



CSU33031 Computer Networks

Assignment #1: File Transfer Protocol

Niall Sauvage, 20334203

June 20, 2023

Contents

1	Introduction	2
2	Overall Design	2
2.1	Stop and Wait	2
2.2	Initial Topology	2
2.3	Communication	3
3	Implementation	3
3.1	Physical Topology	3
3.2	Packet Encoding and Description	4
3.3	User Interaction	5
3.4	Synchronisation / Error Correction	6
3.5	Technical Details	6
4	Discussion and Development	8
4.1	Development Process	8
4.1.1	First Version	8
4.1.2	Second Version	9
4.2	Bugs and Workarounds	10
4.3	Registration	11
5	Summary	11
6	Reflection	11
6.1	Choice of Programming Language	11
6.2	Docker Components	11
6.3	Choice of protocol	11
6.4	Hours Spent and Practices	12
7	Appendix	13
7.1	onreceipt Methods	13
7.1.1	Client	13
7.1.2	Worker	15
7.1.3	Ingress (Server)	17

1 Introduction

This assignment contains the following description: “A client issues requests for files to an ingress node and receives replies from this node. The ingress node processes requests, forwards them to one of the workers that are associated with it, and forwards replies to clients that have send them.” My solution to this problem is based on a simple Stop-And-Wait protocol which allows the operation of the nodes as described above.

2 Overall Design

2.1 Stop and Wait

Stop-and-Wait is a simple flow control protocol in which the sender first sends some data to the receiver, then the receiver responds with an acknowledgement, commonly known as an ACK. The sender is waiting for this ACK, and when received, prompts the sender to reply with another packet of data. It is important to note that this is a flow control protocol and not an error control protocol: It cannot deal with the consequences of a packet not being received. That is to say, it is a protocol which should be implemented in a noiseless channel, such as a docker network. Moreover, as it is a noiseless channel, I did not implement common error correction schemes such as parity bits, CRC or Hamming Codes, although there is some error correction implemented using bytes delivered.

Stop and Wait is usually dismissed as a protocol due to its simplicity, however, in this case, I do not believe it mistake to implement it. This is because, as previously discussed, the network between Docker containers is a noiseless channel and thus requires no error correction. Moreover, due to its simplicity, Stop and Wait is an extremely lightweight protocol, which minimises processing time. Finally, a Docker network has a very low latency, so while only one packet can be in the pipeline being sent at any one time, this will not affect the throughput as much as if it was on a high-latency network.

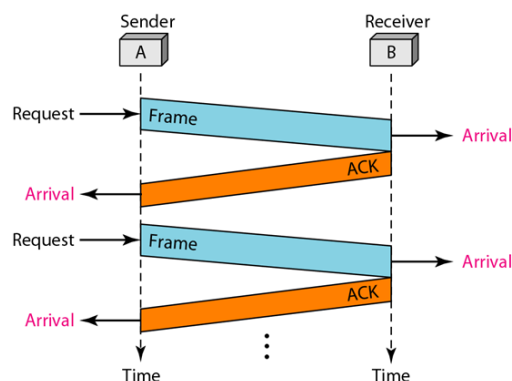


Figure 1: The image above is courtesy of B. Forouzan and found in the lecture slides. It shows us the operation of a Stop-And-Wait protocol between a Sender and a Reciever

2.2 Initial Topology

The topology mentioned in the assignment brief contains a Client, an Ingress and one or more Workers. The Client is connected solely to Ingress and the Ingress itself is connected to both the Client and all N Workers. This meant that two networks were necessary: The Client and Ingress required a network and the Ingress and Workers required a network.

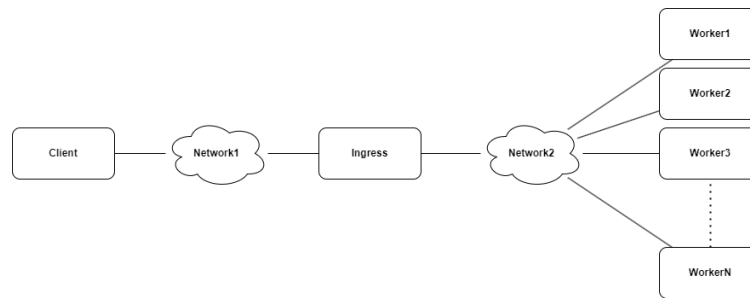


Figure 2: The figure above is a general overview of the topology specified in the assignment brief.

2.3 Communication

We can describe the process that must occur for file transfer to complete successfully using this protocol:

1. The Client will send a request for a file to Ingress.
2. Ingress will distribute the request to all workers.
3. One of the workers will be selected to transfer the file via Ingress.
4. The worker will begin transferring a packetised amount of the file to Ingress.
5. Ingress will forward this packet to the Client.
6. The Client will send an acknowledgement to Ingress.
7. Ingress will forward this acknowledgement to the selected Worker.
8. Repeat until file is transferred in its entirety.

