



CSU33031 Computer Networks

Assignment #1: File Retrieval Protocol

Claire Gregg, 20332015

September 11, 2023

Contents

1	Introduction	1
2	Theory of Topic	2
2.1	UDP and UDP Datagrams	2
2.2	Topology	2
2.3	Flow Control and ARQ	3
3	Implementation	4
3.1	Overall Topology	4
3.2	Header Anatomy	5
3.3	ARQ Implementation	7
3.4	Client	9
3.5	Worker	10
3.6	Ingress	11
4	Discussion	12
5	Summary	12
6	Reflection	13

1 Introduction

The general problem this assignment addresses is the design of a file retrieval protocol which runs over UDP. It described a topology with multiple clients and workers all interacting with an ingress node. This assignment provides an insight into the balancing act between introducing new functionality which increases the size of the header, and the overhead that header adds, as well as efficiency overall. The topology described in the assignment requires clients to request files from ingress, which distributes the requests to workers. In the following, I will discuss my approach to this problem. Then, I will discuss my design of the network elements of my solution, and describe the data passed between network elements, including the headers used to communicate between elements. I implemented a variable length header which allows for files of up to about 4 gigabytes to be sent, potentially in multiple packets, across the system.

2 Theory of Topic

This section will describe the theory underlying the assignment, as well as the theory underlying my implementation. It is important to understand the basis on which my solution is built. In the first subsection I will explain my understanding of UDP and UDP datagrams. In the second I will explain the theoretical topology of the system in this problem. In the third I will explain the concept of ARQs and flow control.

2.1 UDP and UDP Datagrams

UDP (User Datagram Protocol) is a protocol to send messages over IP. It lies in the Transport layer of the OSI stack. Unlike TCP, UDP does not require prior/initial communication to set up communication channels. This leads to a reduced overhead when a system is based on UDP. However, this also leads to UDP being generally unreliable, as it has no error checking or correction. UDP is generally used in cases where speed is most important.

A UDP datagram consists of a UDP header (8 bytes) followed by the data (payload). The UDP header is shown in Table 1.

Bytes	0	1	2	3	4	5	6	7
Content	Source port		Destination port		Length		Checksum	

Table 1: UDP Datagram Protocol Header. The source port refers to the port the datagram has been sent from. The destination port refers to the port the datagram is being sent to. The length refers to the length in bytes of the datagram as a whole (header and data). The checksum field may be used for error checking, although it is not in the solution discussed here.

The maximum length of a standard UDP Datagram is 65,535 bytes. However, 8 bytes of this is the UDP header, and 20 bytes of this is the IP header. Therefore, one UDP datagram can have a maximum of 65,507 actual data bytes. So, files larger than this size will need to be split into multiple datagrams. Also, importantly, the header implemented in my solution is written in these 65,507 bytes, so the number of bytes for data ends up being fewer.

UDP datagrams are sent from sockets to sockets. Python has a library which can be used to work with these sockets.

2.2 Topology

It is important to describe the theoretical topology of the system before going straight into my implementation of it. The protocol involves several types of actors, in varying quantities: one or more clients; one ingress node; and one or more workers. This is shown in Figure 1.

Any client issues requests for files to an ingress node. The ingress node receives these requests and distributes them to the workers. The workers read the file from locally, then send it back to ingress, potentially in several packets. Ingress forwards these replies back to the correct client. The client finally receives these files. However the topology is implemented in the solution, it must (and does) follow this.

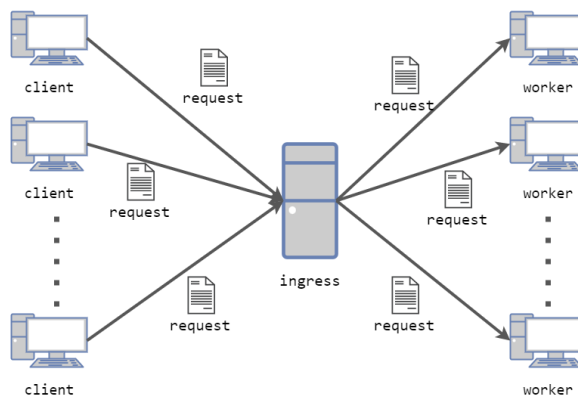


Figure 1: A representation of the potential topology of the system the file transfer will work on. A number of clients send file requests to a server (ingress) which distributes the file requests to available workers.

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." - Forouzan. In sending information, there is always the possibility that the sender can send data faster than the receiver can receive and write the information it can receive. Flow control in general refers the control of the amount of data a sender can transmit before it overloads the receiver.

Pure flow control protocols rely on a noiseless channel. One of these does not have any overhead, and instead relies on every message arriving at the receiver. A flow diagram of this is shown in Figure 2. Parts of my solution are based on a variation of this simplest flow control protocol. An alternate pure flow control protocol is stop and wait, where the sender transmits a message, and waits to receive an acknowledgement message. Once it has received an acknowledgement, it sends the next message.

