NAME: Debadrita Roy

CLASS: BCSE-III

GROUP: A1

ASSIGNMENT NUMBER: 2

PROBLEM STATEMENT: Implement three data link layer protocols, Stop and Wait, Go Back N Sliding Window and Selective Repeat Sliding Window for flow control. Sender, Receiver and Channel all are independent processes. There may be multiple Transmitter and Receiver processes, but only one Channel process. The channel process introduces random delay and/or bit error while transferring frames. Define your own frame format or you may use IEEE 802.3 Ethernet frame format.

DEADLINE: 19th November, 2021

DATE OF SUBMISSION: 19th November, 2021

DESIGN

The purpose of the program is to simulate the real-world network environment and design and implement data link layer protocols to control the flow of data frames from the sender to the receiver. The required programs are written in Python 3.

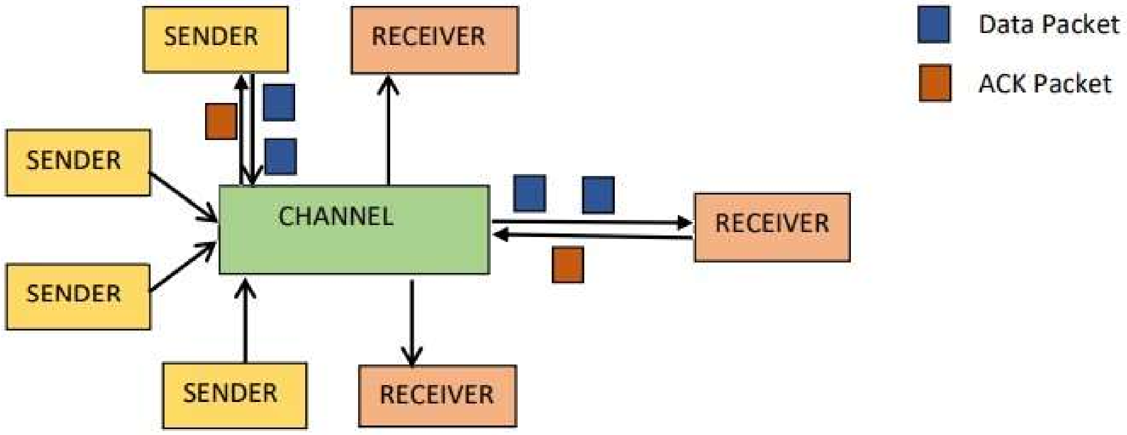
We use sockets to establish the communication link between the sender and the receiver through the channel. The channel (**channel.py**) listens for connection requests from senders and receivers. When a sender is ready to send, it acquires a list of the receivers currently connected to the channel and chooses one of them. Then a link is established between the sender-receiver pair via the channel. According to user's choice, a flow control protocol is used for sending the packets. The different flow control protocols are implemented in different classes, each protocol having its own sender and receiver classes, these classes are available to **sender.py** and **receiver.py**. There is functionality on both sender.py and receiver.py so that the user can stop the running of the processes when transmitting/receiving is completed. There are other classes assisting in the running of the above classes, for creating a packet in IEEE 802.3 format and extracting data or other parts like destination address, source address, sequence number, etc or whether there is error in the packet or not, or for error detection.

Packet Structure:

Preamble	SFD	Destination address	Source address	Length or type	Data and padding	CRC
7 bytes	**1 byte**	**6 bytes**	**6 bytes**	**2 bytes**	**46 bytes**	**4 bytes**

We use the IEEE 802.3 frame format for the packets. Preamble consists of 56 bits of alternating 0's and 1's. SFD is the start frame delimiter, containing 10101011. CRC-32 is used for error detection. Length or type contains 1 byte for type (0 for data, 1 for ack, 2 for nak) and 1 byte for the sequence number of the packet.

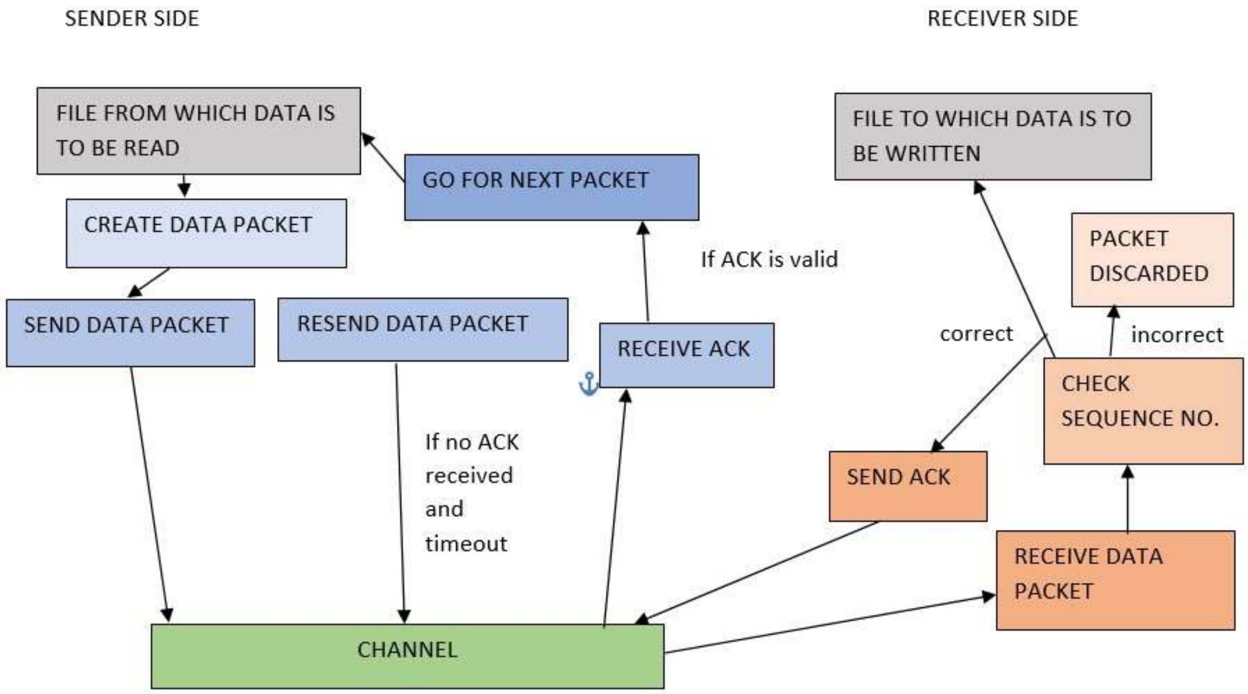
Structure Diagram:



There are multiple senders and multiple receivers, at a time only one sender-receiver pair can communicate.

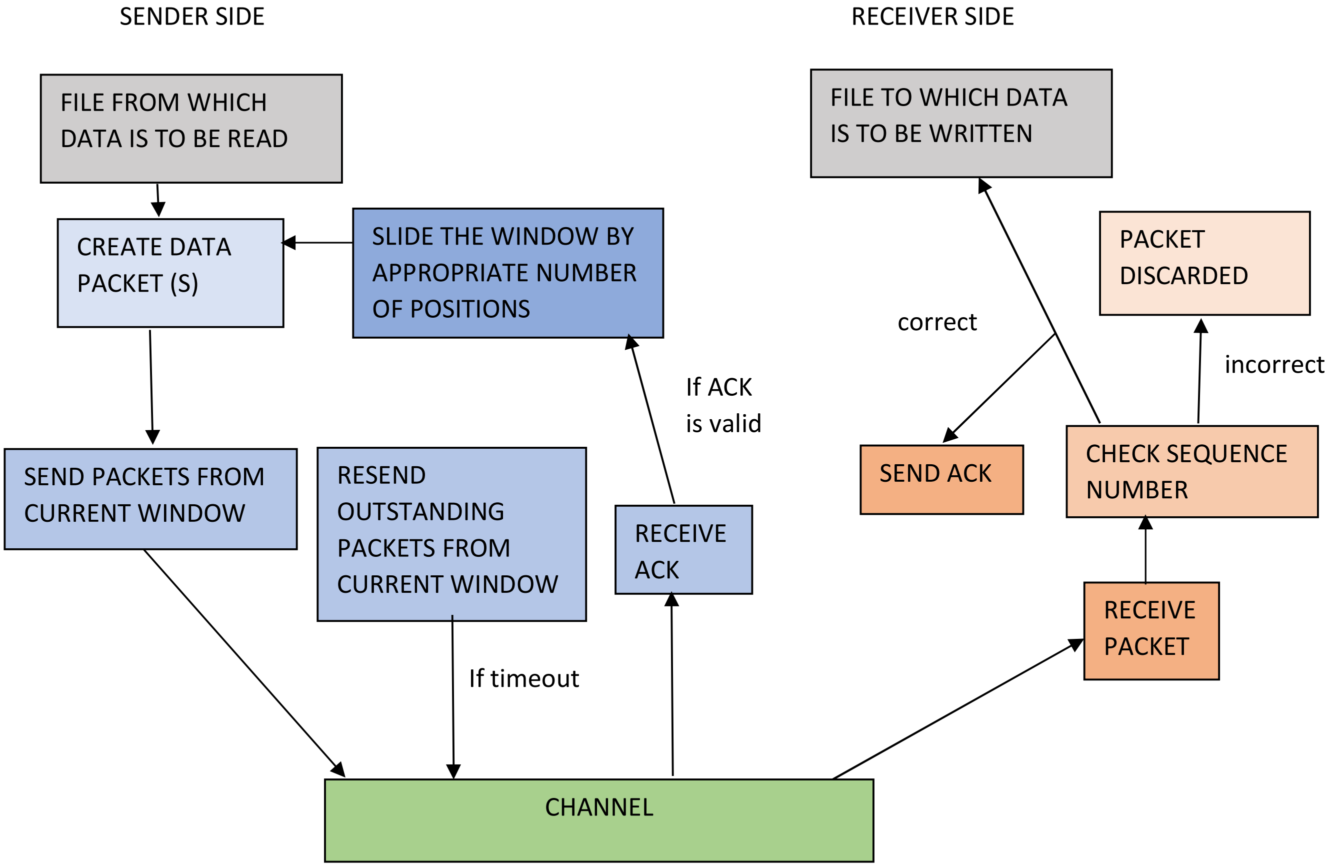
Stop and wait protocol:

Here, the sender sends a packet then waits for the acknowledgement. If acknowledgement is received before timeout, then the next packet is sent. If correct acknowledgement (there might be errors or ACK of any previous packet may arrive after timeout) is not received before timeout, then the packet is retransmitted and timer is once again started. At the receiver end, if the correct packet (may have errors or may be duplicate) is received, then it sends an ACK to the sender and asks for the next packet.