3 Implementation

This section describes the physical implementation of the protocol sketched out above. The implementation was written in Java, with the individual nodes being Docker containers. The protocol is written on the back of UDP, using such programming concepts as Datagram Sockets.

3.1 Physical Topology

To create the topology described in section 2.2, I used Docker to create 5 containers based off the same image:

1. Client
2. Ingress
3. Three Workers

I have also created two networks using docker, csnet and workernet. The Client is connected to the Ingress (Server) via csnet, which has the subnet 172.20.0.0/16. Furthermore, the workers are connected to the Ingress (Server) via workernet, which has the subnet 172.21.0.0/16.

The Client will have port 50000, the Ingress (Server) will have port 50001 and the Workers will ask the user what ports they wish to bind them to, though I have implemented the convention of starting at 50002 and incrementing by 1 for each worker.

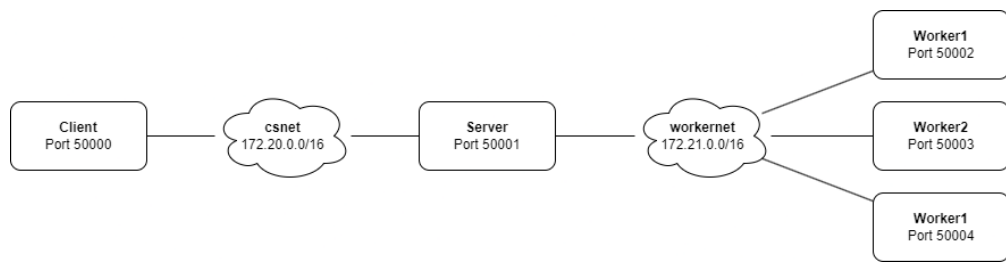


Figure 3: The figure above is a general overview of the topology implemented using docker.

3.2 Packet Encoding and Description

To implement my protocol, I had to create several types of packet:

AckPacketContent or ACKPACKET: is sent by the Client to Ingress upon receipt of a FileContent packet, which is then forwarded to the relevant worker. Contains a String and an Int.

FileContent or FILESEND: is responsible for sending the data contained in the file as a byte array. This will be sent from the Worker to the Ingress and from the Ingress to the Client. Contains a byte array.

FileInfoContent or FILEINFO: is responsible for sending the size of the file in bytes. This will be sent from the Worker to the Ingress and from the Ingress to the Client. Contains an int.

GetFileContent or GETFILE: is the packet which contains the name of the file the Client is requesting. This will be forwarded by the Ingress upon receipt from the Client to the Workers. Contains a String.

NoFile or NOFILE: is the packet which is sent from the Worker to Ingress if the Worker does not have the requested file. Does not contain content.

Register or REGISTER: is sent by the Worker upon startup to Ingress such that Ingress can add it to its list of active Worker ports. Does not contain content.

The packets sent in normal operation is thus as follows:

1. The workers, upon startup, will send a Register packet to Ingress.
2. The Client will send a GetFileContent packet to Ingress.
3. Ingress will distribute the GetFileContent packet to all workers.
4. The workers will send back either a FileInfoContent packet or a NoFile packet, dependent on whether or not they have the file in their directory.
5. Ingress will forward the first FileInfoContent packet to the Client. If all workers respond with a NoFile packet, this is forwarded to the Client.
6. The first worker to respond with a FileInfoContent packet will be selected by Ingress to transfer the file.
7. The Client will respond with an AckPacketContent upon receipt of the FileInfoContent.
8. Ingress will forward this AckPacketContent to the selected Worker.
9. The worker will begin transferring a FileContent packet to Ingress, containing some or all of the file.
10. Ingress will forward this packet to the Client.
11. The Client will respond with an AckPacketContent upon receipt of the FileContent.
12. Ingress will forward this AckPacketContent to the selected Worker.
13. Repeat from step 8 until file is transferred in its entirety.

