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

CSU33031 Computer Networks
Assignment #1: UDP File Transfer

Laura Golec, 20332099

October 28, 2022

Contents

1 Introduction

The problem description for this assignment is as follows:
"The aim of the protocol is to provide a mechanism to retrieve files from a service based on UDP datagrams.The encoding of the header information of this protocol should be implemented in a binary format. The protocol involves a number of actors: One or more clients, an ingress node, and one or more workers. 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 sent them. The header information that you included in your packets has to support the identification of the requested action, the transfer of files - potentially consisting of a number of packets and

the management of the workers by the server."

In the following I will discuss the topology of my solution, along with the software and tools necessary for its completion. I will then include the details of my implemented design.

2 Theoretical Implementation

In this section I will outline 3 of the core elements of my theoretical implementation. These 3 core elements are the UDP protocol, the general topology, and the Flow Control/ARQ. I will outline my actual solutions in the next section, this section contains the theoretical ideas behind my implementation.

2.1 UDP Protocols

The User Datagram Protocol (UDP) protocol was invented as a quicker but less secure way to transfer data over the internet. It uses a simple system which does not require prior connection to transfer data. It is very effective at transferring data quicker than the alternatives and as such it is useful for streaming services. There is no guarantee that any given datagram will be delivered, ordered or not be a duplicate. These datagrams use headers to communicate the destination, origin, the length, and checksum. Below is a figure depicting the typical UDP header.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port															
4	31	Length																Checksum															

Figure 1: The above figure depicts the pseudo contents of a UDP datagram. This header is included after the appropriate IP header.

The most common alternative to UDP is Transmission Control Protocol (TCP) which requires the receiver and sender to have an established connection. While this is more reliable and is preferred protocol for many applications that do not require quick data transfer, but rather reliability. However, UDP can be made much more reliable via ARQs and flow control, which I will be discussing in further parts of the report.

2.2 Topology

The topology of the assignment as described in the brief has a central ingress and multiple client and worker nodes connected to the ingress. The only channel of communication between the clients and workers is the ingress. The workers have no communication between each other and neither do the clients. The ingress is aware of all connected nodes and sends data between the workers and clients.

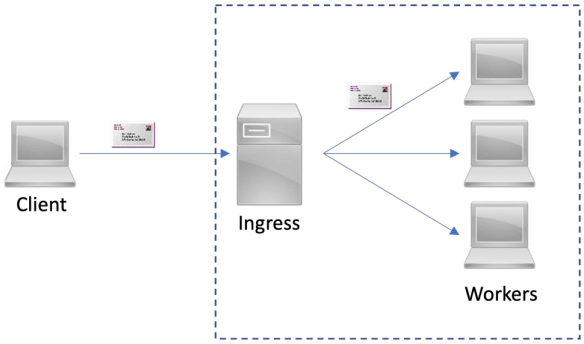

Figure 2: The above figure demonstrated the topology included in the assignment description. It is what I based my implementation on.

2.3 Flow Control and ARQ

"Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment." For my solution, I decided to implement an ARQ, which is an expansion on flow control which assumes that the channel is noisy. This is an accurate assumption as my implementation is designed to work with multiple clients and workers. The difference between the Stop-and-Wait flow control and the Stop-and-Wait ARQ is that the ARQ resends packets which it did not receive positive feedback for the packet it sent during the set timeout period, it will resend it. Meanwhile, the Stop-and-Wait flow control assumes it is in a noiseless channel and does not resend the packet and simply waits for a response from the receiver for an indefinite amount of time.

