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

CSU33031 Computer Networks
Assignment #1: Protocols

Ryan Idowu, Std# 20333751

October 19,2022

Contents

1 Introduction **2**

2 Theory of Topic **3**
2.1 Client 3
2.2 Server 3
2.3 Worker 3
2.4 Communication and Packet Description 4

3 Implementation **6**
3.1 Client 6
3.2 Server 8
3.3 Worker 11
3.4 Packet Encoding 13

4 Summary & Reflection **14**

References **15**

1 Introduction

The focus of this assignment is to expand my knowledge of protocol development and how header information is used to support a protocol.

This assignment involves the implementation of a protocol that provides a mechanism to retrieve files from a service based on User Datagram Protocol (UDP) datagrams. My approach makes use of several actors: a client, a server, and multiple workers to implement this functionality. Additionally, the transfer of packets within my protocol is aided by the header information contained within each packet.

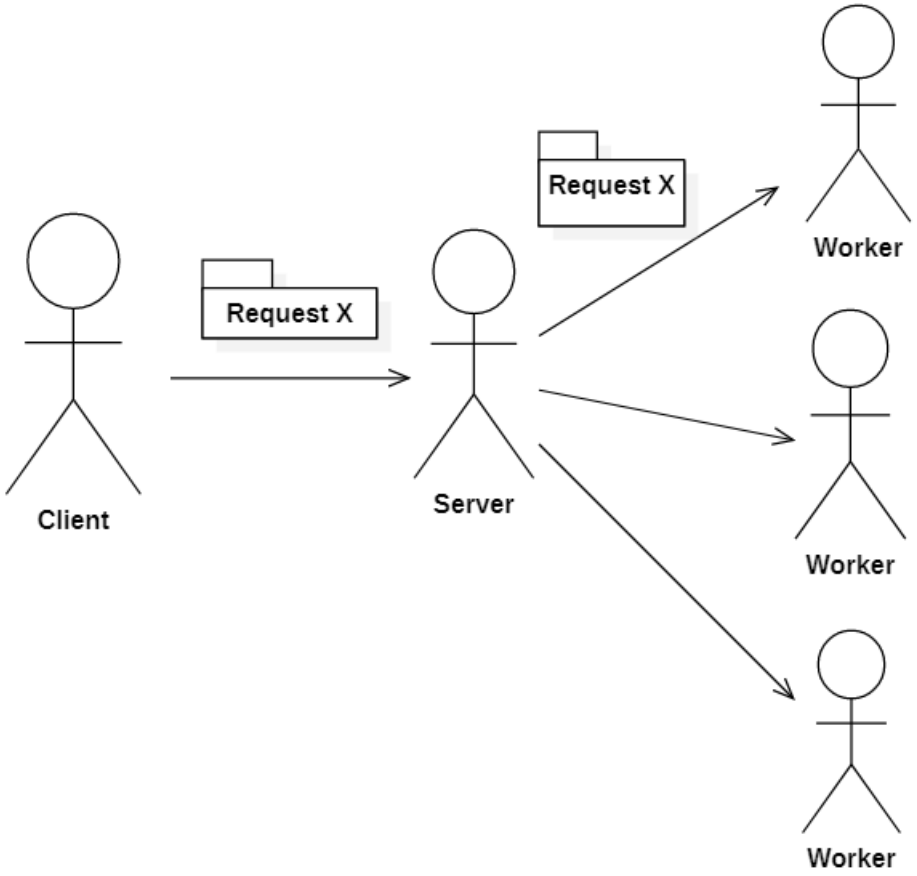

Figure 1: The assignments topology, A client sends a request to a server which distributes requests to relevant workers.

2 Theory of Topic

The assignment involves three types of actors: a user-controlled client, a server, and a number of workers. The way in which each actor interacts is displayed in figure 2. As well as this the assignment requires the packets header information to assist the transfer of files.