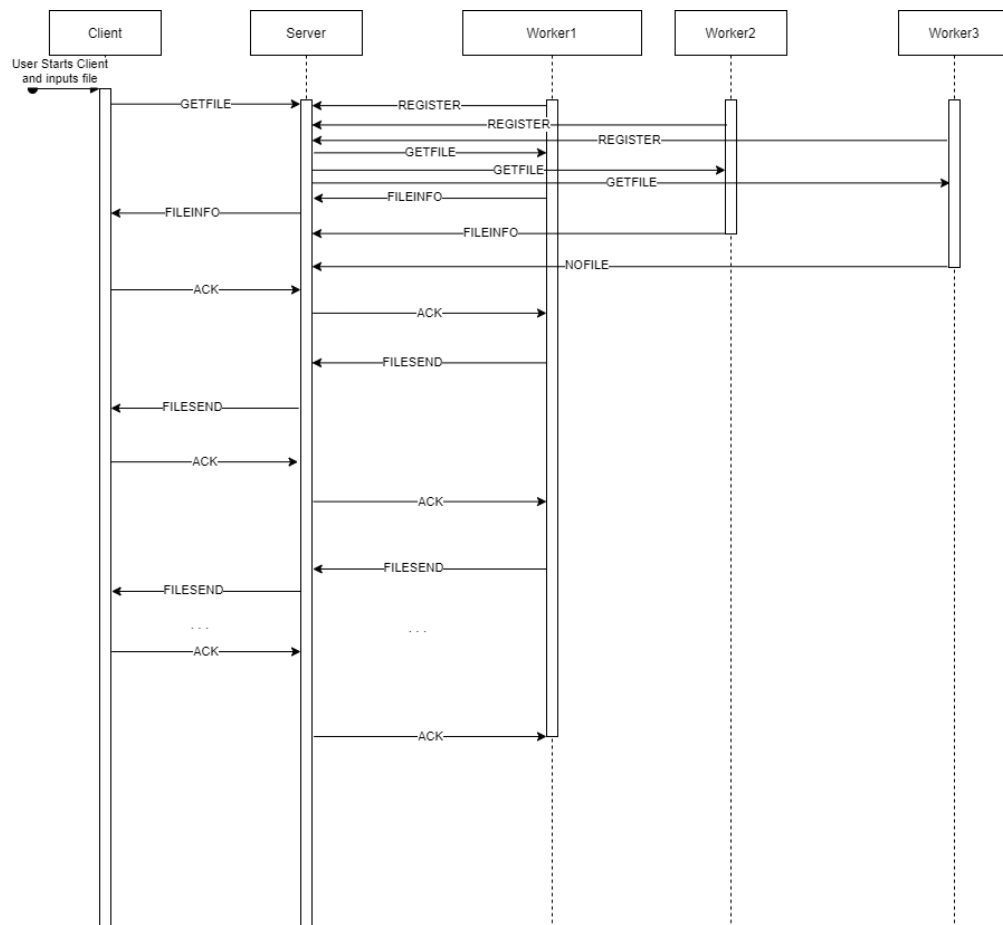


Figure 4: This figure demonstrates how the protocol would proceed if only Worker1 and Worker2 have the file.

3.3 User Interaction

```

root@62139d4d6d61:/compnets/Client# java -cp . Client
Name of file: 

```

Figure 5: This figure demonstrates how a user can use the client to request a file.

The user will first start the client, prompting the client to ask the user which file they wish to request. The protocol will then run, and attempt to retrieve the file for the client. Should the protocol be unsuccessful, the client will notify the user that the file could not be retrieved.

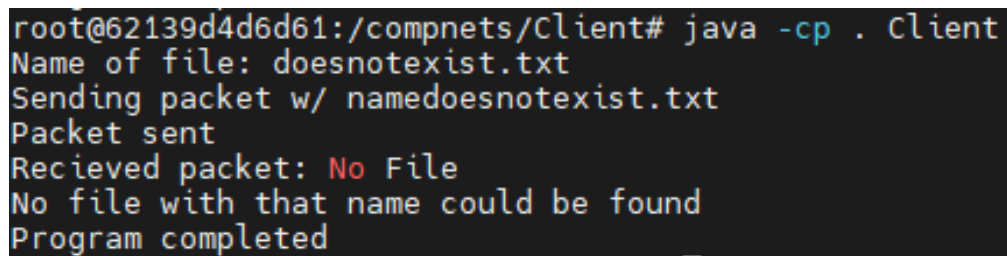
To avoid multiple files with hardcoded ports, the Workers will instead ask for a port to be assigned by the user on startup. This will allow any N workers to be active at any one time, subject to the fact they have different ports.

```

try {
    System.out.println("Please enter the desired port number:");
    int portNum= Integer.parseInt(System.console().readLine());
    (new Worker(portNum)).start();
    System.out.println("Program completed");
} catch (java.lang.Exception e) {e.printStackTrace();}

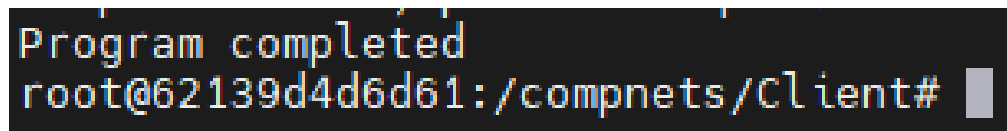
```

Listing 1: The worker asks the user to input a port for it to be assigned to.



```
root@62139d4d6d61:/compnets/Client# java -cp . Client
Name of file: doesnotexist.txt
Sending packet w/ namedoesnotexist.txt
Packet sent
Recieved packet: No File
No file with that name could be found
Program completed
```

Figure 6: This figure demonstrates how the Client will behave if a requested file could not be found.



```
Program completed
root@62139d4d6d61:/compnets/Client#
```

Figure 7: This figure demonstrates how the Client will behave if a requested file was found and transferred.

