put into the assignment. I found it to be very time consuming, as it would take a while for the information to click.

I would have liked if the lecture content had correlated more with the assignment material as I found as the weeks passed, the assignment became more and more far removed from the lecture and tutorial content, making the assignment work feel like work for a separate module.

The time set for the assignments was very fair. If I had more time I would have liked to implement other features. In my initial video, I spoke about potentially including in the header content from Client to Ingress if the Client was authorised, and thus, allowed to access the files being requested or not. I would have implemented this in the same manner as I implemented the file type byte in the header.

5. References

Datta, Subham. "The Difference Between a Port and a Socket." *Baeldung*, 5 June 2021,

<https://www.baeldung.com/cs/port-vs-socket>. Accessed 26 October 2022.

Oracle. "DatagramPacket (Java SE 17 & JDK 17)." *Oracle Help Center*,

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/DatagramPacket.html>. Accessed 26 October 2022.

Oracle. "DatagramSocket (Java Platform SE 7)." *Oracle Help Center*,

<https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html>. Accessed 26 October 2022.

Oracle. "InetAddress (Java Platform SE 7)." *Oracle Help Center*,

<https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html>. Accessed 26 October 2022.