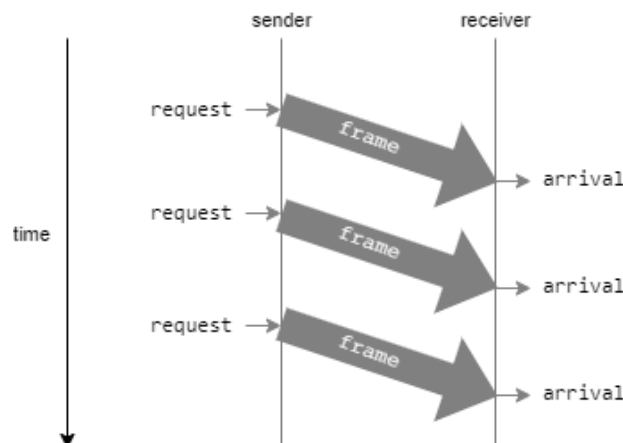


Figure 2: A flow diagram showing the simplest possible flow control protocol. The sender sends packets as fast as it can, and the receiver must receive all packets.

However, flow control alone must assume that all messages eventually arrive. This is not true in general - messages can get lost or corrupted (although corruption is out of the scope of my solution). Then, the system must have a way to deal with this. These methods are as a whole referred to as Error Control, and include the key elements of error detection, positive acknowledgements for valid, error-free frames (ACKs), negative acknowledgements for corrupted frames (NACKs), and automatic retransmission of frames after timeout or receipt of NACKs. However, variations on these exist where the error control only controls for lost messages.

These require only ACKs and automatic retransmission of frames after a timeout. One version of this is a Stop and Wait ARQ (Automatic Repeat reQuest) without error checking.

Flow diagrams for different cases of this ARQ are shown in Figure 3. In Figure 3a, the ARQ is not required as the frames and ACKs are received as appropriate. In Figure 3b, a frame goes missing. When a frame leaves the sender, a timeout is set. If, when that timeout runs out, the ACK for the file segment has not been received, it will resend the frame. It will repeat this until the ACK for the frame is successfully received. In Figure 3c, an ACK goes missing. This is the same as when a frame goes missing from the sender's point of view. However, when the receiver receives the same frame from receiver again, it discards it and resends the ACK.

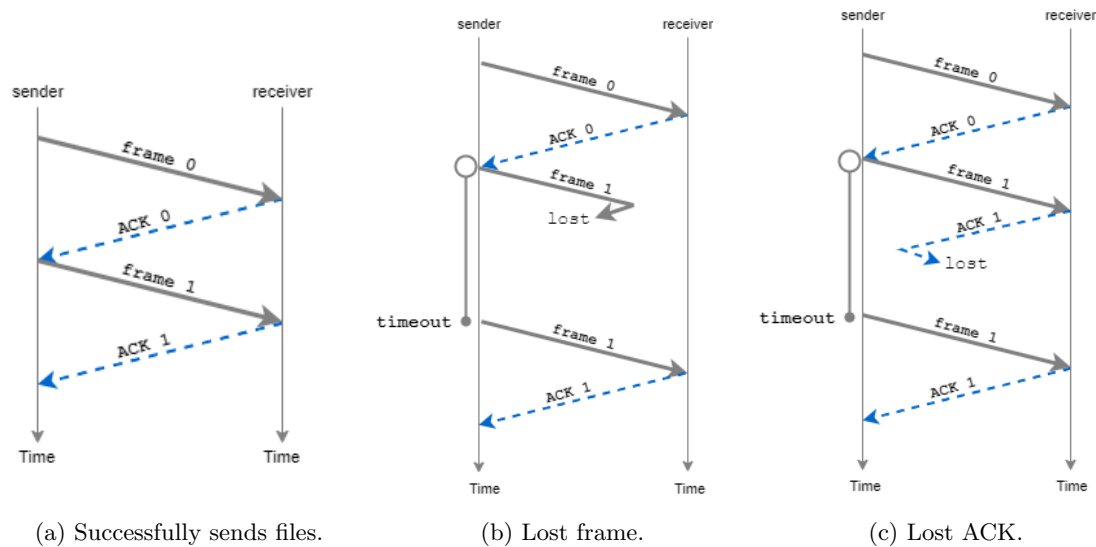


Figure 3: Stop and Wait ARQ in different situations

3 Implementation

This section provides the overall design of my solution followed by the implementation details of the individual network elements: client; ingress; and workers. It discusses the implementation of the network as a whole using docker-compose. It also discusses the layout of the messages between the network elements. Then, it discusses the packetising of large files, and ARQ for lost packets.