3.4 Synchronisation / Error Correction

To aid synchronisation, the ACKPACKET packet type will not only have a string containing “Ok - Received this”, but it will also contain an integer. This integer is how many bytes the Client received in the last packet. This means that should the packet be improperly created or an error occur during transmission such that parts get cut off, the entire file will be successfully transferred nonetheless. This is because the Worker will increase the value of the pointer into its array of bytes by the amount the ACKPACKET contains.

3.5 Technical Details

My implementation is based off the Java example given on Blackboard, as such, it makes liberal use of `ObjectInputStreams` and `ObjectOutputStreams`. All packet classes are subclasses of the `PacketContent` class, which contains a method, `fromDatagramPacket`, which is used to construct a subclass of `PacketContent` using an `ObjectInputStream` and the `Datagram Packet`. Likewise, there is a `toDatagramPacket` method, which uses an `ObjectOutputStream` to turn any `PacketContent` subclass into a byte array, and from there into a `DatagramPacket`, which can be sent to another machine.

The Ingress (Server) implementation largely just forwards packets based on their subclass, represented by the integer “type”. The value of type decides the address of the packet. For example, ACKPACKETs are forwarded to the selected Worker and FILESEND is forwarded to Client, amongst others. The exceptions to this rule are as follows:

1. The server keeps a record of how many workers do not have the requested file, if all do not, then it will forward the last NOFILE packet to the Client.
2. Upon receipt of a GETFILE packet, the Ingress will copy the packet and send to all workers that have registered with it.
3. Upon receipt of a REGISTER packet, the Ingress will add the `InetSocketAddress` of the worker which sent that packet to an arraylist, thus registering it.

The Client implementation will receive one of the following packet types and respond as follows:

FILEINFO: States the size of the file in bytes, at which point the Client will create a byte array of that size and respond with an ACKPACKET with the size 0, indicating that the Worker should begin transfer from the start.

FILESEND: Will be written to the Client's byte array. If the amount of bytes recieved equals the size of the file in bytes, the Client will attempt to write this byte array to a file with the name requested. It will then respond with an ACKPACKET with the size of the last packet recieved.

NOFILE: Will cause the client to reset, setting fname (file name), the byte array fileData and the File reqFile equal to null. It will also print to the console a message stating that no file of that description could be found.

The Worker implementation will receive one of the following packet types and respond as follows:

GETFILE: Causes the worker to attempt to load a file with the filename specified in the GETFILE packet from its local directory. Should it fail to do so, it will send a NOFILE packet to Ingress, otherwise it will respond with a FILEINFO packet, which contains the size of the file in bytes.

ACKPACKET: Causes the worker to add the integer in the ACKPACKET to its pointer into its byte array and then load a new FILESEND packet with data if all data has not been transmitted previously. This FILESEND packet is then sent to Ingress.

4 Discussion and Development

4.1 Development Process

As discussed above, my implementation is based off the Java Example given on Blackboard, which sent the file size and file name of a file stored on a “Server” to the “Client” when the Client entered that filename. I soon came to the conclusion that if I was to send a text file, then I should represent the contents of that text file as a String; However, if the file was an image, a PNG file, for example, then this could cause issues. This led me down the road of using a byte array to represent the contents of the file. This allowed it to be almost polymorphic: All files are ultimately represented by bytes, so with this protocol we could transfer any file.

4.1.1 First Version

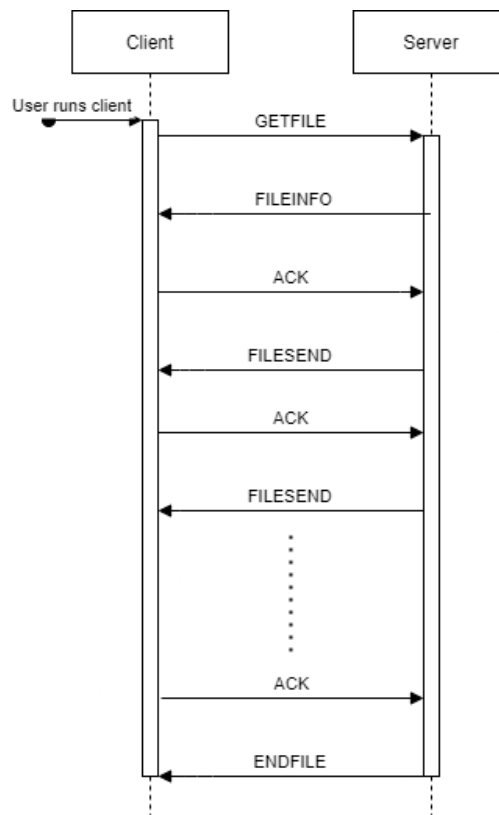


Figure 8: The figure above is a general overview of the first iteration of my protocol which did not have Workers.