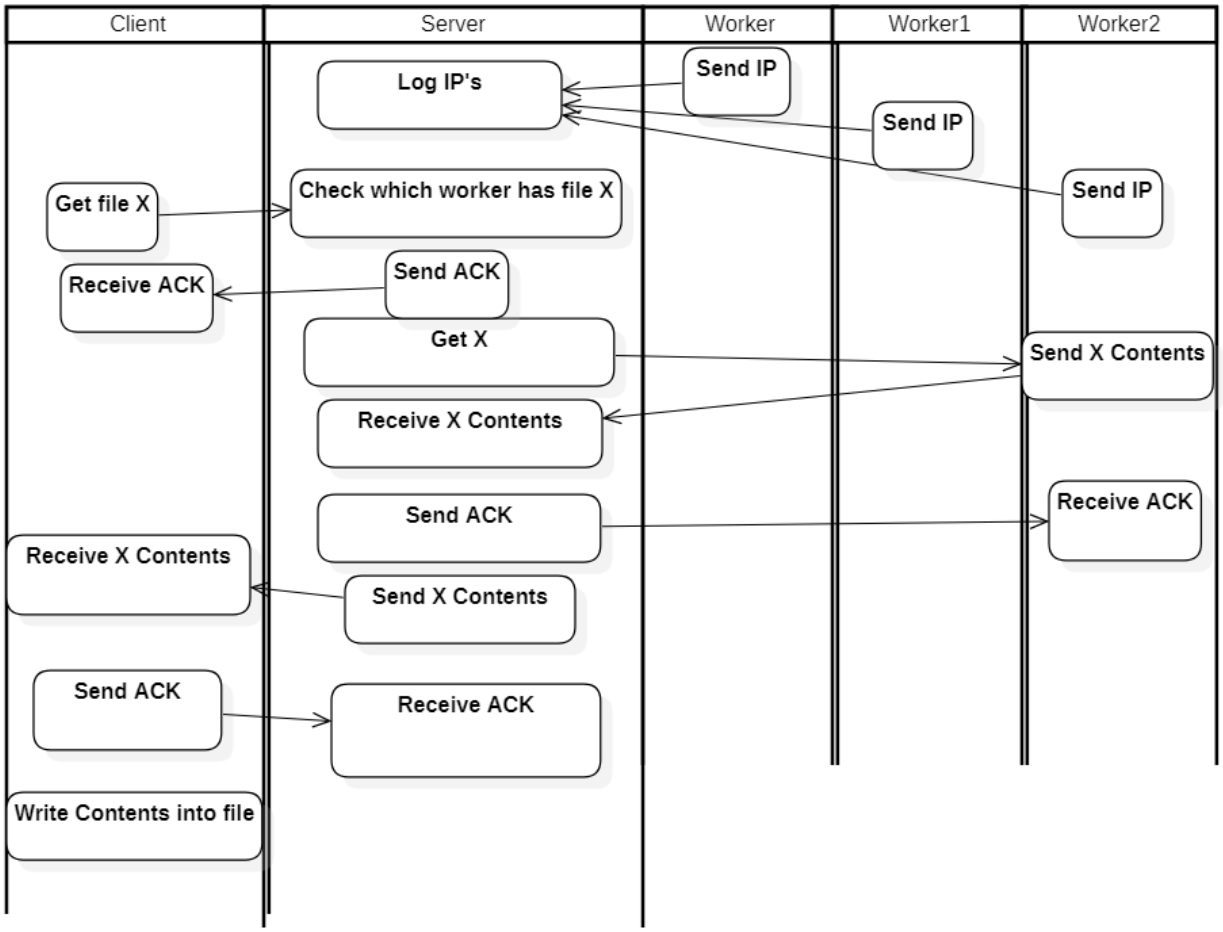

Figure 2: An activity diagram representing the interactions between each actor and the ideal flow of the program.

2.1 Client

The client's job is to request and receive files from the server. In addition to this it is also responsible for ensuring its requests are received by the server and that it receives all packets from the server in the correct order.

2.2 Server

The server's job is to forward requests from the client to the relevant workers and forward files from the workers to the client. Like the client it is also responsible for ensuring there is no packets loss in each of its data transfers.