3.1 Overall Topology

A diagram of the overall topology is provided in Figure 4. Up to 256 clients can request files from ingress. This is limited by the number of bytes in the header provided for identifying which client a request is from. However, this could be easily changed. Requests of files go to ingress, who distributes them to ingress based on a first in first out (FIFO) queue of workers. There is no limit on the number of workers, as each one declares itself to ingress by sending a declaration message. When a worker receives a file request, it reads the file from memory. Then, it sends the file back to the ingress, potentially in multiple sections if it is too large. Ingress returns these files to the clients who requested them.

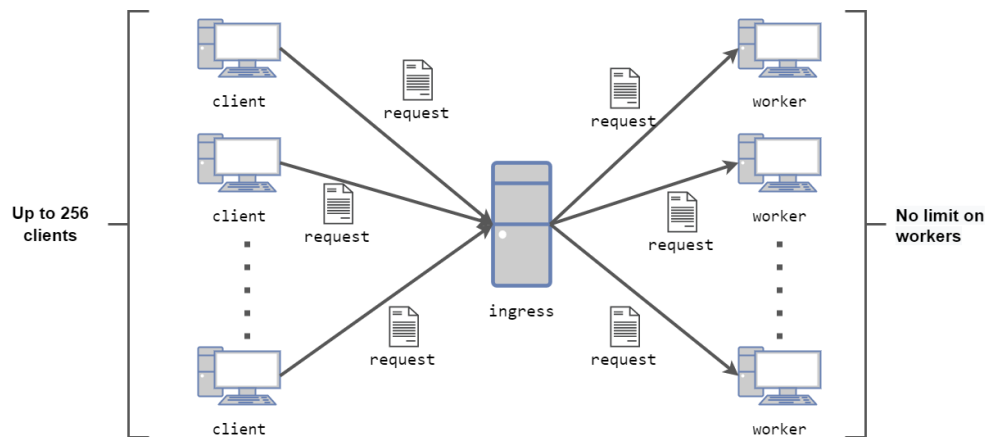


Figure 4: A representation of the topology of my solution. Up to 256 clients can request files from one ingress server. The ingress server distributes these requests to an unlimited number of workers.

I implemented this topology using Docker Compose, which is a tool for working with multi-container Docker applications. For this, each network element has a Dockerfile which defines information like how to run it, where to get its dependencies from, and where to put its output, if it needs to. To use Docker Compose, there is also a docker-compose.yml file, which defines the overall network. This includes each type of "service" (network element), how to build them, what other elements they depend on, how many replications of each of them should exist, and what volumes (for output) they have. This allows very easy testing of different network configurations. Usually, the solution network consists of: the ingress server; 1+ workers, which depend on ingress; 1+ standard clients, which depend on ingress and workers, and put their output in an output volume; 1 interactive client which can be controlled through the command line, which depends on ingress and workers, and which puts its output in another output volume; and a tcpdump service which dumps the PCAP file of network traffic into a volume.

3.2 Header Anatomy

To identify details about a message sent through the network, each frame has a header which has control information about the message. In my solution, the header consists of what is shown in Table 2, which is described in greater detail below.

Bytes	0	1	2	3	4	5	6	7-X	X-Y
Content	Header Length	Control	Client	Part of File		Number of Bytes for Received Parts (Req. only)		File Name	Received Parts (Req. only)

Table 2: Identification of bytes of header for file transfer.

- 0) **Header Length:** This is the number of bytes the header contains (excluding received parts bytes). This length includes: this byte for header length; one control byte; one byte to identify which client a request is coming from; two bytes to identify which segment of a file is currently being sent; two bytes to identify the number of bytes required to state which file segments have already been received (if it is a request); and the number of letters in the name of the file being requested.
- 1) **Control Byte:** This byte contains all information which can be stored in single bits. This is shown in Table 3.

Bits	0	1	2	3	4	5	6	7
Content	Unused		Ack	Request	Not Final Part	From Client	From Worker	Declaration

Table 3: Control Byte in file transfer header. Bit 7 is set if the message is a declaration. This is used when workers are declaring themselves to ingress as available. Bits 6 and 5 are set when a message is coming from a worker and client respectively. These are used in ingress so that it can identify what to do with a message based on where it came from. This could easily be removed with the current design however, as a request or ACK will always come from client, and things which are not requests or ACKs will always come from workers. Bit 4 is set when a file segment is being sent which is not the final segment, so that all of the network elements can identify if a file is finished being sent. Bit 3 is set if a message is a file request so that ingress can identify that case. Bit 2 is set if the client is acknowledging if it has received the whole file so that ingress can free up the worker which was sending the file. The most significant bits are unused as they are not necessary.