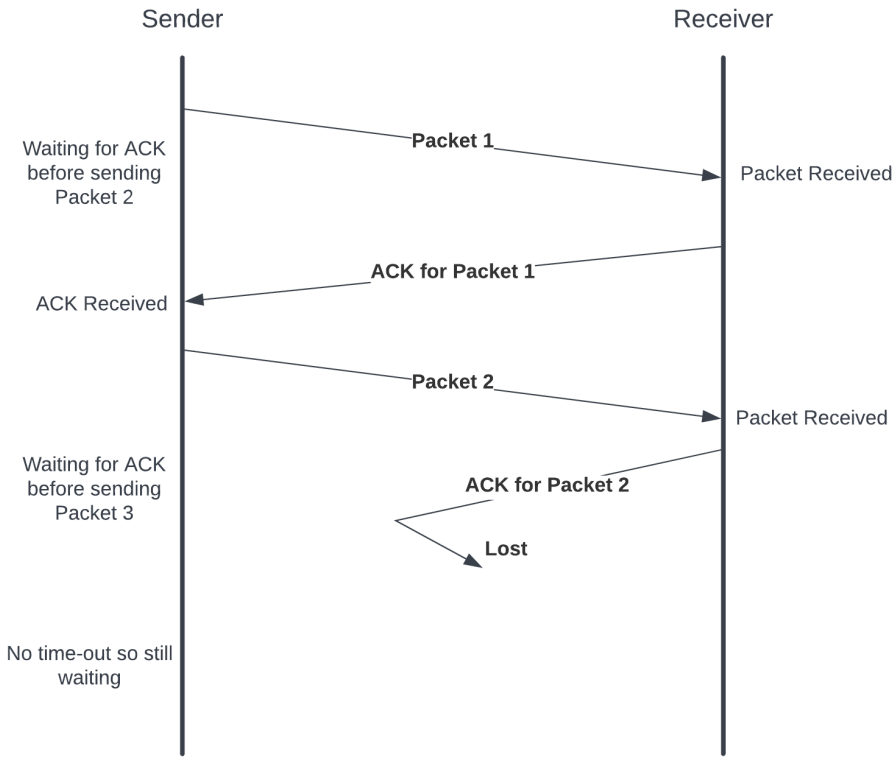

Figure 3: The above figure showcases Stop-and-Wait Flow Control, as is visible, there is no timeout window and as such when the ACK response from the receiver gets lost, the sender does not send another packet