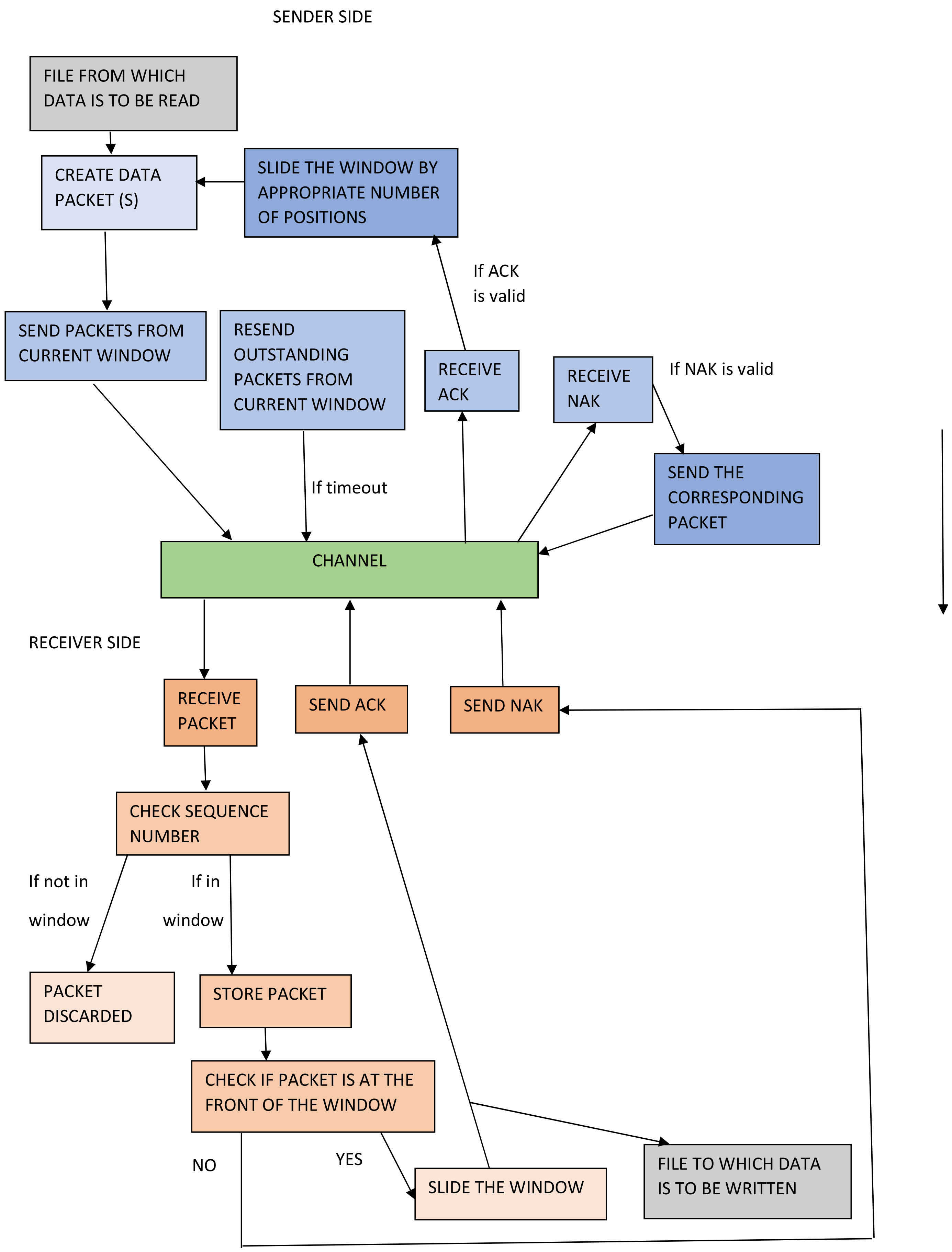
Go Back N Protocol:

Here the sender sends a data packet to the receiver via the channel and starts the timer. It then proceeds to send packets until the total number of packets sent equals the window size. The receiver window has a size of one. It checks the sequence number of the data packet received and if the sequence number is valid then it sends an ACK to the sender, if the sequence number is invalid then the packet is discarded. When the sender receives an ACK say 3 then it concludes that the data packets 0, 1 and 2 are received by the receiver. If timeout happens then it resends the frames for which ACK has not been received.


SENDER SIDE
RECEIVER SIDE
FILE FROM WHICH DATA IS TO BE READ
FILE TO WHICH DATA IS TO BE WRITTEN
CREATE DATA PACKET (S)
SLIDE THE WINDOW BY APPROPRIATE NUMBER OF POSITIONS
PACKET DISCARDED
correct
incorrect
If ACK is valid
SEND PACKETS FROM CURRENT WINDOW
RESEND OUTSTANDING PACKETS FROM CURRENT WINDOW
RECEIVE ACK
SEND ACK
CHECK SEQUENCE NUMBER
RECEIVE PACKET
If timeout
CHANNEL

Selective Repeat Protocol:

Here both the sender window and receiver window are of the same size. The sender sends a data packet to the receiver via the channel and starts the timer. It then proceeds to send packets until the total number of packets sent equals the window size. The receiver side also maintains a timer. If a packet has arrived that has a sequence number which is within the window, then that packet is stored (unlike go back n where the packet would be discarded if it was not in order). When the receiver timer finishes then it checks the receiving window. If any packet in the window has not arrived then it sends a NAK to the sender who then resends the frame.


SENDER SIDE
FILE FROM WHICH DATA IS TO BE READ
CREATE DATA PACKET (S)
SLIDE THE WINDOW BY APPROPRIATE NUMBER OF POSITIONS
If ACK is valid
SEND PACKETS FROM CURRENT WINDOW
RESEND OUTSTANDING PACKETS FROM CURRENT WINDOW
RECEIVE ACK
RECEIVE NAK
If NAK is valid
SEND THE CORRESPONDING PACKET
If timeout
CHANNEL
RECEIVER SIDE
RECEIVE PACKET
SEND ACK
SEND NAK
CHECK SEQUENCE NUMBER
If not in window
If in window
PACKET DISCARDED
STORE PACKET
CHECK IF PACKET IS AT THE FRONT OF THE WINDOW
NO
YES
SLIDE THE WINDOW
FILE TO WHICH DATA IS TO BE WRITTEN

Input Format: The sender and the receiver names are taken as input. Also, which protocol is to be used is also chosen by the user. The data to make the packets are read from a text file specified by the program. The user also has the option of terminating the sender and/or receiver processes by typing ‘1’ when prompted.

Output Format: The various stages of the transmission/reception of the packets are displayed on the terminal so that we can keep track of the program running. The data received by the receiver are stored in text files specified by the program. The analysis of the different protocols are stored in separate text files specified by the program.

IMPLEMENTATION

sender.py

It contains the functionality for establishing and maintaining a connection with the channel and directing the program control to the respective flow control class chosen for sending the packets.

main()

Method Description: The main() function sets up the connection between the sender and the channel using socket programming and also accepts the user’s choice for the flow control mechanism to be used.

Code Snippet:

```
if __name__=='__main__':
  print('[CLIENT] : ')
  print('1.Stop and wait\n2.Go back N\n3.Selective repeat\n')
  fcpType = int(input('Enter choice(1-3): '))
  if fcpType>3 or fcpType<1:
    fcpType = 1
  fcpType -= 1
  SERVER_IP='127.0.0.1'
  SERVER_PORT=1232
  with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as client:
    client.connect((SERVER_IP, SERVER_PORT))
    msg =  client.recv(1024).decode()
    print("From Channel :" , msg, end='')
    name=input()
    client.sendall (bytes(name,'UTF-8'))
    address = client.recv(1024).decode()
    senderAddress = int(address)
    while True:
      print('1.Send data\n2.Close\n')
      choice=int(input('Enter choice (1-2) : '))
      if choice==1:
        client.send(str.encode("request for sending"))
      elif choice==2:
        client.send(str.encode("close"))
        break
      inputs=[client]
      output=[]
      readable,writable,exceptionals=select.select(inputs,output,inputs,3600)
      for s in readable:
        data=s.recv(1024).decode()
        if data== "No client is available":
          print(data)
          break
        elif choice == 1:
          file_name='test.txt'
          receiver_list=data.split('$')
          print('Available clients-----')
          for index in range(0,len(receiver_list)):
            print((index+1),'.',receiver_list[index])
          choice=int(input('\nYour choice : '))
          choice-=1
          while choice not in range(0, (len(receiver_list))):
            choice=int(input('Invalid Input...try again : '))
            choice-=1
          s.send(str.encode(str(choice)))
          receiverAddress = int(s.recv(1024).decode())

my_sender=senderList[fcpType].Sender(client,name,senderAddress,receiver_list[index],receiverAddress,file_name)
          my_sender.transmit()
          data=s.recv(1024)
          data=data.decode()
          print(data)
      if not (readable or writable or exceptionals):
        continue
```

receiver.py

It contains the functionality for establishing and maintaining a connection with the channel and directing the program control to the respective flow control class chosen for receiving the packets.

main()

Method Description: The main() function sets up the connection between the receiver and the channel using socket programming and also accepts the user's choice for the flow control mechanism to be used.

Code Snippet:

```
if __name__=='__main__':
  print('Choose flow-control protocol :-')
  print('1.Stop and wait\n2.Go back N\n3.Selective repeat\n')
  fcpType = int(input('Enter your choice (1-3) :'))
```

```
    if fcpType>3 or fcpType<1:
        fcpType = 1
    fcpType -= 1
    SERVER_IP='127.0.0.1'
    SERVER_PORT=1232
    with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as client:
        client.connect((SERVER_IP, SERVER_PORT))
        msg =  client.recv(1024).decode()
        print("From channel :" , msg, end='')
        name=input()
        client.sendall (bytes(name,'utf-8'))
        address = client.recv(1024).decode()
        senderAddress = int(address)
        while True:
            print('1.Receive data\n2.Close\n')
            choice=int(input('Enter choice : '))
            if choice!=1:
                client.send(str.encode("close"))
                break
            inputs=[client]
            output=[]
            # Wait until any input/output event or timeout occurs
            readable,writable,exceptionals=select.select(inputs,output,inputs,3600)
            for s in readable:
                data=s.recv(1024).decode()
                if data== "No client is available":
                    print(data)
                    break
                elif choice == 1:
                    print('Receiving data-----')
                    file_name=''
                    if fcpType == 0:
                        file_name='SWARQ_rec.txt'
                    elif fcpType == 1:
                        file_name='GBN_rec.txt'
                    else:
                        file_name='SR_rec.txt'
                    receiverAddress = int(data)
                    s.send (bytes("start", 'utf-8'))

my_receiver=receiverList[fcpType].Receiver(client,name,senderAddress,receiverAddress,file
_name)
                    my_receiver.startReceiving()
            if not (readable or writable or exceptionals):
                continue
```

Analysis.py

It contains functionality for storing the values for the performance parameters in order to compare between the different flow control mechanisms. The values are stored in a file with given filename.

errordetect.py

This module is from Assignment 1. It contains functions for detecting errors according to different error detection schemes as well as making codewords according to the different schemes. We use the functions pertaining to CRC for making up the packets and detecting errors in the packets.

PacketManager.py

The Packet class contains a constructor which initializes the packet details, functions to make a packet and functions to return packet details such as type (data/ack/nak), sequence number, data and function to check whether the packet contains error or not.

Channel.py

This module contains the ConnectionThread class (subclass of Thread class) for the channel connections to the sender(s)/receiver(s) and functions to inject random error in the packet (from channel.py in Assignment 1) as well as decide whether the packet is to be sent immediately without error, delayed or errorified.

def process_packet(p)

Method Description: This decides what happens to the packet, p, in the channel.

Code Snippet:

```
def process_packet(p):
  flag=int(random.random()*10)

  if flag<=5:   # original packet sent
    return p
  elif flag<=8:  # introduce error
    return inject_random_error(p)
  else:  # introduce delay
    time.sleep(0.5)
    return p
```

ConnectionThread(Thread) class

This class sets up the connections between the channel and the senders/receivers and runs them.

def __init__ (self, clientSocket, clientAddress)

Method Description: This is the constructor to initialize the client details.