- 2) Client:** This byte identifies which client a request is coming from. When a request is coming directly from a client this is not set, but once it gets to ingress, ingress finds the index of the client in its list of clients and sets this byte to that index. This allows workers to send files (and file segments) back to ingress with ingress knowing which specific client to send them to.
- 3-4) Part of File:** These bytes identify which segment of the file is contained in the message. If there is no segment of file (in the case it is a request) these are set to 0. This allows client to reorder file segments in case they arrive in the wrong order, and allows clients to track which file segments they have received. This, in turn, allows the client to communicate to worker which file segments it has received, so worker knows which file segments to send on again. I will go into more detail of this in the next section, 3.3, ARQ Implementation.
- 5-6) Number of Bytes for Received Parts:** These bytes are only included in file requests. These bytes identify how many bytes the "received parts" bytes will take up. This is necessary as this added with the header length byte gives the overall length of the header, and therefore allows knowledge of where the payload starts. This also allows the client to accurately write which file segments it has received, and worker to know how to read which file segments have been received. This will be discussed further in 3.4 Client and 3.5 Worker.
- 5/7-X) File Name:** These bytes contain the name of the file being requested, regardless of whether the message is a request or a response. This allows every network element to keep track of the requested file, and makes for easier debugging. This also ensures workers can select which file has been requested and send it on.
- X-Y) Received Parts:** These bytes are only included in a request. They store the information about which segments of a file the client has received. When the client has not received any file segments, there are no received parts bytes. If the client has received file segments, indexing from the least significant bit, the bit which corresponds to each file received is set. An example of this is shown in Table 4. This can become space inefficient for large files, which I will discuss later in Section 4, Discussion.

Bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	1	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0

Table 4: Received Segments Byte Example. In this example, file parts 0, 3, 7, and 9 have been received, so those bits are set. The remaining bits are not set. In this case, number of bytes for received parts (bytes 5-6 of the header) would be set to 2.

An example of a request header in Wireshark is shown in Figure 5, and an example of a response header in Wireshark is shown in Figure 6.

0000	00 00 03 04 00 06 00 00	00 00 00 00 00 00 08 00
0010	45 00 00 31 5d c8 40 00	40 11 de f1 7f 00 00 01	E-1] .@- @-.....
0020	7f 00 00 01 ca de 4e 21	00 1d fe 30 15 14 00 00N! ...0-...
0030	00 00 00 74 65 73 74 5f	76 69 64 65 6f 2e 6d 70	..-test_ video.mp
0040	34		4

Figure 5: This shows an example header for a file request. Up to byte 43 is part of the IP and UDP headers - the start of my header is circled. This is the header length, 0x15, which is 21 in decimal. The second byte in the header, the control byte is 0x14. This is 0b00010100 in binary, so it has the *request* and *from client* bits set. The third byte in the header, client, is set to 0. The fourth and fifth bytes are empty as it is a request, and the sixth and seventh bytes are empty as the client has not received any parts yet. Then, the file name starts, with a value of 74, which translates to 't' in ASCII.

00000000	00 00 03 04 00 06 00 00	00 00 00 00 00 00 08 00
00000010	45 00 ff fd 5d d8 40 00	40 11 df 14 7f 00 00 01	E-...]-@- @-.....
00000020	7f 00 00 01 b8 67 4e 21	ff e9 fd fd 13 0a 01 00gN!
00000030	00 74 65 73 74 5f 76 69	64 65 6f 2e 6d 70 34 00	-test_vi deo.mp4-

Figure 6: This shows an example header for a response with a file. Up to byte 43 is part of the IP and UDP headers - the start of the header is once again circled. This is the header length, 0x13, which is 19 in decimal. The second header byte, the control byte, is 0x0a, which is 0b00001010 in binary, so it has the *not final part* and *from worker* bits set. The client byte is set to 1, which is the client the file request came from. The part of file bytes are set to 0 as this is the first file segment. The file name starts immediately after that.

3.3 ARQ Implementation

I implemented my own Flow/Error Control ARQ to experiment with the topic. It acts as a combination of the simplest flow control protocol (sending all messages as fast as you can, and assuming they all arrive), and stop and wait ARQ. Every time a client requests a file, it requests it based on which parts it has already received. The details of how the client sends the request is discussed in more detail in Section 3.4, Client. When the client first requests a file, it requests it saying it has received no parts. Ingress passes this request along to the worker indexing which client requested the file. The worker receives this message. It reads the file (if it has not already got it in memory). Then, it sends only the segments which have not been received (as specified in the Parts Received bytes of the header). This will be discussed in more detail in Section 3.5, Worker. Ingress passes these messages along. Client receives these messages. When its timeout runs out, it notes down which file segments it has received, and sends another request with those details, given it has not received all file segments. How a client tells if it has received all file segments is discussed in in Section 3.4, Client. Eventually, the client will have received all file segments, and sends an ACK to ingress, who frees up the worker by added it to the free workers queue. Flow diagrams of this in different situations are shown in Figure 7 and Figure 8. There is also a larger example with more clients and workers in Figure 9.