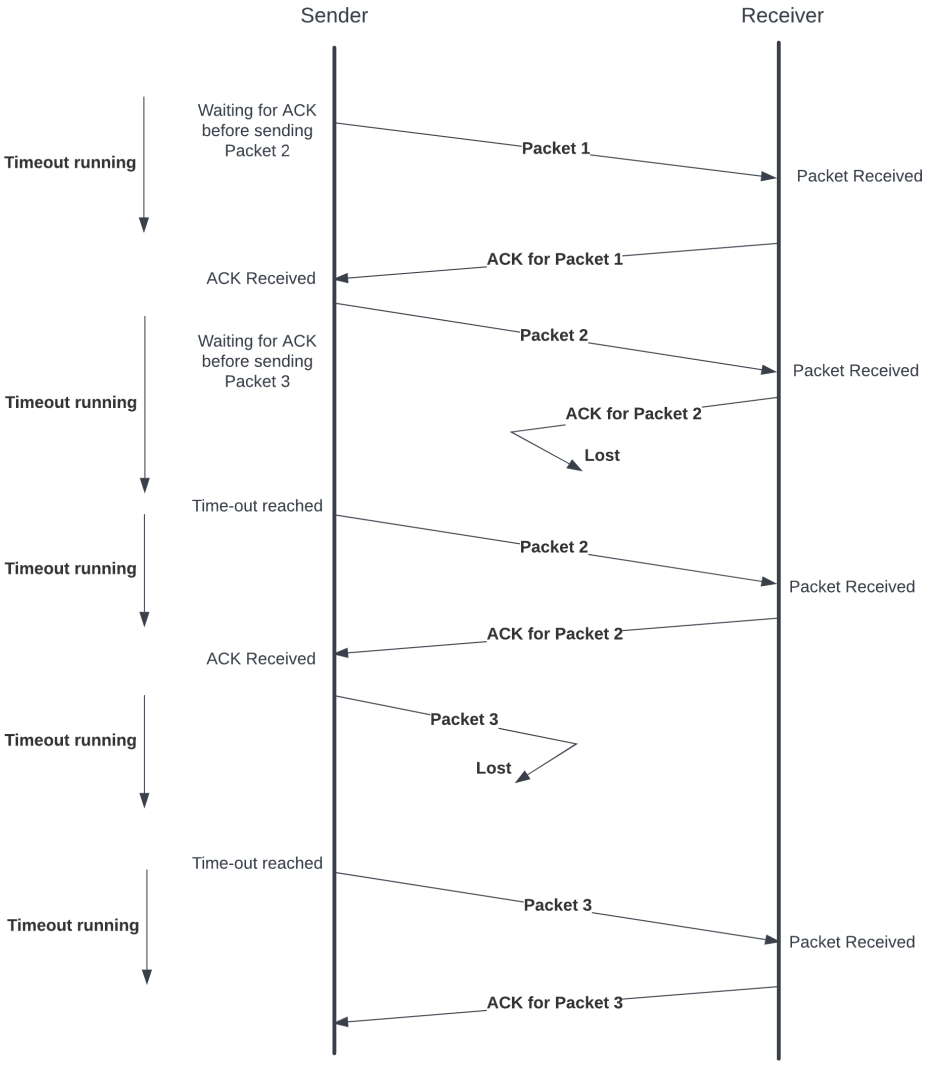

Figure 4: The above figure showcases Stop-and-Wait ARQ, due to the implemented timeout window, when the sender does not receive an ACK in time, it resends the appropriate packet, ensuring its delivery.

3 Implementation

This section goes into detail about my implementation of my solution. It outlines the technologies used along with some of the code written to carry out the protocol. It discusses the topology of my solution, the header, the implemented Stop-and-Wait ARQ, the file partitioning and merging, and queuing system.

3.1 Implemented Topology

The implemented topology of my solution is as follows. The multiple clients choose a random file from the list of files available to them and send their requests to the central ingress. The file is chosen at random and with no user input. The ingress then queues the requests and sends them to the workers. The workers partition the files and prepend the appropriate header to the packet. They then send the data back to the ingress using a Stop-and-Wait ARQ, the ingress then sends the packets to the appropriate client using the same ARQ. The client then sorts and reconstructs the file from the received packets.

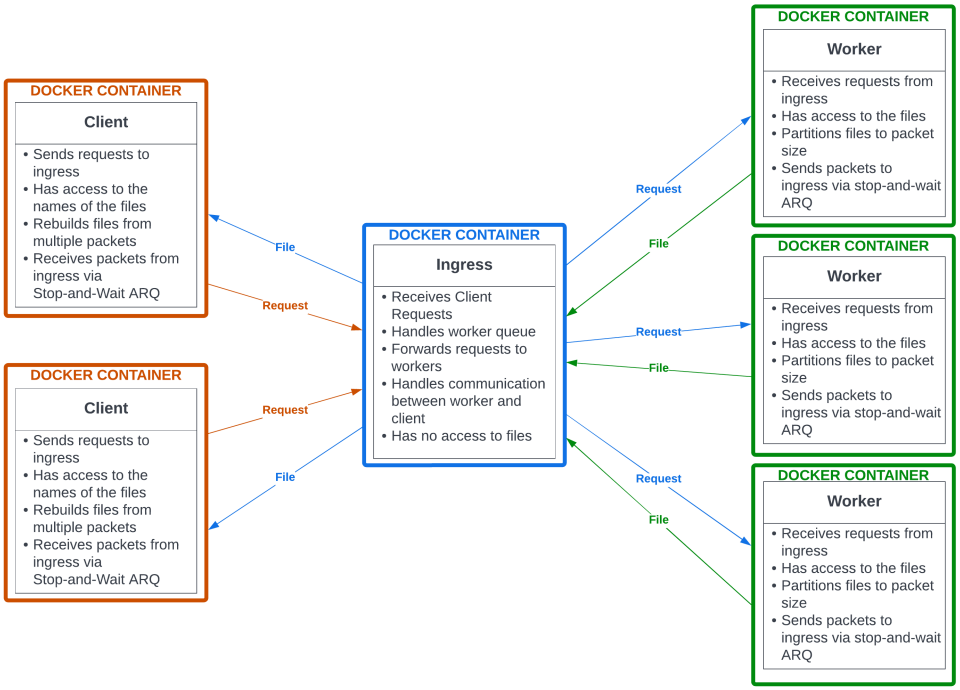

Figure 5: Above is an example of my implementation of the UDP FTP which contains 2 clients and 3 workers. It demonstrates the centering of the ingress as the communication point.

This model of file transfer allows for many clients and any amount of workers. It is scalable and allows for only the workers to have access to the files along with allocating the computational burden of partitioning and merging the files on the worker and client respectively.

3.2 Header

One of the requirements of the assignment was to create a binary header that is used for communication between all units. I decided to implement an 8 byte header. The header included the operation code (1 byte), the client number (2 bytes), part number (2 bytes), file index (2 bytes), last part boolean (1 byte).