Code Snippet:

```
def __init__ (self, clientSocket, clientAddress):
  Thread.__init__ (self)
  self.csocket = clientSocket
  self.caddr = clientAddress
```

```
    print (clientAddress,' connected to channel')
```

def setConnection(self)

Method Description: This function sets the connection between the channel and the client.

Code Snippet:

```
def setConnection (self):
    availableClients = []
    availableClientNames = []
    for address in client_map:
        if address != self.caddr and client_map[address][2] is None:
            availableClients.append(address)
            availableClientNames.append(client_map[address][1])

    if len(availableClients) == 0:
        self.csocket.send("No client is available".encode('utf-8'))
    else:
        self.csocket.send(bytes('$'.join(availableClientNames).encode('utf-8')))
        choice = int(self.csocket.recv(1024).decode())

        my_lock.acquire()
        raddr = availableClients[choice]

        if client_map[raddr][2] is None:
            rsocket = client_map[raddr][0]
            client_map[raddr][2] = self.caddr
            client_map[raddr][3] = 384
            client_map[self.caddr][2] = raddr
            client_map[self.caddr][3] = 576
            self.csocket.send (str(raddr[1]).encode('utf-8'))
            rsocket.send (str(self.caddr[1]).encode('utf-8'))
            print(self.caddr,"is sending data to",raddr)
        else:
            print("receiver is busy..so data cannot be sent at the moment")
        my_lock.release()
```

def revokeConnection(self)

Method Description: This function closes the connection between the channel and the client.

Code Snippet:

```
def revokeConnection (self):
    my_lock.acquire()
    raddr = client_map[self.caddr][2]

    client_map[raddr][2] = None
    client_map[self.caddr][2] = None
    client_map[raddr][3] = client_map[self.caddr][3] = 1024
```

```
      self.csocket.send(str.encode("Sending completed"))
      print(self.caddr,' completed transmission of data to ',raddr)
      my_lock.release()
```

def run(self)

Method Description: This function contains the code for the thread to execute.

Code Snippet:

```
   def run (self) -> None:
      self.csocket.send("Successfully connected to channel.\nWrite your name: ".encode('utf-
8'))
      name=self.csocket.recv(1024).decode()
      self.csocket.send(str(self.caddr[1]).encode('utf-8'))
      client_map[self.caddr]=[self.csocket,name,None,1024]
      data = "open"

      while data!="close":
         inputBuffer = client_map[self.caddr][3]
         data = self.csocket.recv(inputBuffer).decode()
         if client_map[self.caddr][2] is None:
            if data == "request for sending":
               self.setConnection()
            else:
               pass
         else:
            rsocket = client_map[client_map[self.caddr][2]][0]
            if data == "start":
               rsocket.send(str.encode(data))
            elif data == "end":
               rsocket.send(str.encode(data))
               self.revokeConnection()
            else:
               newData = process_packet(data)
               if newData!= '':
                  rsocket.send(str.encode(newData))
      self.csocket.close()
      print ("Client at", self.caddr, "disconnected")
      client_map.pop(self.caddr)
```

main()

Method Description: The main() function sets up the connection between the clients (senders/receivers) and the channel using socket programming and starts those connection threads.

Code Snippet:

```
if __name__=='__main__':
   SERVER_IP='127.0.0.1'
   SERVER_PORT=1232
```

```
with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as server:
    server.setsockopt (socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((SERVER_IP,SERVER_PORT))
    server.listen(5)
    print("Channel is listening for connections")
    while True:
        conn,addr=server.accept()
        newThread = ConnectionThread (conn, addr)
        newThread.start()
```

The following modules contain classes for the sender portion and the receiver portion of the three protocols. For each sender, there are 3 threads—sending thread, ack/nak (as is the case) receiving thread and retransmitting thread. Each receiver sends ack/nak (as is the case) after receiving valid packets from the sender. There are constructors in each class to initialize the details. Selected portions of the code are shown here.

SenderSW.py

This is for the stop-and-wait protocol. The def transmit() function is responsible for starting each of the 3 threads and after the threads finish running, calls the function in Analysis.py to store the analysis. Code Snippets for the three threads are shown.

def sendData(self)

Method Description: This function sends the data packets to the channel.

Code Snippet:

```
def sendData(self):
    time.sleep(0.2)
    print("\n",self.name," starts sending data to ",self.receiver,"\n")
    file = open(self.fileName,'r')
    data_frame = file.read(defaultDataPacketSize)
    self.seqNo = 0
    self.pktCount = 0
    self.totalPktCount = 0
    while data_frame:
        if not self.pktSent:
            packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['data'], self.seqNo, data_frame)
            self.recentPacket = packet
            self.send_lock.acquire()
            self.connection.send(str.encode(packet.toBinaryString(46)))
            self.sentTime = time.time()
            self.pktSent = True
            self.seqNo = (self.seqNo+1)%2
            self.pktCount += 1
            self.totalPktCount += 1
            print("\nPacket ",self.pktCount," sent to channel")
            self.send_lock.release()
            data_frame = file.read(defaultDataPacketSize)
            if len(data_frame) == 0:
                break
    self.endTransmitting = True
    file.close()
```

def receiveAck(self)

Method Description: This function is for receiving ACK and checking its validity.

Code Snippet:

```
def receiveAck(self):
    time.sleep(0.2)
    while (not self.endTransmitting) or self.pktSent:
        if self.pktSent:
            received = self.connection.recv(384).decode()
            packet=PacketManager.Packet.build(received)
        else:
            continue

        if packet.getType() == 1:
            if not packet.hasError():
                if packet.seqNo == self.seqNo:
                    self.receiveTime = time.time()
                    rtt = (self.receiveTime - self.sentTime)
                    rttStore.append(rtt)
                    print("ACK ",packet.seqNo," received successfully\n")
                    self.pktSent = False
                else:
                    print("Wrong ACK")
            else:
                print("ACK has error...so discarded")
        else:
            print("Received packet is not an ACK")
```

def resendPackets(self)

Method Description: This function is for resending the data packets.

Code Snippet:

```
def resendPackets(self):
    time.sleep(0.2)
    while (not self.endTransmitting) or self.pktSent:
        if self.pktSent:
            current_time = time.time()
            waiting_time = (current_time-self.sentTime)
            if waiting_time > timeOut:
                self.send_lock.acquire()
                self.connection.send(str.encode(self.recentPacket.toBinaryString(46)))
                self.sentTime = time.time()
                print('Packet ',self.pktCount,' Resent')
                self.totalPktCount += 1
                self.send_lock.release()
```

SenderGBN.py

This is for the go-back-n protocol. The def transmit() function is responsible for starting each of the 3 threads and after the threads finish running, calls the function in Analysis.py to store the analysis. Code Snippets for the three threads are shown.

def sendData(self)

Method Description: This function sends the data packets to the channel.

Code Snippet:

```
  def sendData(self):
    time.sleep(0.2)
    print("\n",self.name," starts sending data to ",self.receiver,"\n")
    file = open(self.fileName,'r')
    data_frame = file.read(defaultDataPacketSize)
    while data_frame:
      if self.window_size<MAX_WINDOW_SIZE:
        packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['data'], self.end, data_frame)
        self.current_window[self.end] = packet
        self.window_write_lock.acquire()
        self.connection.send(str.encode(packet.toBinaryString(46)))
        print("\nPacket ",self.end," Sent")
        self.packet_timer[self.end] = time.time()
        self.end = ((self.end+1)%(MAX_WINDOW_SIZE+1))
        self.window_size += 1
        self.pktCount += 1
        self.totalPkt += 1
        data_frame = file.read(defaultDataPacketSize)
        self.window_write_lock.release()
      if len(data_frame) == 0:
        break
    self.endTransmitting = True
    file.close()
```

def receiveAck(self)

Method Description: This function is for receiving ACK and checking its validity.

Code Snippet:

```
def receiveAck(self):
time.sleep(0.2)
while (not self.endTransmitting) or (self.window_size>0):
  if self.window_size>0:
        received = self.connection.recv(384).decode()
        packet=PacketManager.Packet.build(received)
    else:
        continue
    if packet.getType() == 1:
        if not packet.hasError():
            if self.validACK(packet.seqNo):
                self.window_write_lock.acquire()
                while self.front!=packet.seqNo:
                    rtt = (time.time() - self.packet_timer[self.front])
                    rttStore.append(rtt)
                    print("Packet ",self.front," has reached successfully\n")
                    self.front = ((self.front+1)%(MAX_WINDOW_SIZE+1))
                    self.window_size -= 1
                self.window_write_lock.release()
            else:
                print("ACK is wrong...so discarded")
        else:
            print("ACK has error...so discarded")
    else:
        print("Received packet is not an ACK")
```

def resendPackets(self)

Method Description: This function is for resending the data packets.

Code Snippet:

```
def resendPackets(self):
time.sleep(0.2)
while (not self.endTransmitting) or (self.window_size>0):
    if self.window_size>0:
        current_time = time.time()
        front_waiting_time = (current_time-self.packet_timer[self.front])
        if front_waiting_time > timeOut:
            self.window_write_lock.acquire()
            temp=self.front
            while temp!=self.end:
                self.connection.send(str.encode(self.current_window[temp].packet))
                print('Packet ',temp,' Resent')
                self.packet_timer[temp] = time.time()
                temp=((temp+1)%(MAX_WINDOW_SIZE+1))
                self.totalPkt += 1
            self.window_write_lock.release()
```

SenderSR.py

This is for the selective repeat protocol. The def transmit() function is responsible for starting each of the 3 threads and after the threads finish running, calls the function in Analysis.py to store the analysis. Code Snippets for the three threads are shown.

def sendData(self)

Method Description: This function sends the data packets to the channel.

Code Snippet:

```
    def sendData(self):
        time.sleep(0.2)
        print("\n", self.name, " starts sending data to ", self.receiver, "\n")
        file = open(self.fileName, 'r')
        data_frame = file.read(defaultDataPacketSize)
        while data_frame:
            if self.window_size < MAX_WINDOW_SIZE:
                packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['data'],
                                            self.end, data_frame)
                self.current_window[self.end] = packet
                self.window_write_lock.acquire()
                self.connection.send(str.encode(packet.toBinaryString(46)))
                print("\nPacket ", self.end, " Sent to channel")
                self.packet_timer[self.end] = time.time()
                self.end = ((self.end + 1) % MAX_SEQUENCE_NUMBER)
                self.window_size += 1
                self.pktCount += 1
                self.totalPkt += 1
                data_frame = file.read(defaultDataPacketSize)
                self.window_write_lock.release()
            if len(data_frame) == 0:
                break
        self.endTransmitting = True
        file.close()
```

def receiveAck(self)

Method Description: This function is for receiving ACK and checking its validity.

Code Snippet:

```
def receiveAck(self):
time.sleep(0.2)
while (not self.endTransmitting) or (self.window_size > 0):
    if self.window_size > 0:
        received = self.connection.recv(384).decode()
        packet = PacketManager.Packet.build(received)
    else:
        continue
    if packet.getType() == 1:
        if not packet.hasError():
            if self.validACK(packet.seqNo):
                self.window_write_lock.acquire()
                while self.front != packet.seqNo:
```

```
                            rtt = (time.time() - self.packet_timer[self.front])
                            rttStore.append(rtt)
                            print("Packet ", self.front, " has reached successfully\n")
                            self.current_window[self.front] = 0
                            self.front = ((self.front + 1) % MAX_SEQUENCE_NUMBER)
                            self.window_size -= 1
                        self.window_write_lock.release()
                    else:
                        print("Wrong ACK...so discarded")
                else:
                    print("ACK has error...so discarded")
            elif packet.getType() == 2:
                if not packet.hasError():
                    if self.validACK(packet.seqNo):
                        self.window_write_lock.acquire()
                        if self.current_window[packet.seqNo] != 0:

self.connection.send(str.encode(self.current_window[packet.seqNo].toBinaryString(46)))
                            print('Packet ', packet.seqNo, ' resent from NAK')
                            self.packet_timer[packet.seqNo] = time.time()
                            self.totalPkt += 1
                        self.window_write_lock.release()
                    else:
                        print("Wrong NAK...so discarded")
                else:
                    print("NAK has error...so discarded")
            else:
                print("RECEIVED PACKET IS NOT AN ACK")
```

def resendPackets(self)

Method Description: This function is for resending the data packets.