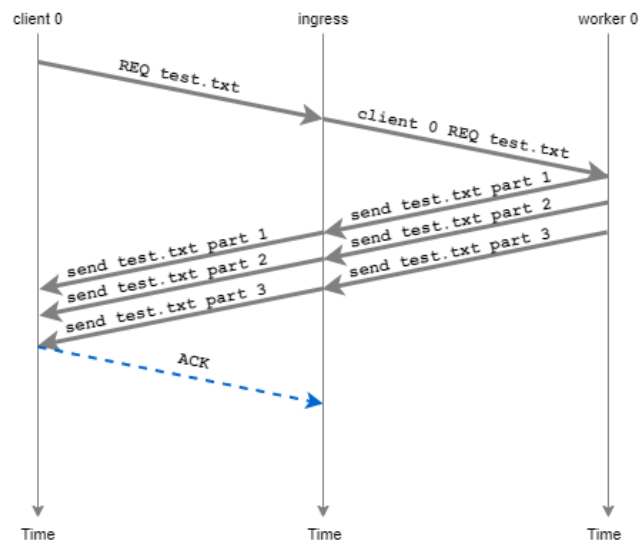


Figure 7: In this flow diagram, the client requests a file, and receives it in segments without any repeat request necessary.

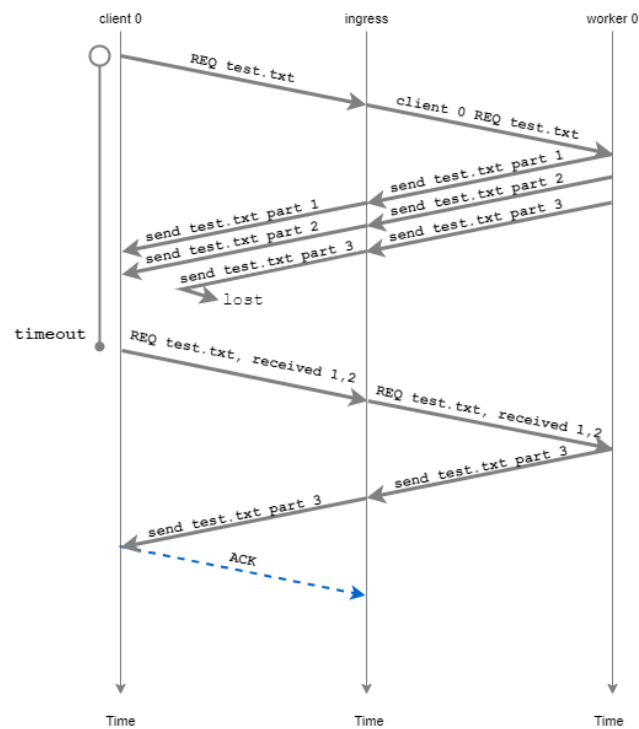


Figure 8: In this flow diagram, the client requests a file. The worker sends all segments of the file, but one of them (segment 3) gets lost. It acts the same whether the segment gets lost between worker and ingress or between ingress and client. Client receives the first 2 segments of the file. Eventually its timeout runs out, at which point it sends a request as it has not received all segments of the file. Worker receives this request, and sends only the file segments which have not already been received.

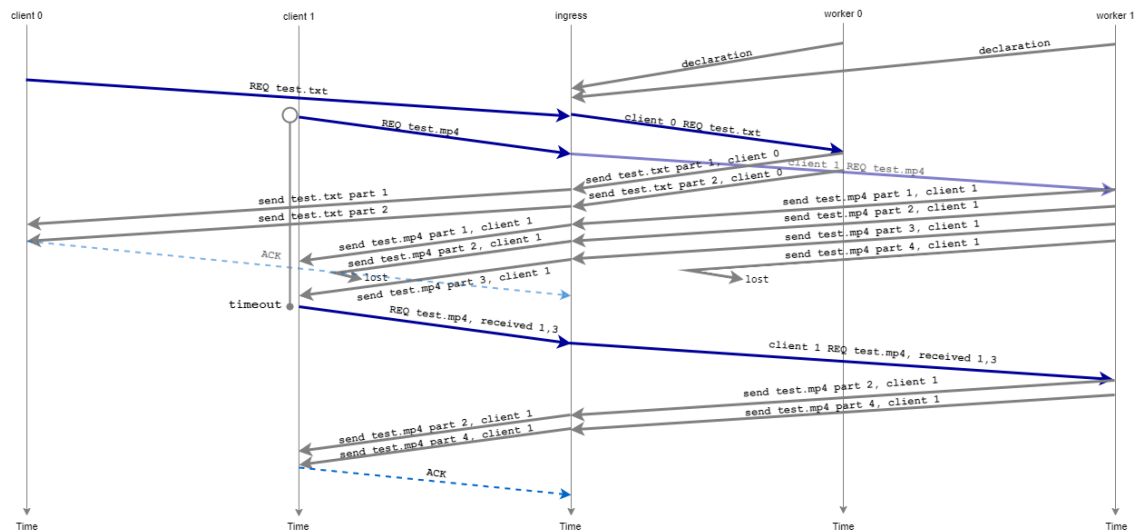


Figure 9: This flow diagram shows the ARQ in action with multiple clients and workers, so also shows the topology. The first client requests and receives all segments of a file like in Figure 7 and the second client has a similar experience to that in Figure 8, but they are shown here together.