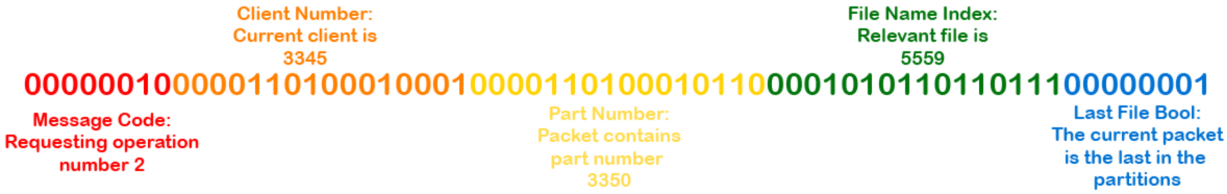

Figure 6: Above is an example of a typical header that would be sent either on its own or with packet data appended to it. As a byte is 8 bits long, the entire header comes out to 64 bits.

The operation/message code tells the unit what the message contains. For example it can be a worker declaration, or a client request, or the worker sending a file, etc. The client number is the number of the client that is involved in the transaction. This allows the ingress to send the correct file to the correct client, it is not used by the worker or client. The part number is used to keep track of the sequence number of the packet being sent, it is necessary for the Stop-and-Wait ARQ. The file index is the index in the list of

available files to request, it is used only by the client and worker. The last byte is the last file boolean. This is used to signify that the last packet in a group of packets is being sent, it is used in the ARQ. I chose to use a boolean instead of another solution which would entail having the total number of packets in a group be sent. I chose this as the boolean only requires 1 bit (1 byte in this solution) while the other solution would require 2 bytes. This header is static, even though some applications do not require the entire 8 bytes.

3.3 Implemented Stop-and-Wait ARQ

The implemented Stop-and-Wait ARQ is made up of 2 main methods with 2 helper methods. It is divided into two parts, the receiver and the sender. All of these functions are available to all units as they all make use of the ARQ. The ARQ sender sends a single packet at a a time and awaits a confirmation that the packet has been received. If the ACK is received, it proceeds to the next packet. If the ACK is not received in the timeout window, it re-sends the packet and repeats the process. The receiver waits to receive files from the sender and sends back an ACK when a packet is received. If the receiver receives duplicate packets due to the unnecessary re-sending of packets by the sender, it discards it. The receiver has no time-out and only ends the process when the file received has the "last file boolean" part of the header equal to 1. This is the most basic form of ARQ, If a new client request is received by the ingress when this ARQ is active, it is saved to an array and is returned alongside the array of packets.

```
def stopAndWaitARQSender(senderSocket, addressPort, allPacketPartitions):
    counter = 0
    receivedRequests = []
    while True:
        nextPacket = 0
        if counter < len(allPacketPartitions):
            # set the current packet to the correct packet
            currentPacket = allPacketPartitions[counter]
        else:
            # if not possible, reached the end and break function
            senderSocket.settimeout(None)
            break
        while nextPacket == 0:
            # if ACK not received, send the packet
            senderSocket.sendto(currentPacket, addressPort)
            listenPair = listenForACK(senderSocket)
            nextPacket = listenPair[0]
            if listenPair[1] != 0:
                # if request received, append list of requests
                receivedRequests.append(listenPair[1])
        if nextPacket == 1:
            # if ACK received, update the packet being sent
            counter += 1
    # return the requests received during the ARQ
    return receivedRequests
```

Listing 1: Above is the sender Stop-and-Wait ARQ. It utilises a helper function which listens for ACKs and returns any potentially received requests. The code cycles through all packets in a given array and sends them one by one, waiting for an ACK every time a new packet is sent. Once an ACK is received it moves on to the next packet in the array. The code breaks once it receives an ACK for the last packet in the sequence.

```
def stopAndWaitARQReceiver(receiverSocket):
    counter = 0
    totalReceivedPackets = []
    packetsAndRequests = [[0]]*2
    receivedRequests = []
```

```
    mostRecentPacket = 0
    received = 0
    while True:
        receivedPair = receiveFiles(receiverSocket, counter)
        mostRecentPacket = receivedPair[1]
        received = receivedPair[0]
        # if requests were received during the stop and wait arq
        if receivedPair[2] != 0:
            receivedRequests.append(receivedPair[2])
        # if the packet received is the last packet, save it and break the loop
        if received == -1:
            totalReceivedPackets.append(mostRecentPacket)
            break
        # if received, add to list and move on to next packet
        if received == 1:
            counter += 1
            totalReceivedPackets.append(mostRecentPacket)
    receiverSocket.settimeout(None)
    packetsAndRequests[0] = totalReceivedPackets
    packetsAndRequests[1] = receivedRequests
    return packetsAndRequests
```