Code Snippet:

```
def resendPackets(self):
time.sleep(0.2)
while (not self.endTransmitting) or (self.window_size > 0):
    if self.window_size > 0:
        current_time = time.time()
        oldest_packet = 0
        max_waiting_time = 0
        temp = self.front
        while temp != self.end:
            spent_time = (current_time - self.packet_timer[temp])
            if spent_time > max_waiting_time:
                max_waiting_time = spent_time
                oldest_packet = temp
            temp = (temp + 1) % MAX_SEQUENCE_NUMBER
        if max_waiting_time > timeOut:
          self.window_write_lock.acquire()
          if self.current_window[oldest_packet] != 0:

self.connection.send(str.encode(self.current_window[oldest_packet].toBinaryString(46)))
            print('Packet ', oldest_packet, ' Resent')
            self.packet_timer[oldest_packet] = time.time()
            self.totalPkt += 1
          self.window_write_lock.release()
```

ReceiverSW.py

This is for the stop-and-wait protocol. It contains a constructor to initialize the details, functions to send ACK, resend previous ACK and startReceiving() function which calls the above functions as required.

def sendAck(self)

Method Description: This sends an ACK to the channel.

Code Snippet:

```
  def sendAck(self):
    packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['ack'], self.seqNo, 'acknowledgement Packet')
    self.recentACK = packet
    self.connection.send(str.encode(packet.toBinaryString(22)))
```

def resendPreviousACK(self)

Method Description: This resends a previous ACK to the channel.

Code Snippet:

```
  def resendPreviousACK(self):
    self.connection.send(str.encode(self.recentACK.toBinaryString(22)))
```

def startReceiving(self)

Method Description: This function calls the above functions as required for successful execution of the receiver algorithm.

Code Snippet:

```
  def startReceiving(self):
    time.sleep(0.4)
    data = self.connection.recv(576).decode()
    total_data = ''
    while data != 'end':
      packet = PacketManager.Packet.build(data)
      print("\nPacket ",packet.getSeqNo()," Received")
      if not packet.hasError():
        print("No error")
        seqNo = packet.getSeqNo()
        if self.seqNo == seqNo:
            data = packet.getData()
            total_data += data
            self.seqNo = (self.seqNo + 1) % 2
            self.sendAck()
            print("ACK sent\n")
        else:
            self.resendPreviousACK()
            print("Previous ACK resent")
    else:
        print("Packet has error...so discarded")
    data = self.connection.recv(576).decode()
file = open(self.file_name, 'a')
file.write(total_data)
file.close()
```

ReceiverGBN.py

This is for the go-back-n protocol. It contains a constructor to initialize the details, functions to send ACK, resend previous ACK and startReceiving() function which calls the above functions as required.

def sendAck(self)

Method Description: This sends an ACK to the channel.

Code Snippet:

```
    def sendAck(self):
    packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['ack'], self.seqNo, 'acknowledgement Packet')
    self.recentACK = packet
    self.connection.send(str.encode(packet.toBinaryString(22)))
```

def resendPreviousACK(self)

Method Description: This resends a previous ACK to the channel.

Code Snippet:

```
def resendPreviousACK(self):
self.connection.send(str.encode(self.recentACK.toBinaryString(22)))
```

def startReceiving(self)

Method Description: This function calls the above functions as required for successful execution of the receiver algorithm.

Code Snippet:

```
def startReceiving(self):
time.sleep(0.4)
data = self.connection.recv(576).decode()
total_data = ''
while data != 'end':
    packet = PacketManager.Packet.build(data)
    print("\nPacket ",packet.getSeqNo()," Received")
    if not packet.hasError():
        print("No error")
        seqNo = packet.getSeqNo()
        if self.seqNo == seqNo:
            data = packet.getData()
            # print(data)
            total_data += data
            self.seqNo = ((self.seqNo + 1) % WINDOW_SIZE)
            self.sendAck()
            print("ACK Sent\n")
        else:
            self.resendPreviousACK()
            print("Previous ACK Resent")
    else:
        print("Packet has error...so discarded")
    data = self.connection.recv(576).decode()
file = open(self.file_name, 'a')
file.write(total_data)
file.close()
```

ReceiverSR.py

This is for the selective repeat protocol. It contains a constructor to initialize the details, functions to check validity of the sequence number of the packet, send ACK, send NAK, resend previous ACK and startReceiving() function which calls the above functions as required.

def validSEQ(self, seq_no)

Method Description: This function checks whether the packet received is within the current receiving window by comparing the sequence number.

Code Snippet:

```
    def validSEQ(self, seq_no: int):
        if (self.front <= seq_no < self.end) or (self.end < self.front <= seq_no) or (seq_no <
self.end < self.front):
            return True
        else:
            return False
```

def sendAck(self)

Method Description: This sends an ACK to the channel.

Code Snippet:

```
    def sendAck(self):
        packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['ack'], self.front, 'acknowledgement Packet')
        self.recentACK = packet
        print('Sent ACK no = ', self.front)
        self.connection.send(str.encode(packet.toBinaryString(22)))
        self.lastACKsent = time.time()
```

def sendNak(self)

Method Description: This sends a NAK to the channel.

Code Snippet:

```
    def sendNak(self):
        packet = PacketManager.Packet(self.senderAddress, self.receiverAddress,
self.packetType['nak'], self.front, 'No acknowledgement')
        self.connection.send(str.encode(packet.toBinaryString(22)))
        print('Sent NAK no = ', self.front)
```

def resendPreviousACK(self)

Method Description: This resends a previous ACK to the channel.

Code Snippet:

```
def resendPreviousACK(self):
while not self.endReceiving:
    if self.lastACKsent == 'not started':
        continue
    current_time = time.time()
    total_spent = (current_time - self.lastACKsent)
    if total_spent > 1:
        self.connection.send(str.encode(self.recentACK.toBinaryString(22)))
        self.lastACKsent = time.time()
```

def startReceiving(self)

Method Description: This function calls the above functions as required for successful execution of the receiver algorithm.

Code Snippet:

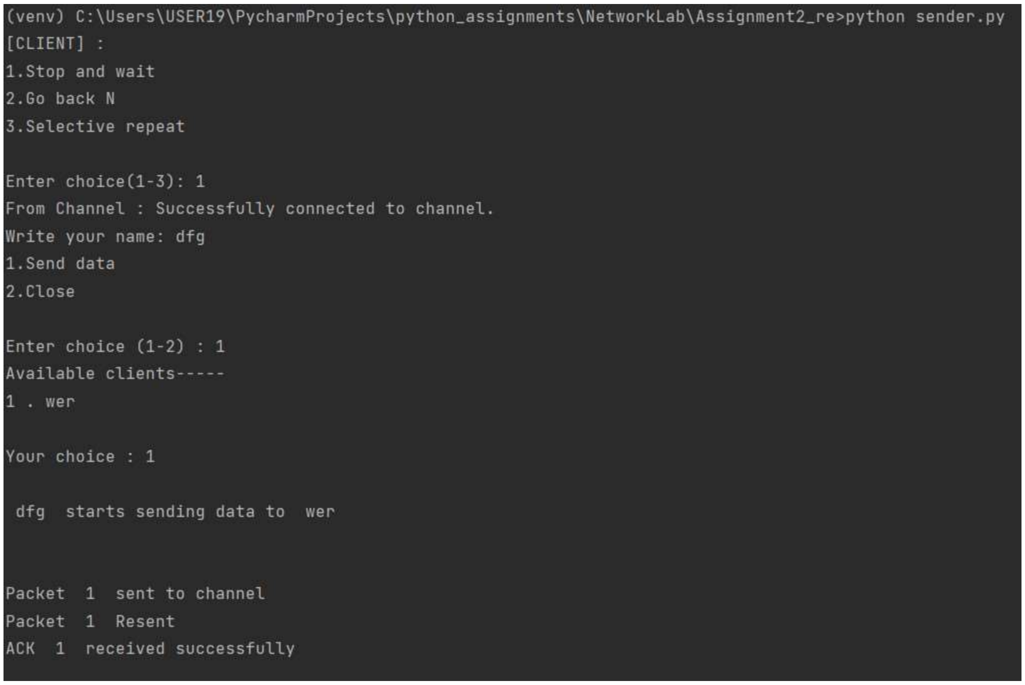
```
def startReceiving(self):
time.sleep(0.4)
ACKresendingThread = threading.Thread(target=self.resendPreviousACK)
ACKresendingThread.start()
data = self.connection.recv(576).decode()
total_data = ''
while data != 'end':
    packet = PacketManager.Packet.build(data)
    print("\nPacket ",packet.getSeqNo()," Received")
    if not packet.hasError():
        print("No error")
        seqNo = packet.getSeqNo()
        if seqNo != self.front and self.NAK_sent == False:
            self.sendNak()
            self.NAK_sent = True
```

```
                if self.validSEQ(seqNo) and self.filled_up[seqNo] == False:
                    self.filled_up[seqNo] = True
                    self.window[seqNo] = packet.getData()
                    # print(packet.getData())
                    while self.filled_up[self.front]:
                        total_data += self.window[self.front]
                        self.filled_up[self.front] = False
                        self.front = (self.front + 1) % MAX_SEQUENCE_NUMBER
                        self.end = (self.end + 1) % MAX_SEQUENCE_NUMBER
                        self.ACK_needed = True
                        print('Packet Received successfully')
                    if self.ACK_needed:
                        self.sendAck()
                        self.ACK_needed = False
                        self.NAK_sent = False
            else:
                print("Packet has error...so discarded")
            data = self.connection.recv(576).decode()
        self.endReceiving = True
        ACKresendingThread.join()
        file = open(self.file_name, 'a')
        file.write(total_data)
        file.close()
```

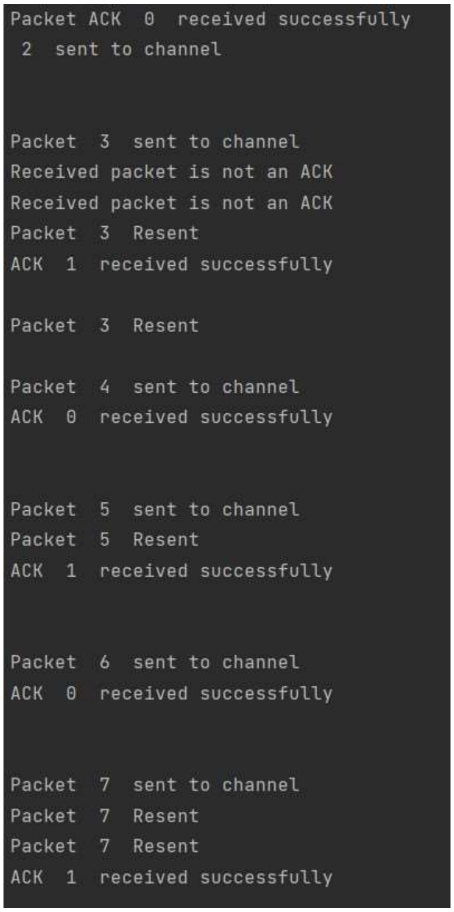
TEST CASES

Sample Test 1: To check the working of the stop-and-wait protocol (snippets are shown as the entire output was too large)