3.4 Client

There are two versions of the client network in my solution, so that the system can be tested in a variety of solutions. Both versions initially set up their UDP sockets so that they can send and receive datagrams, with no particular port number or IP address, leaving Docker to assign those. The standard version of client does not require any input, and chooses which file it requests randomly from the list of possible files. These clients are set up automatically based on the Docker Compose setup when `docker-compose up` is run. The other type of client is the interactive client. This takes the file name to request from the command line in a loop as shown in Listing 1. The interactive client is set up when `docker-compose up`, but to interact with it, `docker-compose run interactive_client sh` must be run in another terminal, while the other containers are still up.

```
while True:
    chosenFile = input("Please select a file to request or 'exit' if you are
    ↪ finished:")
    if chosenFile == "exit":
        break
    if chosenFile not in fileNames:
        print("Sorry, that file is not available.")
        continue
    #... requests file
```

Listing 1: At the interactive *client*, the user is being asked to input the name of the file that is to be requested from the workers. This happens in a loop, and once a valid file name has been entered, the file is requested. Once the file has been received, the loop to get a file name and request it restarts.

Once a file has been selected to request, a request is sent and the timeout on the socket is set.

```
UDPClientSocket.sendto(bytesToSend, ingressAddressPort)
UDPClientSocket.settimeout(timeout)
```

Listing 2: In *client*, a file request is sent. *bytesToSend* follows the structure described in Section 3.2. The control byte has the request and from client bits set. The client byte and part of file bytes are set to 0. Number of Bytes for Received Parts is set to 0, and there are no received parts bytes. The file name is set to whichever file has been selected.

Once the request has been sent, while the socket timeout has not run out, the client waits to receive file segments. It receives file segments using the inbuilt `socket.recvfrom(buffer_size)`, which returns the datagram sent to the socket. The command also just blocks until it receives a message. Once a segment is received, it adds it to the list of received segments. Importantly, if the file segment received is the last file segment, the client now knows exactly how many segments it needs. This is shown in Listing 3. If the total length of the file segments received equals the total number of segments received, then the file has been received. In that case, the file segments are ordered, an ACK is sent to ingress, and the file is written to the client's volume.

```
if message[protocol.lib.controlIndex] & protocol.lib.notFinalSegmentMask !=
    → protocol.lib.notFinalSegmentMask:
    totalFileSegmentNumber = fileSegmentNumber+1
```

Listing 3: If a segment bitwise ADDED to the `notFinalSegmentMask` gives the mask back, then the segment is not the final segment. Therefore, otherwise, it is the final segment, and the client knows the total number of parts of the file.

In the case that the timeout on the socket runs out before all segments have been received, another request is sent with the Received Parts bytes set based on what parts it has received. The way these bytes are calculated is shown in Listing 4. With these *segments received* bytes, the file is requested again, and the timeout is reset. This is repeated until the file is fully received.

```
def write_segments_received(receivedSegmentNumbers, numBytesPartsReceived):
    segmentsReceived = 0
    for i in range((numBytesPartsReceived*8)-1):
        if i in receivedSegmentNumbers:
            segmentsReceived = segmentsReceived | 0b1
        segmentsReceived = segmentsReceived << 1
    return segmentsReceived.to_bytes(numBytesPartsReceived, 'big')
```

Listing 4: The function which calculates the segments received bytes for the header of a request. *receivedSegmentNumbers* is a list of the segment numbers of segments received. *numBytesPartsReceived* is calculated by dividing the max value in *receivedSegmentNumbers* by 8 (number of bits in a byte), and rounding it up to the nearest integer. For each but index of those bytes, if that index is in the list, it is set to 1 and shifted left, so that you end up with bytes describing the received parts as described in Section 3.2

3.5 Worker

Each worker starts by setting up its socket and declaring itself to ingress. The worker declares itself to ingress by setting the bits for *declaration* and *from worker* in the control bit, and sending that message.

Whenever a worker receives a message, it gets the file name. If the file name is the same as the file currently saved in memory (which is initially empty), it returns the existing file name and file. Otherwise, it reads the file from memory using `f = open(fileName, "rb")`, then `bytes_read = f.read()`. Then, the worker loops through *bytes_read*, splitting it into a list where each element is the max length there is space to send back in one UDP frame, ensuring there is enough space for the header. When the final segment is being read, it is usually shorter than the max possible length, and then the loop is broken out of.

To read the file parts received bytes, worker essentially works through the code in Listing 4 in reverse. It shifts the bytes right, and at each index, if it is set, add that index to a list.