Listing 2: Above is the receiver Stop-and-Wait ARQ. It requires a helper function which listens for packets and sends an ACK if the file is received. It also returns any requests received during the ARQ. The code creates a 2D array which will store the reeived packets along with any received requests and return them at the end of the function. It awaits a packet and sends an ACK when a packet is received. Any duplicate packets are discarded.

3.4 File Partitioning/Merging

The file partitioning system is implemented on the worker side. If a file is too big to send in the UDP packet size limit (65507 bytes), it has to be partitioned into smaller packets. This is handled by a simple for loop which continuously takes the first 65,489 bytes (the maximum UDP packet size, minus the header bytes and minus 10 bytes for some cushioning) of the remaining file bytes and prepends an appropriate header to the sectioned off bytes. It then appends it to a list of the partitioned packets. When the for loop ends, the array is returned to the main part of the worker, who then sends it to the ingress using a Stop-and-Wait ARQ.

```
def packetPartitioning(fileIndex, clientNumber):
    partitionedFileBytes = []
    file = open(availableFiles[fileIndex], "rb")
    fileBytes = file.read()
    numberOfParts = math.ceil(len(fileBytes)/maxDataSize)
    for x in range(numberOfParts):
        if x < numberOfParts - 1:
            partition = fileBytes[x*maxDataSize : (x+1)*maxDataSize]
            lastFile = 0
        else:
            partition = fileBytes[x*maxDataSize : len(fileBytes)]
            lastFile = 1
        header = createHeader(2, clientNumber, x, fileIndex, lastFile)
        partition = header + partition
        partitionedFileBytes.append(partition)
    return partitionedFileBytes
```

Listing 3: The function takes in the client number and file index. The client number is used for the header, and the file index is used for partitioning the correct file. This code checks if there are enough remaining bytes to create a new packet, if the number of bytes left is lesser than or equal to the maximum amount of bytes permitted in a packet, it marks that packet as the last file and returns the array of packets.

Once the client receives all the packets, it sorts them, then it loops through the array of packets and removes the header from each one. It then joins the packets all together as a byteArray object. This object is then written to a file with the appropriate name. This file is saved to the designated directory. This allows for easy debugging as any corruption is immediately visible when the saved file is opened.

```
def rebuildFile(partitionArray):
    partitionArray.sort(key = findPartNum)
    fileNumber = findfileNameNum(partitionArray[0])
    for index in range(len(partitionArray)):
        partitionArray[index] = removeHeader(partitionArray[index])
    finalFile = b''.join(partitionArray)
    fileInfo = [fileNumber, finalFile]
    return fileInfo
```

Listing 4: The function loops through the received pakcets and removes the header, then it joins them. It returns the reassembled file data and file name.

3.5 Queuing System

In my solution, I decided to handle multiple clients sending requests at the same time by implementing a queue. This queue would allow the ingress to work through a single request at a time. Every time a client sends request it is added to the queue in ingress. If the client request is sent during the process of the Stop-and-Wait ARQ, it is saved in an array and is added to the queue at the end of the ARQ. This solution is limited to working with only one request at a time. Once the queue is empty, the ingress waits indefinitely for the next request. It is a FIFO queue, completing requests in the order they are received.

3.6 Docker Compose

By using docker compose, I was able to easily store all docker arguments in a separate file. This allowed me to create replicas of the workers and clients, along with a designated directory which stored the received files. This was useful for debugging purposes as it served for a convenient way to check if the file had been corrupted during the transfer process.
Where Docker and Docker-Compose differ is in the building and running of the containers. Docker-Compose uses simple commands to build and run the containers as all arguments are saved in a docker-compose.yml file.
To build images using Docker-Compose, when in the project directory in terminal, the user needs to type in:

```
docker-compose build
```

This builds the images, to run the containers the user then needs to type in:

```
docker-compose up
```