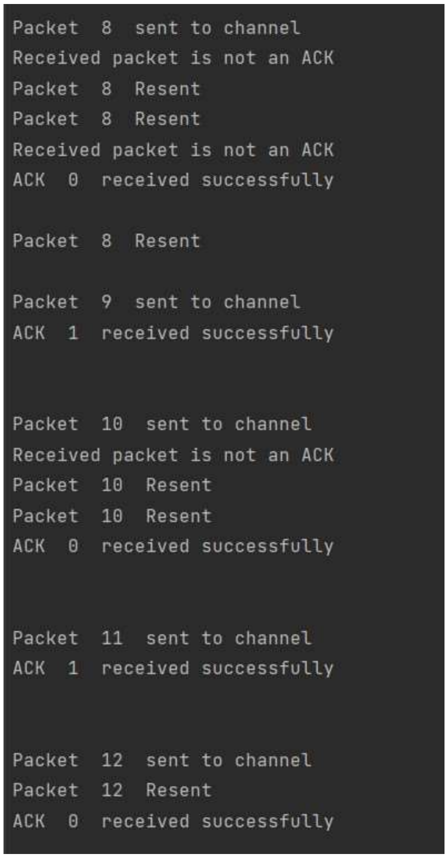
SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab\Assignment2_re>python sender.py
[CLIENT] :
1.Stop and wait
2.Go back N
3.Selective repeat

Enter choice(1-3): 1
From Channel : Successfully connected to channel.
Write your name: dfg
1.Send data
2.Close

Enter choice (1-2) : 1
Available clients-----
1 . wer

Your choice : 1

 dfg  starts sending data to  wer


Packet  1  sent to channel
Packet  1  Resent
ACK  1  received successfully
```

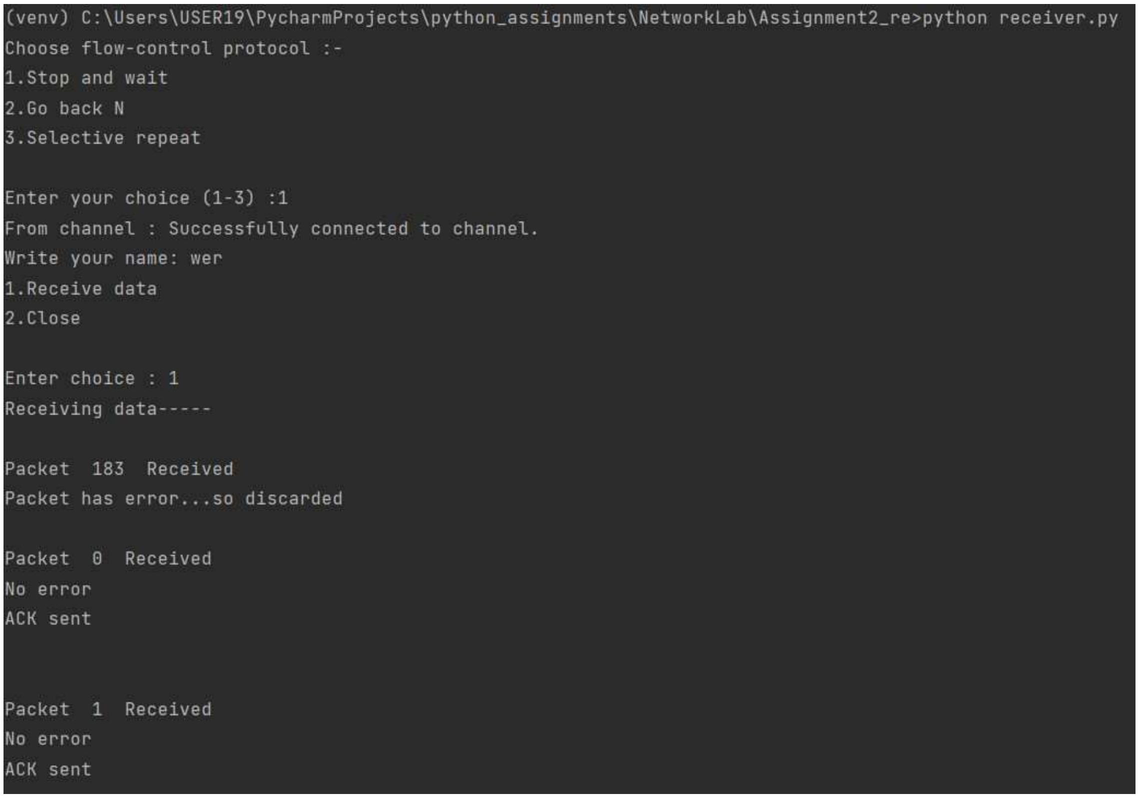


```
Packet ACK  0  received successfully
 2  sent to channel


Packet  3  sent to channel
Received packet is not an ACK
Received packet is not an ACK
Packet  3  Resent
ACK  1  received successfully

Packet  3  Resent

Packet  4  sent to channel
ACK  0  received successfully


Packet  5  sent to channel
Packet  5  Resent
ACK  1  received successfully


Packet  6  sent to channel
ACK  0  received successfully


Packet  7  sent to channel
Packet  7  Resent
Packet  7  Resent
ACK  1  received successfully
```

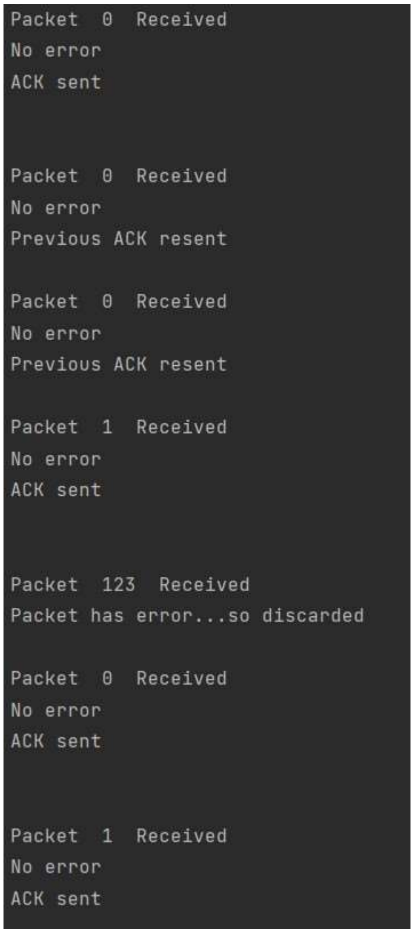


```
Packet  8  sent to channel
Received packet is not an ACK
Packet  8  Resent
Packet  8  Resent
Received packet is not an ACK
ACK  0  received successfully

Packet  8  Resent

Packet  9  sent to channel
ACK  1  received successfully


Packet  10  sent to channel
Received packet is not an ACK
Packet  10  Resent
Packet  10  Resent
ACK  0  received successfully


Packet  11  sent to channel
ACK  1  received successfully


Packet  12  sent to channel
Packet  12  Resent
ACK  0  received successfully
```

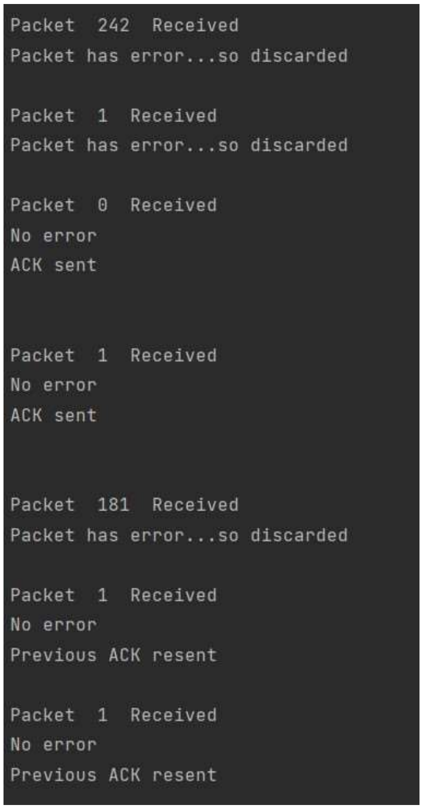
RECEIVER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab\Assignment2_re>python receiver.py
Choose flow-control protocol :-
1.Stop and wait
2.Go back N
3.Selective repeat

Enter your choice (1-3) :1
From channel : Successfully connected to channel.
Write your name: wer
1.Receive data
2.Close

Enter choice : 1
Receiving data-----

Packet  183  Received
Packet has error...so discarded

Packet  0  Received
No error
ACK sent


Packet  1  Received
No error
ACK sent
```



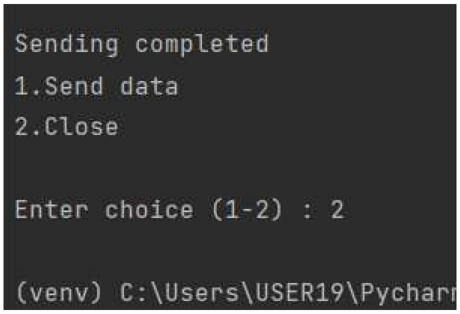
```
Packet  0  Received
No error
ACK sent


Packet  0  Received
No error
Previous ACK resent

Packet  0  Received
No error
Previous ACK resent

Packet  1  Received
No error
ACK sent


Packet  123  Received
Packet has error...so discarded

Packet  0  Received
No error
ACK sent


Packet  1  Received
No error
ACK sent
```



```
Packet  242  Received
Packet has error...so discarded

Packet  1  Received
Packet has error...so discarded

Packet  0  Received
No error
ACK sent


Packet  1  Received
No error
ACK sent


Packet  181  Received
Packet has error...so discarded

Packet  1  Received
No error
Previous ACK resent

Packet  1  Received
No error
Previous ACK resent
```

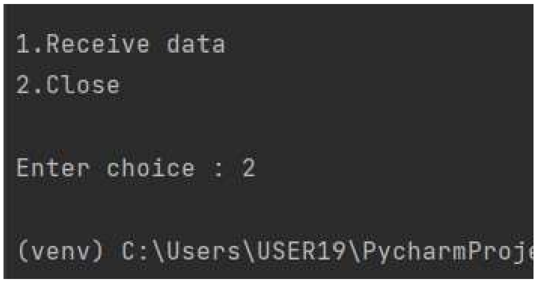
Sample Test 2: To check whether the connections are closed successfully

SENDER TERMINAL



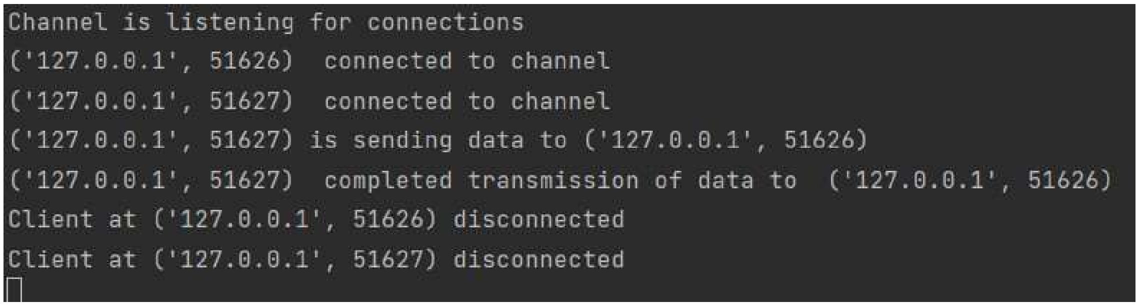
```
Sending completed
1.Send data
2.Close

Enter choice (1-2) : 2

(venv) C:\Users\USER19\Pychar
```