2.3 Worker

The role of each worker is to send requested files to the server. Similar to both the client and the server it is also responsible for ensuring there is no packet loss in each of its data transfers

2.4 Communication and Packet Description

Each actor communicates through packet transfers. In each packet sent from one actor to another there is information in its header supports the identification of the requested item, assists in the correct transfer of files and provides information to the receiver about the senders IP address. A Stop-and-wait Automatic Repeat Request (ARQ) protocol was implemented to provide reliable transfers within the assignment.

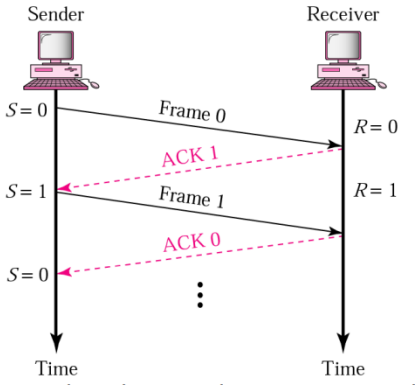

Figure 3a: [1]Diagram showing the flow of the Stop-and-wait ARQ protocol when there is no packet loss. The sender sends a packet with sequence number X, the receiver sends back an acknowledgement (ACK) with sequence number X + 1 to communicate it is expecting a packet with sequence number X + 1 next.

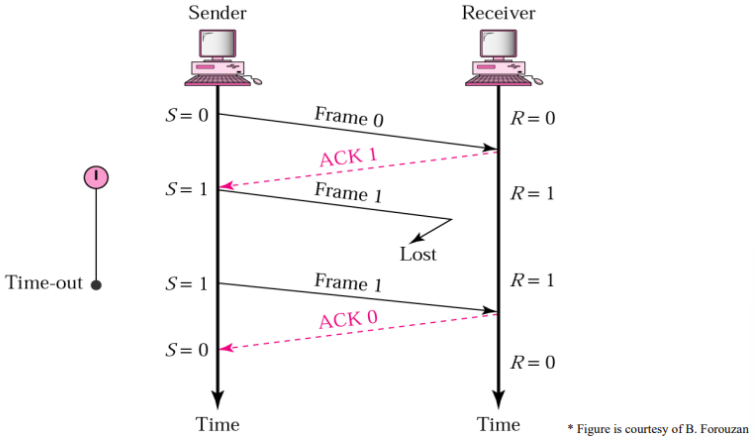

Figure 3b: [1]Diagram showing the flow of the Stop-and wait ARQ protocol when there is packet loss. If the sender does not receive an ACK before a timeout, it will resend the same packet.

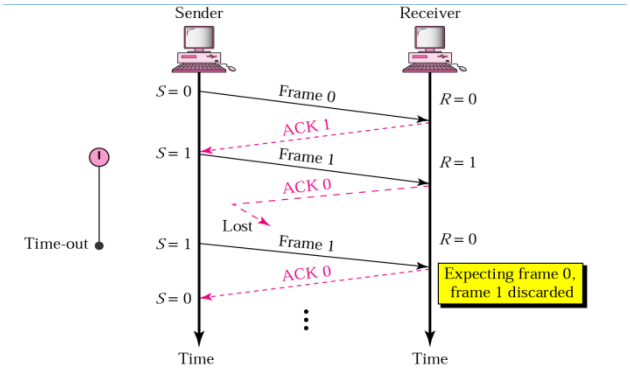

Figure 3c: [1]Diagram showing the flow of the Stop-and wait ARQ protocol when there is a lost ACK. If the sender does not receive an ACK before a timeout, it will resend the same packet. The receiver will discard any duplicate packets and resend an ACK to the sender