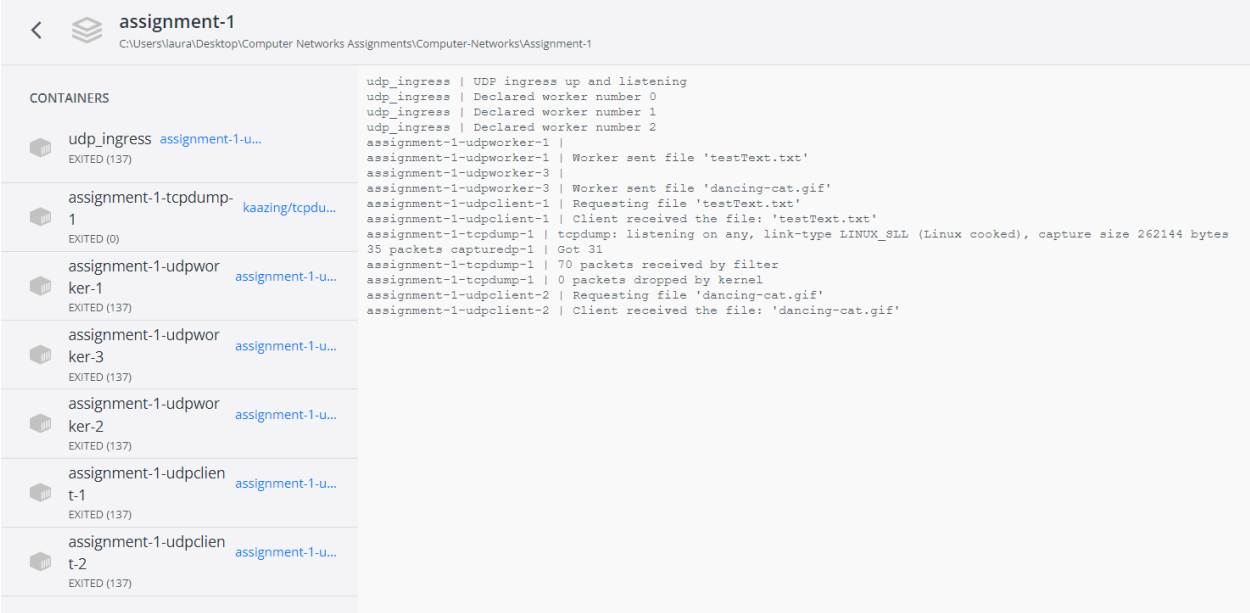

Figure 7: Above is an example of the protocol running in multiple containers in the docker desktop application.

4 Discussion

The strengths of my solution is that it allows for multiple clients and workers to be easily implemented. It also allows for files up to the size of roughly 4.3GB to be transferred. It is also quick and reliable as the Stop-and-Wait ARQ is stable and has a short time-out period of 0.01 seconds.
The queuing system implemented in the ingress is not the ideal solution, however it was implemented due to time constraints. Ideally, the protocol would implement multiprocessing which would allow for processing multiple requests at the same time. This would allow for overall quicker file transfer and for a more reliable protocol. My current protocol can encounter issues of timing out if a particularly large file is being transferred. Although file sizes of up to 4.3GB are enough for proof of concept, ideally my header would be dynamic and allow files of any size. However, as this is not necessary for this assignment, I've elected to ignore it.

```
C:\Users\laura\Desktop\Computer Networks Assignments\Computer-Networks\Assignment-1>docker-compose up
[+] Running 3/3
 - tcpdump Pulled
   - 627beaf3eaaf Pull complete
   - ae4814bdd895 Pull complete
[+] Running 7/7
 - Container udp_ingress               Created
 - Container assignment-1-tcpdump-1    Created
 - Container assignment-1-udpworker-3  Created
 - Container assignment-1-udpworker-1  Created
 - Container assignment-1-udpworker-2  Created
 - Container assignment-1-udpclient-2  Created
 - Container assignment-1-udpclient-1  Created
Attaching to assignment-1-tcpdump-1, assignment-1-udpclient-1, assignment-1-udpclient-2, assignment-1-udpworker-1, assignment-1-udpworker-2, assignment-1-udpworker-3, udp_ingress
udp_ingress               | UDP ingress up and listening
assignment-1-tcpdump-1    | tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
udp_ingress               | Declared worker number 0
udp_ingress               | Declared worker number 1
udp_ingress               | Declared worker number 2
assignment-1-udpclient-1  | Requesting file 'testText.txt'
assignment-1-udpclient-2  | Requesting file 'dancing-cat.gif'
assignment-1-udpworker-1  |
assignment-1-udpworker-1  | Worker sent file 'testText.txt'
assignment-1-udpclient-1  | Client received the file: 'testText.txt'
assignment-1-udpworker-3  |
assignment-1-udpworker-3  | Worker sent file 'dancing-cat.gif'
assignment-1-udpclient-2  | Client received the file: 'dancing-cat.gif'
```

Figure 8: This figure shows the transfer of files in the terminal.