RECEIVER TERMINAL



```
1.Receive data
2.Close

Enter choice : 2

(venv) C:\Users\USER19\PycharmProj
```

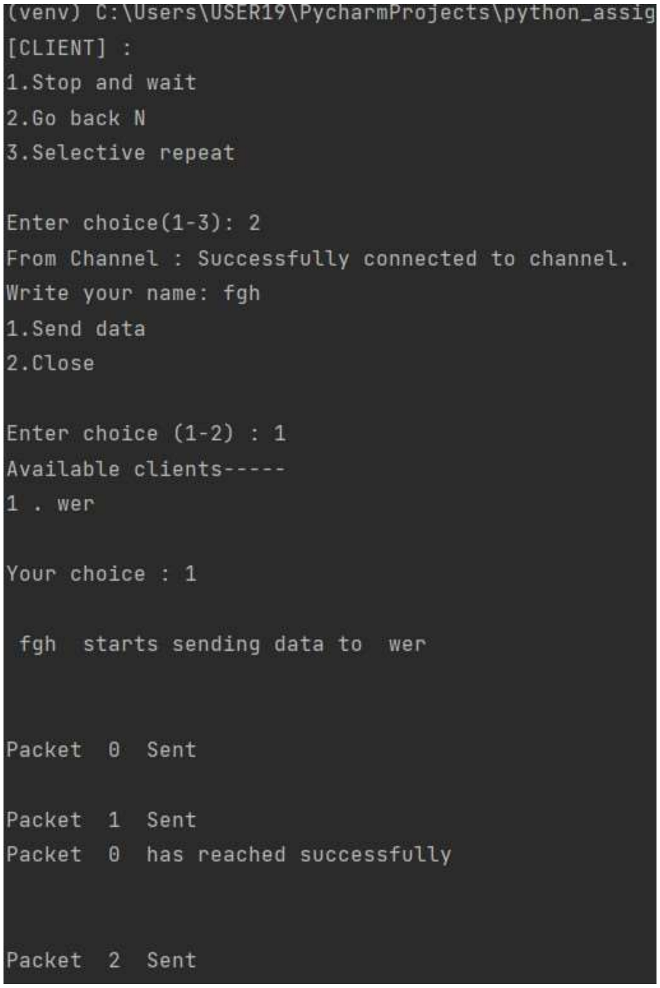
CHANNEL TERMINAL



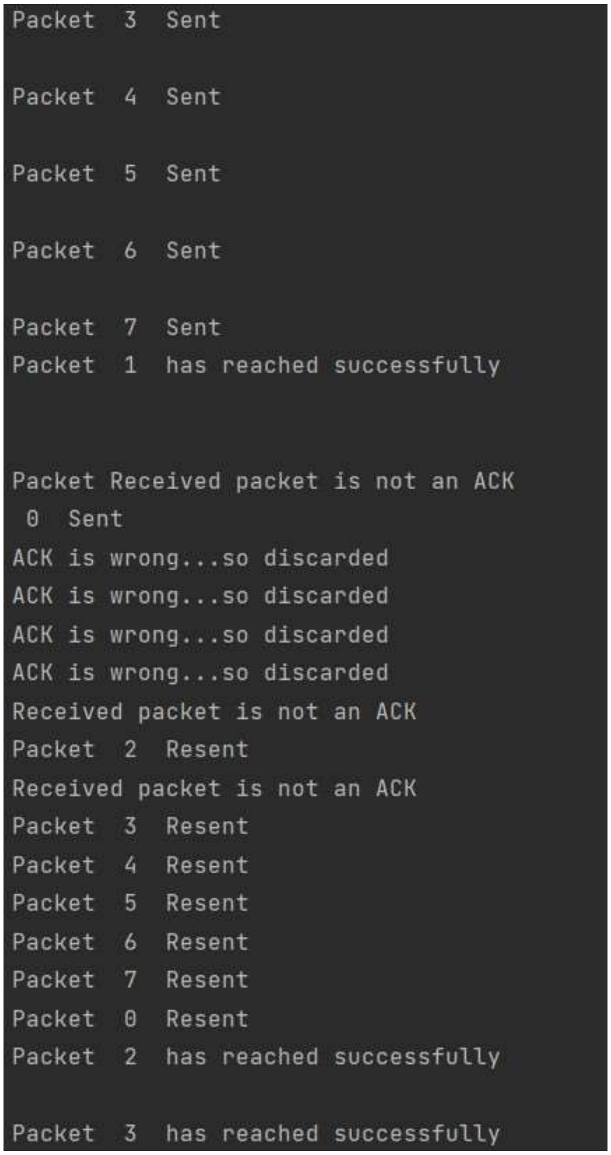
```
Channel is listening for connections
('127.0.0.1', 51626)  connected to channel
('127.0.0.1', 51627)  connected to channel
('127.0.0.1', 51627) is sending data to ('127.0.0.1', 51626)
('127.0.0.1', 51627)  completed transmission of data to  ('127.0.0.1', 51626)
Client at ('127.0.0.1', 51626) disconnected
Client at ('127.0.0.1', 51627) disconnected
```

Sample Test 3: To check the working of the go-back-n protocol (snippets are shown as the entire output was too large)

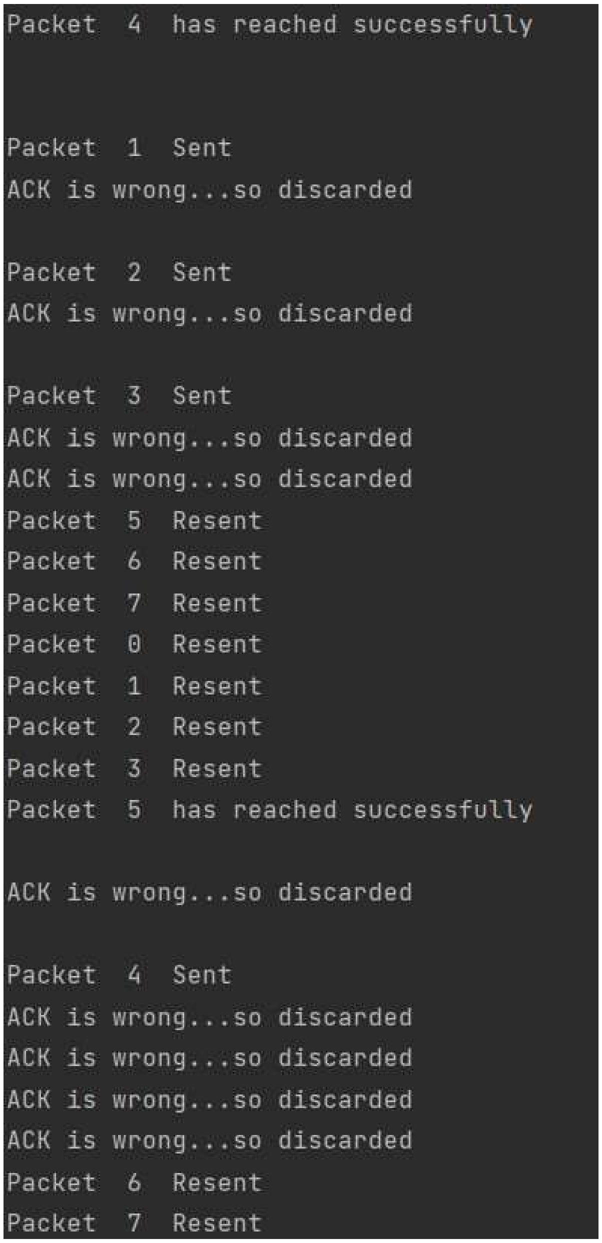
SENDER TERMINAL



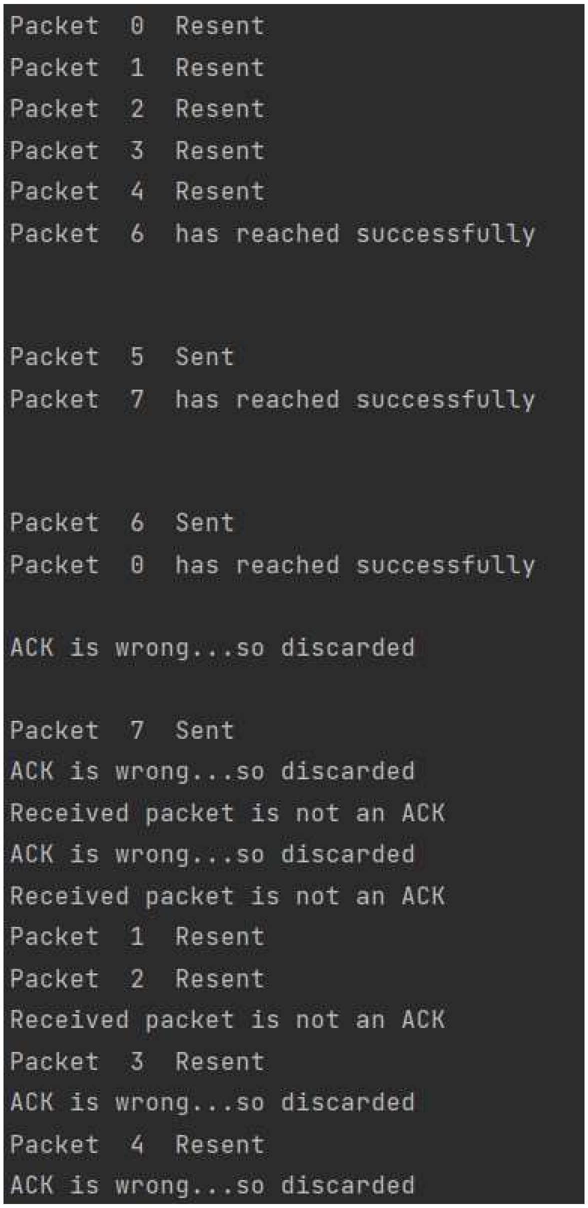
```
(venv) C:\Users\USER19\PycharmProjects\python_assig
[CLIENT] :
1.Stop and wait
2.Go back N
3.Selective repeat

Enter choice(1-3): 2
From Channel : Successfully connected to channel.
Write your name: fgh
1.Send data
2.Close

Enter choice (1-2) : 1
Available clients-----
1 . wer

Your choice : 1

 fgh  starts sending data to  wer


Packet  0  Sent

Packet  1  Sent
Packet  0  has reached successfully


Packet  2  Sent
```



```
Packet  3  Sent

Packet  4  Sent

Packet  5  Sent

Packet  6  Sent

Packet  7  Sent
Packet  1  has reached successfully


Packet Received packet is not an ACK
 0  Sent
ACK is wrong...so discarded
ACK is wrong...so discarded
ACK is wrong...so discarded
ACK is wrong...so discarded
Received packet is not an ACK
Packet  2  Resent
Received packet is not an ACK
Packet  3  Resent
Packet  4  Resent
Packet  5  Resent
Packet  6  Resent
Packet  7  Resent
Packet  0  Resent
Packet  2  has reached successfully

Packet  3  has reached successfully
```