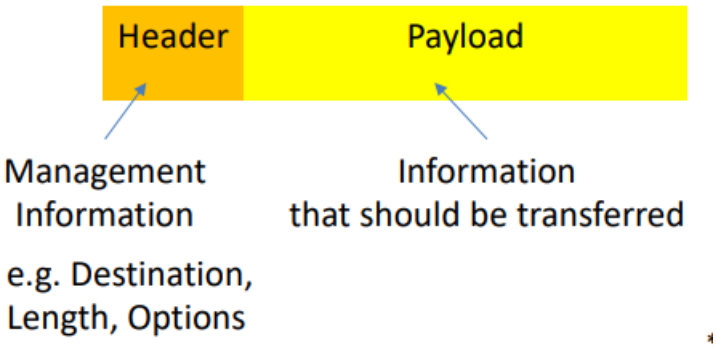

Figure 4: [1]Diagram showing the basic structure of a packet. The header of a packet contains management information such as source address, destination address and packet sequence numbers. The payload of a packet contained the information being transferred from the sender to the receiver.

3 Implementation

My solution is implemented using multiple python programs and the python library socket for sending and receiving UDP datagrams

3.1 Client

The client is implemented in the client.py file.

```
msgFromClient = input('Enter request file')
   # Send request to Server
   resend = True
   UDPClientSocket.sendto(bytesToSend, serverAddressPort)
```

Listing 1: User Interaction within client.py. The user requests a file, and the client packetizes the request and sends it to the server.

```
try:
        # Set timeout for retransmission of request
        UDPClientSocket.settimeout(10)
.....
.....
# Server doesnt send result before timeout - resend request
      except timeout:
        UDPClientSocket.sendto(bytesToSend, serverAddressPort)
```

Listing 2: Client resends request if it times out

```
sender, serverSeqNum, data = packetOperations.depacketize(packetFromServer)
        seqNum = 0
        f = open("Client {}".format(msgFromClient), 'w')
        acknowledge = packetOperations.packetize("Client", seqNum,
                                    "received packet {} expecting packet {} ".format(
                                      seqNum, seqNum + 1))
        while data:
          if data == 'end':
            print("File Downloaded")
            f.close()
            acknowledge = packetOperations.packetize("Client", seqNum + 1,
                                    "received packet {}".format(
                                      seqNum))
            UDPClientSocket.sendto(acknowledge, serverAddressPort)
            break
          # Check if packet sent is expected packet
          if int(serverSeqNum) == seqNum:
            f.write(data)
            seqNum += 1
            # Send ack with expected packet
            acknowledge = packetOperations.packetize("Client", seqNum,
                                    "received packet {} expecting packet {} ".format(
                                      seqNum - 1, seqNum))
          UDPClientSocket.sendto(acknowledge, serverAddressPort)
          packetFromServer = UDPClientSocket.recvfrom(bufferSize)
          name, serverSeqNum, data = packetOperations.depacketize(packetFromServer)
```

Listing 3: Client receiving the files contents and writing it into a new file. Using the stop and wait ARQ protocol, Client checks for the sequence number of each packet and acts accordingly. If the sequence number is the same as the sequence number expected by the client, the client will send an acknowledgement to the server and write the contents into the new file. If the sequence number is different to the expected sequence number, the client resends the previous acknowledgement and discards the content.

3.2 Server

The server's functionality Is implemented in the server.py file.

```
# Contents of each worker
workerContents = [['a.txt', 'b.txt', 'c.txt'],
                  ['d.txt', 'e.txt', 'f.txt'],
                  ['g.txt', 'h.txt', 'i.txt']]
```

Listing 4: Contents of each worker is hard coded into the server

```
for i in range(NUMBER_OF_WORKERS):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    workerNum = bytesAddressPair[0].decode()
    workersIP[workerNum] = bytesAddressPair[1]
    print("Worker: {} registered!".format(workerNum))
```

Listing 5: Server waits for each worker to register with it. Each workers IP address added to an arraylist

```
# Find worker with file
    workerIndex = 'null'
    for files in workerContents:
        if filename in files:
            workerIndex = str(workerContents.index(files))
            print(workerIndex)
```