The file is sent simply. It loops through the list of file segments, and sends each one with the appropriate header given their segment number. Importantly, when it is the last file, the `notFinalPart` bit is *not* set. To improve efficiency, the file segments are sent in chunks of 3, with a very short wait between each chunk. I found that this improved the proportion of segments received by the client each time.

3.6 Ingress

Ingress' role is to organise requests from clients, transmit them to the appropriate worker, and send responses from workers to the necessary client. As ingress must be capable of dealing with a large number of inputs, I have implemented multiprocessing for it. The first thing ingress does when set up is initialise its socket at the well known port 20001, so that clients and workers know where to find it. Then, ingress initialises all of its variables which must be able to be safely accessed by several processes safely. For this reason, ingress' queue of free workers, list of clients for indexing, and dictionary of workers which are being used (which is indexed by the client indexes) are set up with mutual exclusion locks.

Once all of the necessary parts of ingress are set up, it loops on receiving messages, as shown in Listing 5.

```
while True:
    bytesAddressPair = UDPServerSocket.recvfrom(protocol_lib.bufferSize)
    process = multiprocessing.Process(target=deal_with_recv, args=(
        ↪ bytesAddressPair, workers, clients, workersInUse, lockWorkers,
        ↪ lockClients, lockWorkersInUse))
    process.start()
```

Listing 5: Ingress loops receiving messages from its socket. Whenever it receives a message, it starts a new process with that as well as all of the shared variables. That process deals with whatever is inside the message.

In `deal_with_recv`, ingress deals with all possibilities. It finds out which case has occurred by masking the control byte with the relevant mask for each possibility

Message is from client If this is the first time the client has sent a request, ingress will add the client's address to the list of clients. Regardless, ingress finds the client's index in that list of clients. Then, there are 2 possibilities.

Message is an ACK Remove the worker associated with that client's index from the dictionary of workers which are being used, and add it back to the queue of free workers.

Message is a request Find the client's worker. It may need to take one from the free workers queue, shown in Listing 6. Then, send the request (with the client byte replaced with the client's index) to the worker.

```
if clientIndex not in workersInUse:
    # If there is no worker currently available, block until there is
    worker = workers.get(block=True, timeout=None)
    lockWorkersInUse.acquire()
    workersInUse[clientIndex] = worker
    lockWorkersInUse.release()
else:
    worker = workersInUse[clientIndex]
```

Listing 6: If client already has a worker, return that. Otherwise, take one from the free workers queue, and add it to the workers in use dictionary, indexed on the client's index. If no worker is available, this will block until one is made free.

Message is from worker There are two possibilities if a message is from a worker.

Message is a declaration If the worker is declaring itself to ingress, add it to the free workers queue.

Message is a response Find the client number from the header. Get the address of that client using the list of clients, indexing with the client number. Then, send the response on to the appropriate client.

4 Discussion

The strengths of my solution are the speed at which files can be sent in a quiet system, and the speed at which medium sized files are transmitted. This is because there is no need to wait for ACKs for individual file segments, and instead we rely on a timeout in client. When the timeout in client runs out, the client re-requests the file. My solution's use of Docker Compose for testing allowed far greater complexity in testing, as there was no need to create each network element individually. My solution can also deal with files of sizes up to about 4GB in theory.

However, this solution is far from ideal. The larger a file gets, the more inefficient the ARQ becomes, as for every about 525kB (kilobytes) in a file, another byte will be added to the header for a request. Therefore, for a file of size 3.5GB, the header size for a request will approach 7000 bytes when the file has almost completely arrived. This is quite inefficient.

Another weakness is that there is no system for dealing with a lost ACK from a client to ingress, which means that workers might eventually all be trapped in the workers being used dictionary. However, I never found this to be an issue. As well as this, there is no way for my solution to deal with over 256 clients, which is not ideal. However, the number of clients possible to be dealt with could be easily increased.

As well as those, there is no checking for a file which does not exist being requested on the worker's side. If the client manages to request a file which does not exist, it will cause the worker to crash. Then when the client timed out, it would request the file again. This is prevented by manually keeping an accurate list of the files available in *client*. Ideally, I would also implement a request which causes a worker to return all the files it has access to, which is how client would have access to the list of files, but time prevented this feature.

However, overall I feel that I have a good solution to the problem presented, and provide an interesting, if inefficient, ARQ option.