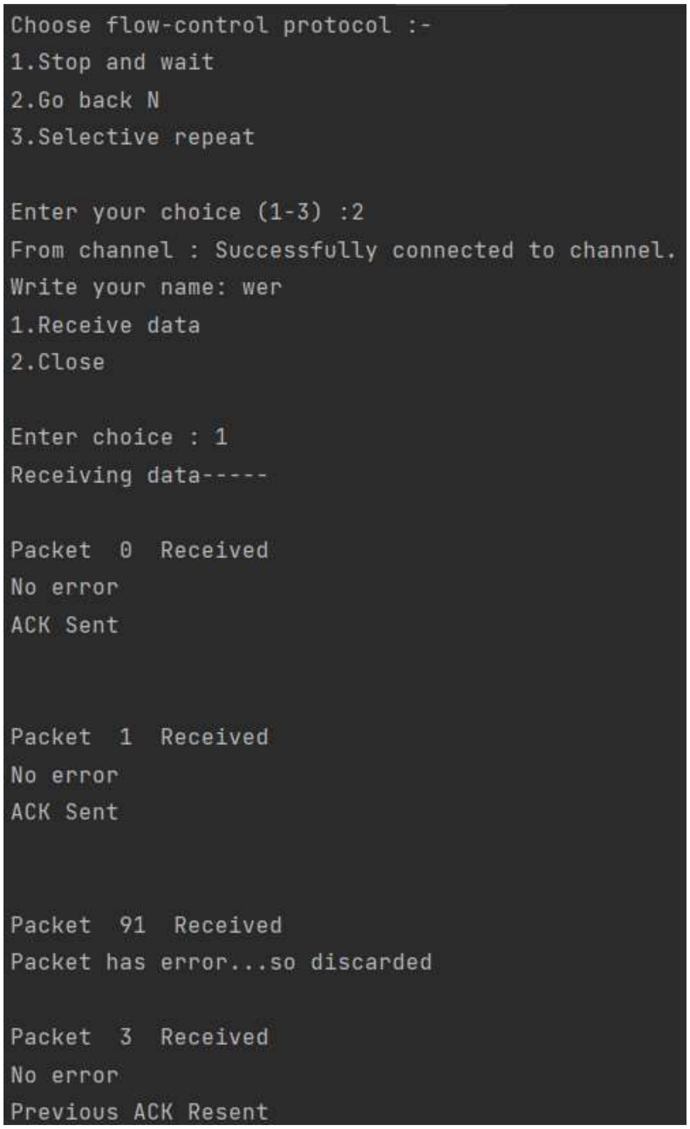


```
Packet  4  has reached successfully


Packet  1  Sent
ACK is wrong...so discarded

Packet  2  Sent
ACK is wrong...so discarded

Packet  3  Sent
ACK is wrong...so discarded
ACK is wrong...so discarded
Packet  5  Resent
Packet  6  Resent
Packet  7  Resent
Packet  0  Resent
Packet  1  Resent
Packet  2  Resent
Packet  3  Resent
Packet  5  has reached successfully

ACK is wrong...so discarded

Packet  4  Sent
ACK is wrong...so discarded
ACK is wrong...so discarded
ACK is wrong...so discarded
ACK is wrong...so discarded
Packet  6  Resent
Packet  7  Resent
```

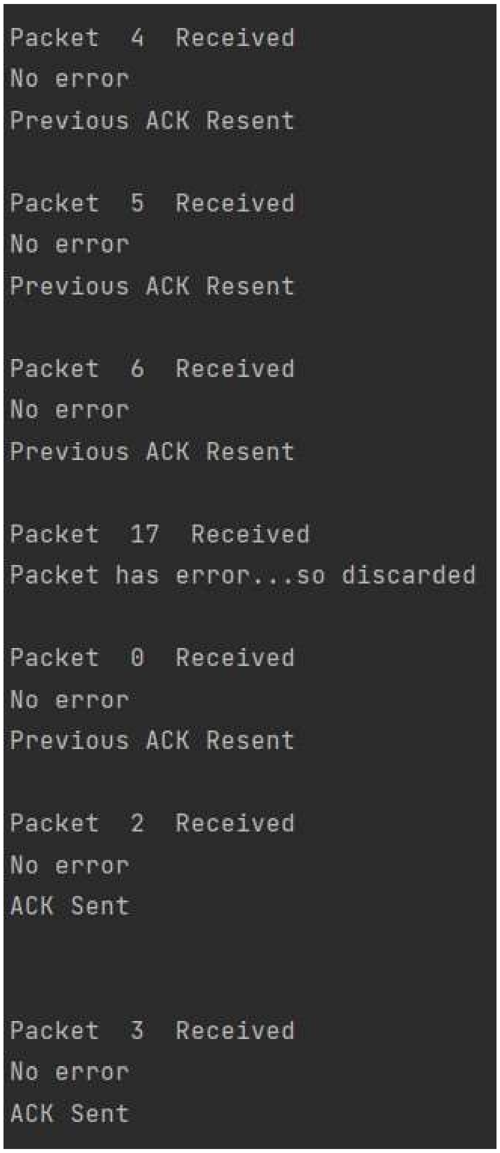


```
Packet  0  Resent
Packet  1  Resent
Packet  2  Resent
Packet  3  Resent
Packet  4  Resent
Packet  6  has reached successfully


Packet  5  Sent
Packet  7  has reached successfully


Packet  6  Sent
Packet  0  has reached successfully

ACK is wrong...so discarded

Packet  7  Sent
ACK is wrong...so discarded
Received packet is not an ACK
ACK is wrong...so discarded
Received packet is not an ACK
Packet  1  Resent
Packet  2  Resent
Received packet is not an ACK
Packet  3  Resent
ACK is wrong...so discarded
Packet  4  Resent
ACK is wrong...so discarded
```

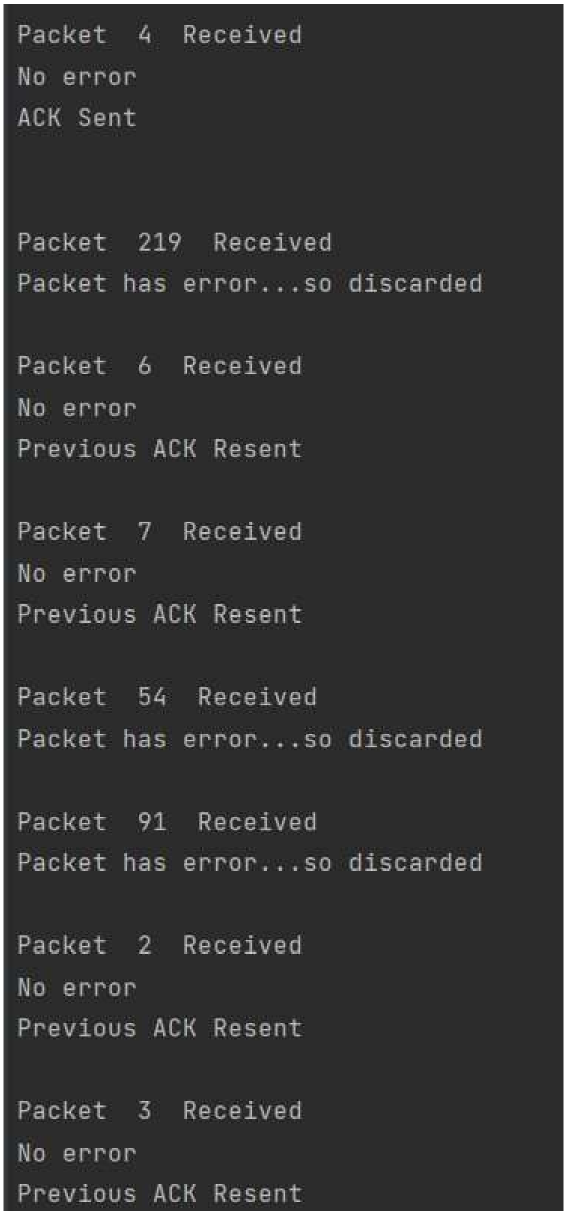
RECEIVER TERMINAL



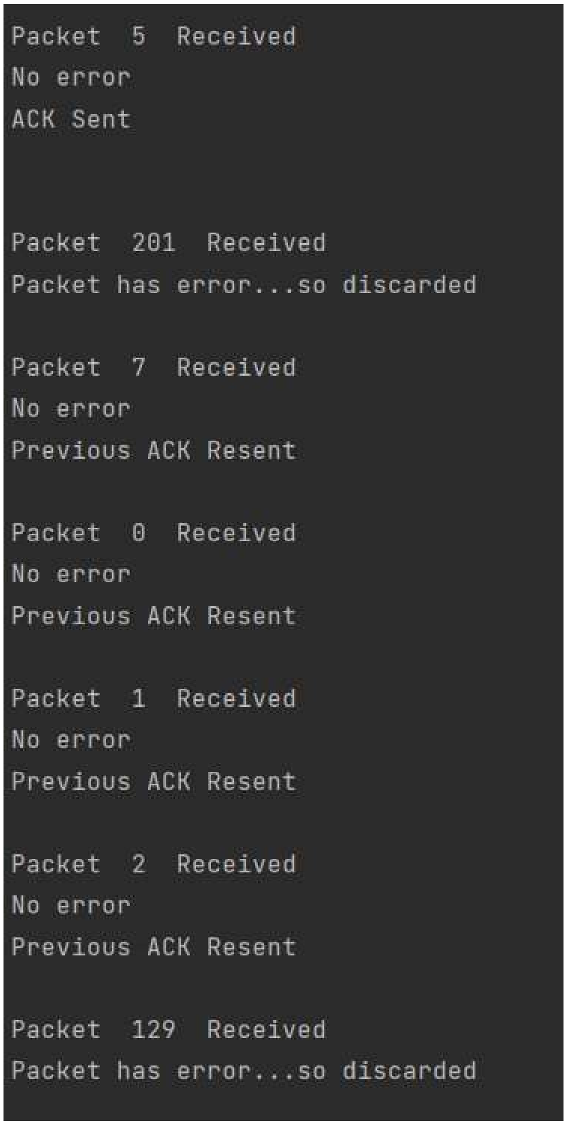
```
Choose flow-control protocol :-
1.Stop and wait
2.Go back N
3.Selective repeat

Enter your choice (1-3) :2
From channel : Successfully connected to channel.
Write your name: wer
1.Receive data
2.Close

Enter choice : 1
Receiving data-----

Packet  0  Received
No error
ACK Sent


Packet  1  Received
No error
ACK Sent


Packet  91  Received
Packet has error...so discarded

Packet  3  Received
No error
Previous ACK Resent
```



```
Packet  4  Received
No error
Previous ACK Resent

Packet  5  Received
No error
Previous ACK Resent

Packet  6  Received
No error
Previous ACK Resent

Packet  17  Received
Packet has error...so discarded

Packet  0  Received
No error
Previous ACK Resent

Packet  2  Received
No error
ACK Sent


Packet  3  Received
No error
ACK Sent
```