Listing 6: Server receives filename request from client and checks for which worker has the file.

```
# Set timeout for retransmission of request
        UDPServerSocket.settimeout(10)
        # Receive packet from worker
        packetFromWorker = UDPServerSocket.recvfrom(bufferSize)
        resend = False
        UDPServerSocket.settimeout(None)
        sender, workerSeqNum, data = packetOperations.depacketize(packetFromWorker)
        print(data)
        seqNum = 0
        f = open("Server {}".format(filename), 'w')
        acknowledge = packetOperations.packetize("Server", seqNum,
                                "received packet {} expecting packet {} ".format(
                                  seqNum , seqNum + 1))
        # Receive file from worker
        while data:
          if data == 'end':
            print("File Downloaded")
            acknowledge = packetOperations.packetize("Server", seqNum + 1,
                                    "received packet {}  ".format(
                                      seqNum))
            UDPServerSocket.sendto(acknowledge, worker)
            f.close()
            break
          # Check if packet received is expected packet
          if sender != "Client":
            if int(workerSeqNum) == seqNum:
              f.write(data)
              seqNum += 1
              # Send ack with next expected packet
              acknowledge = packetOperations.packetize("Server", seqNum,
                                    "received packet {} expecting packet {} ".format(
                                      seqNum - 1, seqNum))
          UDPServerSocket.sendto(acknowledge, worker)
          packetFromWorker = UDPServerSocket.recvfrom(bufferSize)
          sender, workerSeqNum, data = packetOperations.depacketize(packetFromWorker)
```

Listing 7: The server then requests and receives the file from the worker in the exact manner as the Client in Listing 3.

```
# Send file to Client
    f = open("Server {}".format(filename), 'r')
    data = f.read(bufferSize -24)
    toSend = packetOperations.packetize("Server", 0, data)
    UDPServerSocket.sendto(toSend, address)
    clientSeqNum = 0
    UDPServerSocket.settimeout(2)
    while int(clientSeqNum) < int(seqNum) + 1:
        try:
            # Receive ack from client
            packetFromClient = UDPServerSocket.recvfrom(bufferSize)
            name, clientSeqNum, message = packetOperations.depacketize(packetFromClient)
            if int(clientSeqNum) > int(seqNum) + 1:
                break
            if int(clientSeqNum) == int(seqNum):
                toSend = packetOperations.packetize("Server", clientSeqNum, "end")
                UDPServerSocket.sendto(toSend, address)
            else:
                toSend = packetOperations.packetize("Server", clientSeqNum, data)
                print(message)
                UDPServerSocket.sendto(toSend, address)
                print("sending ...")
                data = f.read(bufferSize - 24)
        except timeout:
            if clientSeqNum == seqNum:
                break
            UDPServerSocket.sendto(toSend, address)
            print("timeout resending ...")
    # Send end packet to server
    UDPServerSocket.settimeout(None)
    print("Sent")
```

Listing 8: Server sends files contents to Client. In line with the Stop-and-wait ARQ protocol in the event of a timeout the server will resend the packet with the sequence number of the last acknowledgement received. If the sequence number of the ack received is equal to or greater than the number of packets in the file the Server sends a message ‘end’ to the client indicating the end of the file.

3.3 Worker

The workers functionality are implemented in the worker.py, worker1.py and worker2.py files

```
class Worker:
    def __init__(self, num):
        self.num = num
        bytesToSend = str.encode(str(num))
        # Create a udp socket at worker side
        self.workerSocket = socket(AF_INET, SOCK_DGRAM)
        # Send address to server
        self.workerSocket.sendto(bytesToSend, serverAddressPort)
        print("UDP worker{} up and listening".format(num))
```

Listing 9: The worker classes initialization. The worker creates a UDP socket and registers with server by sending its number to the server

```
def listen(self):
        # Listen for incoming datagrams
        while True:
            # Receive request from Server
            serverRq = self.workerSocket.recvfrom(bufferSize)
            name, seqNum, filename = packetOperations.depacketize(serverRq)
            address = serverRq[1]
            print('Message from Server {}'.format(filename))
            # Send ack to Server
            # self.workerSocket.sendto(ack, address)
            # Send file to Server
            f = open(filename, 'rb')
            data = f.read(bufferSize -24)
            toSend = packetOperations.packetize("Worker", 0, data.decode())
            self.workerSocket.sendto(toSend, address)
            serverSeqNum = 0
            endSeq = 0
            #Get length of file in sequence numbers
            while data:
                print(data)
                endSeq += 1
                data = f.read(bufferSize)
```