Above, we can see a simplified first version of my protocol - it is also prescient to note the ENDFILE packet type. This was persistent across the first two iterations of my protocol: It was sent by the Ingress (Server) or Worker once it ran out of data to transmit. Upon receipt of an ENDFILE packet type, the Client would attempt to write the bytes out to a new file.

4.1.2 Second Version

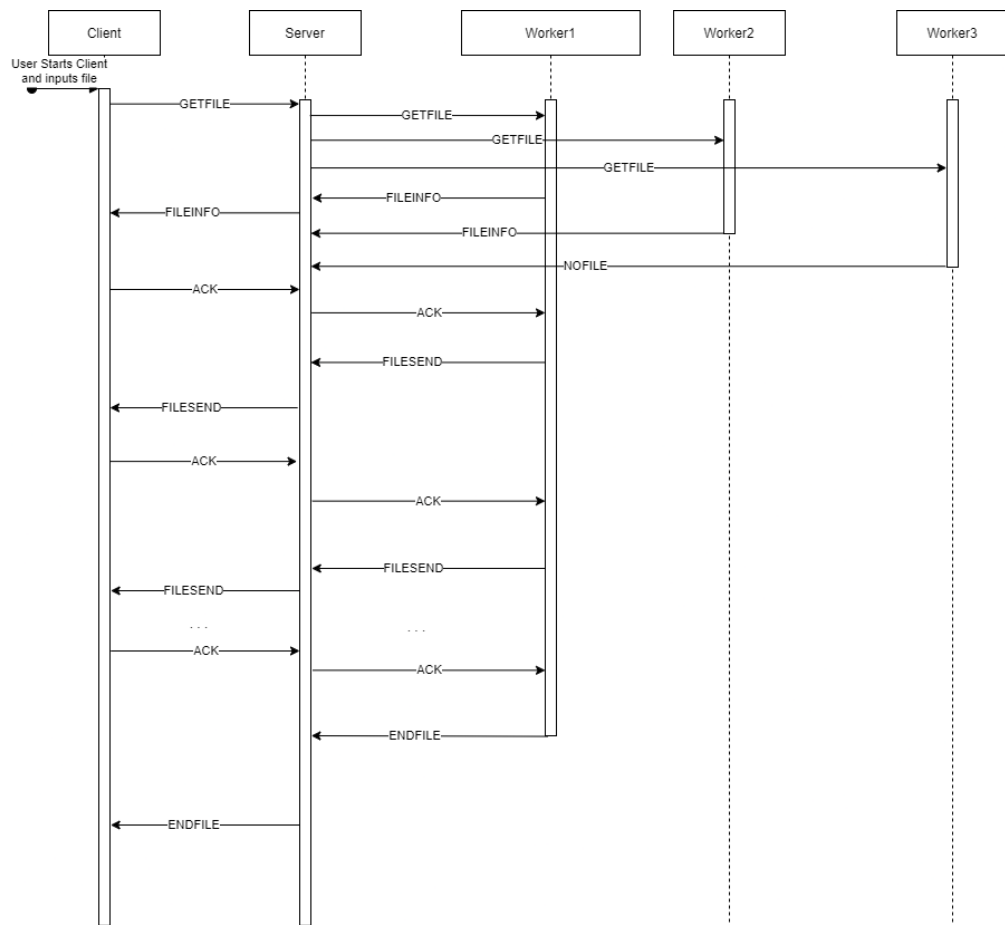


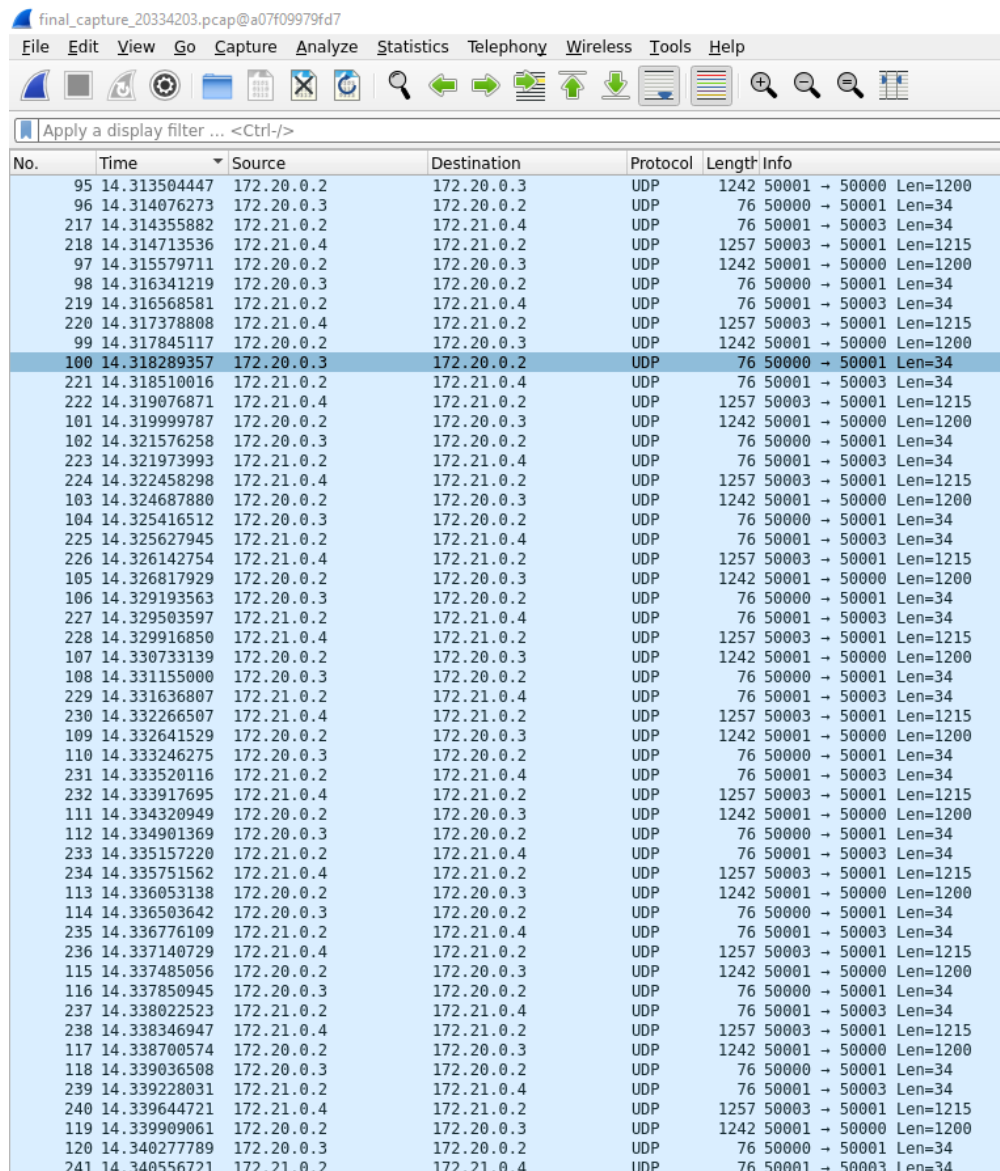
Figure 9: The figure above is a general overview of the second iteration of my protocol which implemented hardcoded Workers without registration.

We can see the evolution of my protocol above. Workers have been introduced, albeit with hardcoded sockets on the Server (Ingress) side and no registration. Additionally, ENDFILE is still a part of the protocol. The final flow diagram is available as Figure 4 above. Workers now register with the Ingress upon startup with the REGISTER packet type and as the Client keeps a count of bytes, it is no longer necessary to send an ENDFILE packet, as this can be done Clientside.

4.2 Bugs and Workarounds

In my process of developing this solution, I came across a particularly persistent bug: When I attempted to send a FILEINFO packet, the final few bytes would be cut off. I tested with various different sizes, for example, if I had PACKETSIZE set to 1024, only 1011 bytes of useful information would actually get through, although, quizzically, wireshark would detect the packet size as 1037, 13 bytes over instead of 13 bytes less.

I attempted several solutions to this, including rebuilding the docker image and recreating the docker containers, doing the same on a different PC, analysing the traffic using wireshark, using a text compare site to compare the data sent and the data received as well as printing the packet size received in the client. I also attempted to debug my protocol using loopback, however, I could not run the code locally to debug it as I would get exceptions.



No.	Time	Source	Destination	Protocol	Length	Info
95	14.313504447	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
96	14.314076273	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
217	14.314355882	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
218	14.314713536	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
97	14.315579711	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
98	14.316341219	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
219	14.316568581	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
220	14.317378808	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
99	14.317845117	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
100	14.318289357	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
221	14.318510016	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