5 Summary

All of what I have described here comes together to allow file transfer of files up to about 4 gigabytes over a virtual topology. It is shown functioning in Figures 10 and 11.

```

> OPEN EDITORS
  NETWORKS_UDP_PROTOCOL
  client
  ingress
  interactive_client
  interactive_output
  test_video.mp4
  test.txt
  output
  tcpdump
  worker
  .gitignore
  docker-compose.yml
  protocol.lib.py
  README.md
  U

> TERMINAL
ing file medium_test_image.png | Requesting file medium_tes
networks_udp_protocol-client-1 |
t_image.png | Requesting file test_video
networks_udp_protocol-client-3 | .mp4
networks_udp_protocol-client-10 | Requesting file test.txt
networks_udp_protocol-client-9 | Requesting file medium_tes
t_image.png |
networks_udp_protocol-client-8 | Requesting file longer tha
n_buffer_test.txtnetworks_udp_protocol-client-7 | Received
file.
networks_udp_protocol-client-7 exited with code 0
networks_udp_protocol-client-5 | Received file.
networks_udp_protocol-client-4 | Received file.
networks_udp_protocol-client-5 exited with code 0
networks_udp_protocol-client-4 exited with code 0
networks_udp_protocol-client-10 | Received file.
networks_udp_protocol-client-8 | Received file.
networks_udp_protocol-client-1 | Received file.
networks_udp_protocol-client-10 exited with code 0
networks_udp_protocol-client-9 | Received file.
networks_udp_protocol-client-8 exited with code 0
networks_udp_protocol-client-9 exited with code 0
networks_udp_protocol-client-1 exited with code 0
networks_udp_protocol-client-6 | Received file.
networks_udp_protocol-client-6 exited with code 0
networks_udp_protocol-client-3 | Received file.
networks_udp_protocol-client-3 exited with code 0
networks_udp_protocol-client-2 | Received file.
networks_udp_protocol-client-2 exited with code 0

PS C:\Users\crrgr\Documents\College\ComputerNetworks\networks_udp_prot
col> docker-compose run interactive_client sh
[+] Running 11/0
- Container ingress Running
- Container networks_udp_protocol-worker-2 Running
- Container networks_udp_protocol-worker-1 Running
- Container networks_udp_protocol-worker-10 Running
- Container networks_udp_protocol-worker-5 Running
- Container networks_udp_protocol-worker-9 Running
- Container networks_udp_protocol-worker-3 Running
- Container networks_udp_protocol-worker-6 Running
- Container networks_udp_protocol-worker-4 Running
- Container networks_udp_protocol-worker-8 Running
- Container networks_udp_protocol-worker-7 Running
Please select a file to request or 'exit' if you are finished: test.txt
Requesting file test.txt
Received file.
Please select a file to request or 'exit' if you are finished: test_vid
eo.mp4
Requesting file test_video.mp4
Received file.
Please select a file to request or 'exit' if you are finished: exit
PS C:\Users\crrgr\Documents\College\ComputerNetworks\networks_udp_prot
col>

```

Figure 10: This shows the file transfer working in VSCode. In the left terminal, docker-compose is running, and allowing the system to be up. The randomised clients are requesting and receiving files. In the right terminal, the interactive client is being run, and the user is requesting and receiving files, which can be seen in the file explorer to the left under interactive volume.

Containers [Give Feedback](#) [Learn more](#)

A container packages up code and its dependencies so the application runs quickly and reliably from one computing environment to another. [Learn more](#)

Showing 14 items

Search


















































	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	 networks_udp_protocol 13 containers	-	Running (13/1)	-	-	  
<input type="checkbox"/>	 ingress 8441004187a0 	networks_udp_protocol-ingress:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 interactive_client-1 d0f1ef912eed 	networks_udp_protocol-interactive_client:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 tcpdump-1 c77ffc426114 	kaazing/tcpdump:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-1 a4afd50890c9 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-10 5f1228325af2 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-2 aebb4fa02c66 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-3 ff5ed962d4c8 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-4 fe4d213e6261 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  
<input type="checkbox"/>	 worker-5 4a7b0ce4e365 	networks_udp_protocol-worker:latest	Running	-	8 minutes ago	  

Figure 11: This screenshot shows my solution working in Docker Desktop. Containers for ingress, the interactive client, and workers can be seen under the overall docker compose combination called networks_udp_protocol.

This report has described my attempt at a solution to address file transfer over a UDP protocol. The implementation of basic file transfer is standard and generally simple. The implementation of packetising for large files is standard enough and works well. The ARQ implementation is unorthodox but works exceptionally well for medium sized files (70kB - 15MB), but reduces significantly in efficiency for files larger than that. The description of the implementation in this document provides the essential components and design elements required for my solution to work, as well as demonstrating the execution of the solution in an example topology with a potentially large number of clients.

6 Reflection

I found the intention of this assignment difficult to understand at first, and spent a significant amount of time trying to understand what was required at a basic level. However, once I understood the intention of the project, I found it a very good opportunity to solve a complex problem in my own way. However, I believe it would have been better to create more documentation while creating the protocol. I appreciated the assignment as an opportunity to test out what I thought might be an interesting naive ARQ. I now have a far better understanding of not only the positives and downfalls of UDP, but also of ARQs and flow control in networking.

Overall, I would estimate this assignment took me about 40 hours, although I was not tracking this, which I will do for the next assignment.