Listing 10: The workers listen function, this function begins by receiving a request from the server, opening the file being requested and checking the number of packets needed to send the file

```
        self.workerSocket.settimeout(2)
        while True:
            try:
                # Receive ack from server
                packetFromServer = self.workerSocket.recvfrom(bufferSize)
                name, serverSeqNum, message = packetOperations.depacketize(packetFromServer)
                print(message)
                print("data: {}".format(data))
                if int(serverSeqNum) > endSeq:
                    break
                if int(serverSeqNum) == endSeq:
                    toSend = packetOperations.packetize("Worker", serverSeqNum, "end")
                    self.workerSocket.sendto(toSend, address)
                else:
                    toSend = packetOperations.packetize("Worker", serverSeqNum, data.decode())
                    self.workerSocket.sendto(toSend, address)
                    print("sending ...")
                    data = f.read(bufferSize -24)
            except timeout:
                if int(serverSeqNum) > endSeq:
                    break
                self.workerSocket.sendto(toSend, address)
                print("timeout resending ...")
        # Send end packet to server
        self.workerSocket.settimeout(None)
        print("Sent")
```

Listing 11: Within the listen function, worker sends files contents to Server in the exact manner as the server in Listing 8

3.4 Packet Encoding

The packet encoding is done using two functions in the PacketOperations.py file.

```
def packetize(name, seqNum, message) :
    packet = name + "|||" + str(seqNum) + "|||" + message
    return str.encode(packet)
```

Listing 12: The packetize function, converts a message in to 'packet' format by adding a pseudo-header before the message being sent. The pseudo-header contains the senders name and the packets sequence number. The functions adds a maximum of 24 bytes to the message being sent

```
def depacketize(packet):
    packetMessage = packet[0].decode()
    name, seqNum, message = packetMessage.split("|||")
```

Listing 13: The depacketize function, decodes a packet in the packet format described in Listing 12 and returns name of sender, sequence number and the message contained in the packet.

Note on Python socket library[2]

In addition to the extra pseudo-header added in the packetize function, python's socket library provides more header information such as destination address, source address, checksum for error checking and much more information

4 Summary & Reflection

Overall, I was satisfied with my solution. I believe it expanded my knowledge of protocol development and how header information is used to support a protocol. I also believe the project has elevated my standard of code and am grateful for the project exposing me to new features in both Python and Docker.

Things that went well

The protocol succeeds in providing a mechanism to retrieve files with the use of UDP datagrams. The protocol allows the transfer of files using 1 or more packets. As well as this the protocol encodes header information to assist in the data transfers between the three types of actors: client, server, and workers. The protocol also takes measures to prevent packet loss.

Improvements I would have liked to implement

1. Stop-and-wait ARQ, while effective in addressing packet loss, is an inefficient flow control protocol in comparison to others such as Selective repeat ARQ or Go-Back-N ARQ. As Stop-and-wait only allows for one packet to be outstanding from the sender at a time, its efficiency isn't very high. I would have liked to include a more efficient protocol for flow control
2. The worker classes were all in separate files. I would have preferred to have them run concurrently as nodes within the server class however I didn't have the time to implement this functionality.
3. Due to each worker needing to register with the server, the protocol requires that the server.py be run first then worker.py, worker1.py, worker2.py and finally client.py. It would have been better if there was a way around this.

To conclude I was happy with how my solution turned out. A lot of things went well for it, and I believe it sufficiently satisfies the assignments requirements. On the other hand, I still believe there is a lot of room for improvement within the protocol

References

[1] Trinity College. CSU33031 Lecture Slides, visited Sep 2021.

[2] Python, Python socket library. https://docs.python.org/3/library/socket.html, visited Sep 2021.