222	14.319076871	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
101	14.31999787	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
102	14.321576258	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
223	14.321973993	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
224	14.322458298	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
103	14.324687880	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
104	14.325416512	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
225	14.325627945	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
226	14.326142754	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
105	14.326817929	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
106	14.329193563	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
227	14.329503597	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
228	14.329916850	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
107	14.330733139	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
108	14.331155000	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
229	14.331636807	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
230	14.332266507	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
109	14.332641529	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
110	14.333246275	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
231	14.333520116	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
232	14.333917695	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
111	14.334320949	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
112	14.334901369	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
233	14.335157220	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
234	14.335751562	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
113	14.336053138	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
114	14.336503642	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
235	14.336776109	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
236	14.337140729	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
115	14.337485056	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
116	14.337850945	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
237	14.338022523	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
238	14.338346947	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
117	14.338700574	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
118	14.339036508	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
239	14.339228031	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34
240	14.339644721	172.21.0.4	172.21.0.2	UDP	1257	50003 → 50001 Len=1215
119	14.339909061	172.20.0.2	172.20.0.3	UDP	1242	50001 → 50000 Len=1200
120	14.340277789	172.20.0.3	172.20.0.2	UDP	76	50000 → 50001 Len=34
241	14.340556721	172.21.0.2	172.21.0.4	UDP	76	50001 → 50003 Len=34