.
In figure 8, we can see the process of the file being transferred in the terminal. On the left we can see the different clients and workers along with the single ingress. We can see the workers declaring themselves, then the clients requesting files, the workers sending the files to ingress and ingress sending them to the clients.

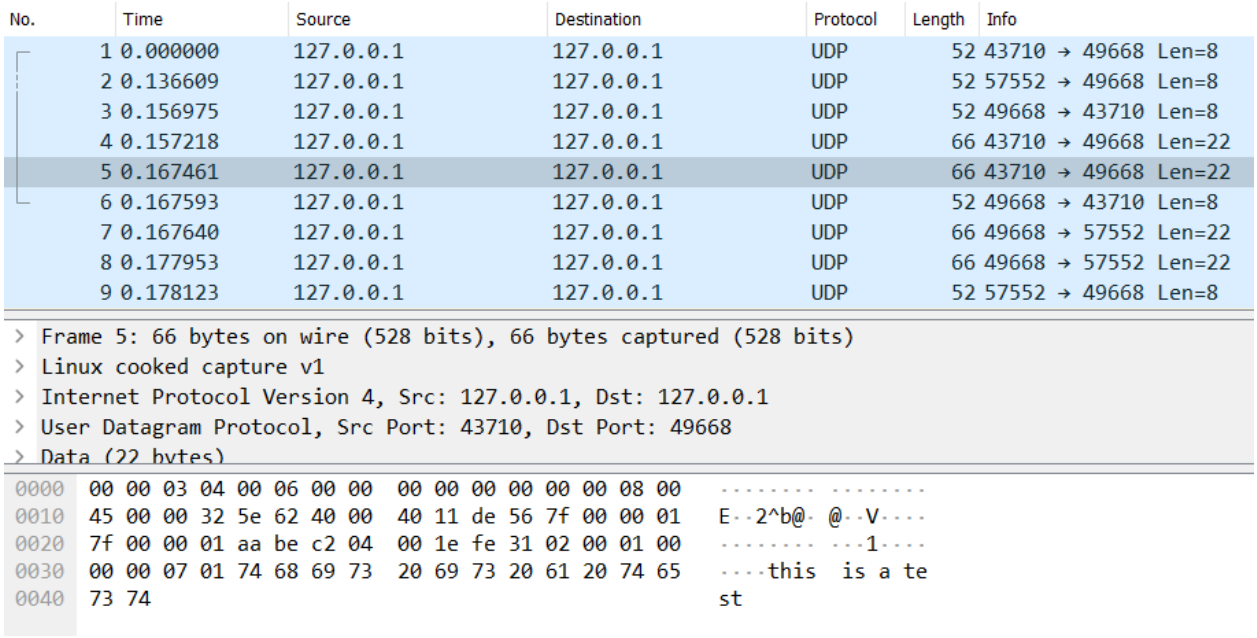

Figure 9: This figure shows the transfer of a small text file over the network captured in a PCAP file.

Figure 9 is a demonstration of the text file containing "this is a test" being requested by a client and sent from the worker to the client via the ingress.

5 Summary

This report has discussed my solution to implementing a File Transfer Protocol utilising UDP using containerisation software. While the solution has a limit on the amount of clients, different files and file size, it demonstrates a possible approach, that can be altered to potentially be fully automatically scalable. The description of my implementation in this report highlights the most import aspect of my solution that are required for the protocol to work in the example topology given.

6 Reflection

Overall, I am satisfied with my solution despite its inability to work concurrently and also the multiple limitations on the scalability due to the fixed header size. I greatly enjoyed the structure of the assignment as it allowed for greater freedom in time allocation and it allowed for working at a suitable pace. It also allowed me to do lots of research early on, which led to a much greater understanding of the topic. I am very satisfied with my understanding of both UDP and file transfer protocols, including various ARQs.
To summarise, excluding the report the assignment took me around 40 hours to complete.