Figure 10: The figure above shows Wireshark running and capturing packets on my virtual network. I used this for debugging.

As a response to this, I piggybacked my ACK packets to contain the amount of bytes that the Client received.

While initially created to solve a problem, I would argue that this solution has its merits even without such a problem existing in the first place: If a packet is truncated, it allows the entire file to be transferred with minimal data loss.

4.3 Registration

As mentioned above, the final version contains a change from the previous versions: Workers will now automatically register with the Ingress, as opposed to being hardcoded. The intention behind this change is to make the system simultaneously more scaleable and extensible. This means that any amount of Docker containers could be started which run the Worker code, and as long as each is given a unique port number when started. This system also allows the client to receive a requested file as long as 1 of the N workers has that file.

5 Summary

In this document, I have walked through some theory behind the “Stop and Wait” protocol, the design process of my protocol, the technical details that it involved, an overview of the nodes on my network, the topology of my network, some theory about Stop-And-Wait and details about the packets I implemented to make this possible.

6 Reflection

I would like to change several things if I could start again: I would use a different programming language, I would edit my Docker setup to be less intensive on the end user, I would choose a different protocol and implement error correction schemes, and finally I would use better development practices when attempting this assignment.

6.1 Choice of Programming Language

While the abstractions Java brought initially were useful for getting up and running, I believe at a certain point it became a hindrance. The abstractions, while they lowered the barrier to entry, also raised the barrier to understanding. Object input and output streams were treated as black boxes which performed a function, but I did not understand how they performed that function.

Additionally, Java brought a lot of boilerplate code to the table: All packet subclasses were close to the same, with a few exceptions. This meant the programme was neither concise nor elegant in the end. If I had to start again with my current understanding, I believe I would choose a language such as Go to implement the protocol, as it does not engage to the same level as Java in abstractions and brings nicities such as automatic type inference to the table.

6.2 Docker Components

My Dockerfile is based on the Dockerfile given for the Java example on blackboard, with an automatically installed JDK as well. If I could change how I set up my topology, I would create several Dockerfiles, one for each type of Node on the network. I would also base my containers off these images, rather than from a single image, as it is currently. This would allow me to set environment variables so that I could easier access Wireshark and GUI applications.

I would also add some applications to the installation section of the Dockerfile, such as the text editor nano, for quick inspection of my files.

6.3 Choice of protocol

If I had more time, I would like to implement a more complex protocol with a greater degree of error control, such as Selective Repeat ARQ. This would have meant an implementation of timeouts and a greater use of

threads. Using this protocol would allow me to make a greater use out of the bandwidth of the connection between Nodes on my network.

Moreover, if I had more time I would also have implemented some error correction scheme such as CRC, Hamming Codes or Parity Checks. If I had to choose one, I believe I would like to implement Hamming Codes. Bit manipulation and low-level programming always fascinated me, so not having the time to implement these is a disappointment.

6.4 Hours Spent and Practices

I believe I spent somewhere in the realm of 20-40 hours on this assignment. Unlike other assignments, a significant portion of this was spent theorising an appropriate protocol. Nonetheless, I think I would have liked to spend even more time on designing my protocol and less on implementing it.

I should also have had a greater emphasis on the correct use of version control systems, as small edits were made within the Docker containers themselves when I needed to fix a bug, instead of fixing locally and transferring.

Another reflection I have is that I should have dedicated more time to documenting my code with neat comments - usually I believed that the code should be self-explanatory if written correctly, but with a project of this scale and speciality, that does not hold.

7 Appendix

7.1 onreceipt Methods

7.1.1 Client

```

public synchronized void onReceipt(DatagramPacket packet) {
    PacketContent content= PacketContent.fromDatagramPacket(packet);
    DatagramPacket response;
    //System.out.println("Recieved packet: " + content.toString());
    switch (content.type) {
        case PacketContent.FILEINFO:
            System.out.println("It was a FILEINFO CONTENT");
            fileData = new byte [((FileInfoContent) content).size
                ];
            response= new AckPacketContent("OK-Received this",
                0).toDatagramPacket();
            response.setSocketAddress(packet.getSocketAddress());
            try {
                socket.send(response);
                System.out.println("Send ACKPACKET");
            } catch (IOException e) {
                e.printStackTrace();
            }
            break;
        case PacketContent.FILESEND:
            for(int i = 0; i < ((FileContent)content).file.length;
                i++){
                fileData[i + bytesRec] = ((FileContent)content
                    ).file[i];
            }
            bytesRec += ((FileContent)content).file.length;
            if(bytesRec == fileData.length){
                try {
                    reqFile.createNewFile();
                    Files.write(reqFile.toPath(), fileData
                        );
                } catch (Exception e){
                    e.printStackTrace();
                    System.out.println("An exception
                        occurred creating new file: " + e)
                        ;
                }
                reqFile = null;
                fileData = null;
                fname = null;
                bytesRec = 0;
                this.notify();
            }
            response= new AckPacketContent("OK-Received this",
                ((FileContent)content).file.length).
                toDatagramPacket();
            response.setSocketAddress(packet.getSocketAddress());
            try {
                socket.send(response);
            } catch (IOException e) {

```

```
                e.printStackTrace();
            }
            break;
        case PacketContent.NOFILE:
            System.out.println("No file with that name could be found");
            reqFile = null;
            fileData = null;
            fname = null;
            bytesRec = 0;
            this.notify();
            break;
    }
}
```

Listing 2: This method is called every time the Client receives a packet.

7.1.2 Worker

```
public void onReceipt(DatagramPacket packet) {
    try {
        PacketContent content= PacketContent.fromDatagramPacket(packet
        );
        //System.out.println("Received packet" + content.toString());
        switch (content.getType()) {
            case PacketContent.GETFILE:
                String fname = ((GetFileContent)content).
                    filename;
                DatagramPacket response;
                try {
                    reqFile = new File(fname);
                    fileContent = Files.readAllBytes(reqFile.
                        toPath());
                    response = new FileInfoContent(fileContent.length).
                        toDatagramPacket();
                } catch (Exception e) {
                    response = new NoFile().toDatagramPacket();
                }

                response.setSocketAddress(packet.
                    getSocketAddress());
                socket.send(response);
                break;
            case PacketContent.ACKPACKET:
                if (fileContent != null) {
                    byte[] newData = new byte[fileContent.
                        length - ((AckPacketContent)
                        content).size];
                    for(int i = 0; i < newData.length; i
                        ++){
                        newData[i] = fileContent[((
                            AckPacketContent)content).
                            size + i];
                    }
                    fileContent = newData;
                    if (fileContent.length > PACKETSIZE){
                        byte[] slice;
                        slice = new byte[PACKETSIZE];
                        for(int i = 0; i < PACKETSIZE;
                            i++){
                            slice[i] = fileContent
                                [i];
                        }
                        response = new FileContent(
                            slice).toDatagramPacket();
                    }
                } else{
                    response = new FileContent(
                        fileContent).
                        toDatagramPacket();
                    fileContent = null;
                }
            }
        }
    }
}
```

```
                response.setSocketAddress(packet.  
                    getSocketAddress());  
                socket.send(response);  
            }  
            break;  
        }  
    }  
    catch (Exception e) {e.printStackTrace();}  
}
```

Listing 3: This method is called every time the Worker receives a packet.

7.1.3 Ingress (Server)

```

public void onReceipt(DatagramPacket packet) {
    try {
        PacketContent content= PacketContent.fromDatagramPacket(packet
        );
        //System.out.println("Received packet" + content.toString());
        switch (content.getType()) {
            case PacketContent.GETFILE:
                WORKERPORT = null;
                CLIENT_PORT = packet.getSocketAddress();
                WORKERS_WITHOUT_FILE = 0;
                for(InetSocketAddress dstAddress :
                    workerAddresses){
                    packet.setSocketAddress(dstAddress);
                    socket.send(packet);
                }
                break;
            case PacketContent.FILEINFO:
                if(WORKERPORT == null){
                    WORKERPORT = packet.getSocketAddress
                    ();
                    packet.setSocketAddress(CLIENT_PORT);
                    socket.send(packet);
                }
                break;
            case PacketContent.ACKPACKET:
                packet.setSocketAddress(WORKERPORT);
                socket.send(packet);
                break;
            case PacketContent.FILESEND:
                packet.setSocketAddress(CLIENT_PORT);
                socket.send(packet);
                break;
            case PacketContent.NOFILE:
                WORKERS_WITHOUT_FILE += 1;
                if(WORKERS_WITHOUT_FILE == workerAddresses.
                    size()){
                    packet.setSocketAddress(CLIENT_PORT);
                    socket.send(packet);
                }
                break;
            case PacketContent.REGISTER:
                workerAddresses.add(
                    (InetSocketAddress) packet.
                        getSocketAddress()
                );
                break;
        }
    }
    catch(Exception e) {e.printStackTrace();}
}

```

Listing 4: This method is called every time Ingress receives a packet.