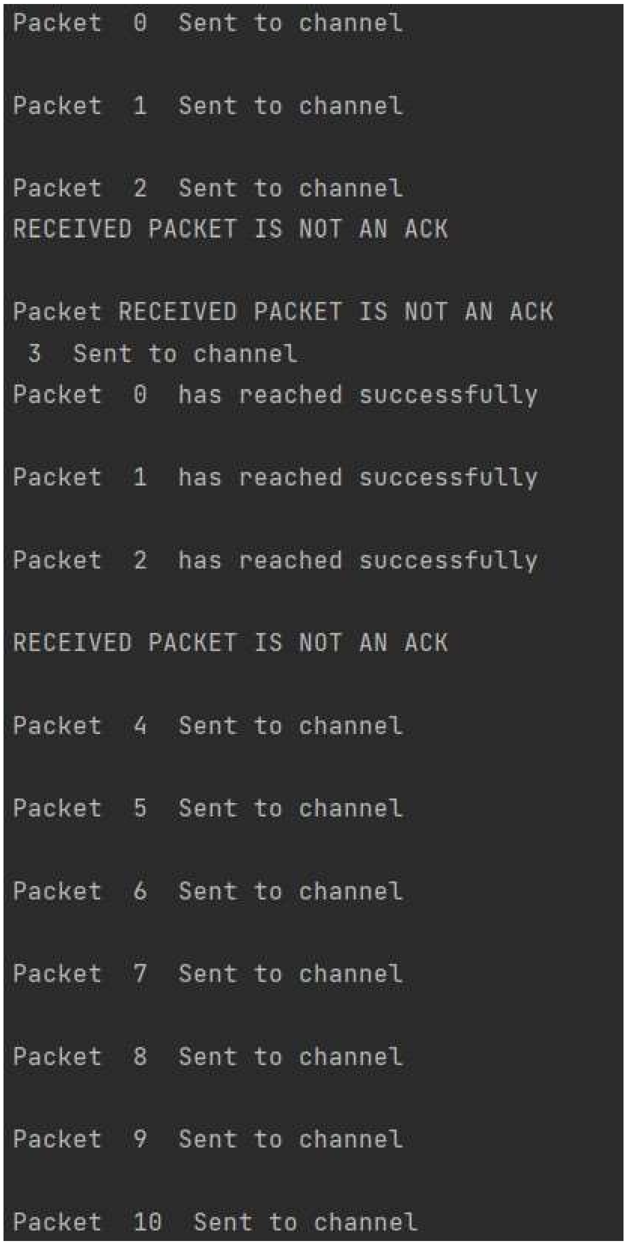
```
Packet  4  Received
No error
ACK Sent


Packet  219  Received
Packet has error...so discarded

Packet  6  Received
No error
Previous ACK Resent

Packet  7  Received
No error
Previous ACK Resent

Packet  54  Received
Packet has error...so discarded

Packet  91  Received
Packet has error...so discarded

Packet  2  Received
No error
Previous ACK Resent

Packet  3  Received
No error
Previous ACK Resent
```



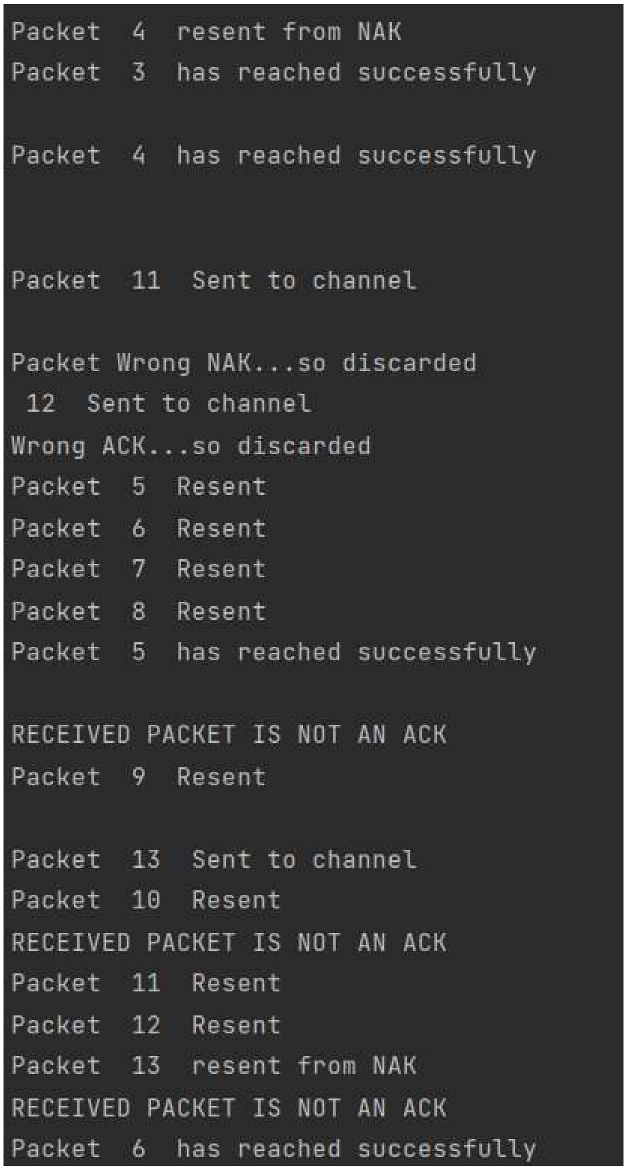
```
Packet  5  Received
No error
ACK Sent


Packet  201  Received
Packet has error...so discarded

Packet  7  Received
No error
Previous ACK Resent

Packet  0  Received
No error
Previous ACK Resent

Packet  1  Received
No error
Previous ACK Resent

Packet  2  Received
No error
Previous ACK Resent

Packet  129  Received
Packet has error...so discarded
```

Sample Test 4: To check the working of the selective-repeat protocol (snippets are shown as the entire output was too large)

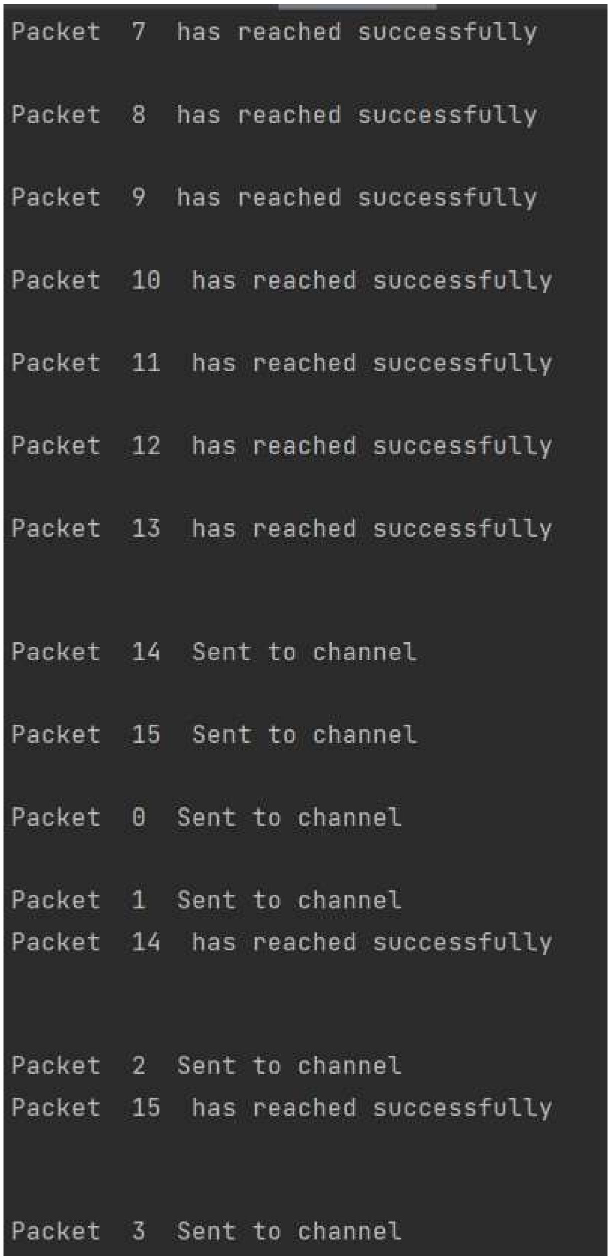
SENDER TERMINAL



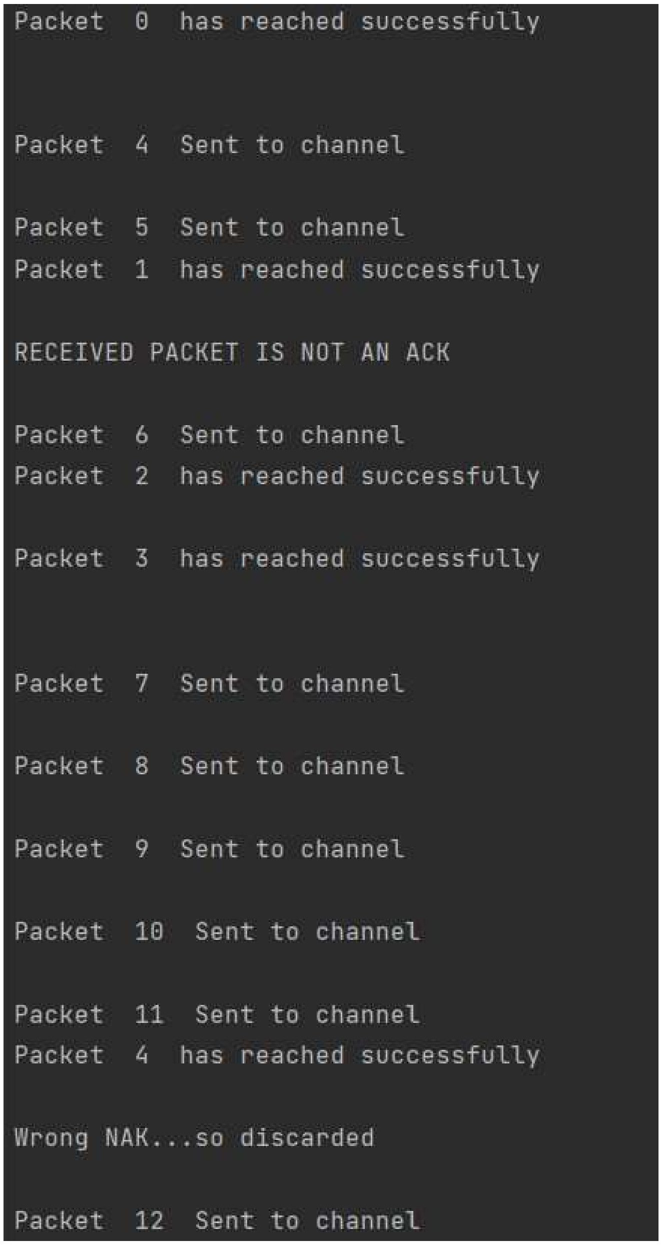
```
Packet  0  Sent to channel

Packet  1  Sent to channel

Packet  2  Sent to channel
RECEIVED PACKET IS NOT AN ACK

Packet RECEIVED PACKET IS NOT AN ACK
 3  Sent to channel
Packet  0  has reached successfully

Packet  1  has reached successfully

Packet  2  has reached successfully

RECEIVED PACKET IS NOT AN ACK

Packet  4  Sent to channel

Packet  5  Sent to channel

Packet  6  Sent to channel

Packet  7  Sent to channel

Packet  8  Sent to channel

Packet  9  Sent to channel

Packet  10  Sent to channel
```



```
Packet  4  resent from NAK
Packet  3  has reached successfully

Packet  4  has reached successfully


Packet  11  Sent to channel

Packet Wrong NAK...so discarded
 12  Sent to channel
Wrong ACK...so discarded
Packet  5  Resent
Packet  6  Resent
Packet  7  Resent
Packet  8  Resent
Packet  5  has reached successfully

RECEIVED PACKET IS NOT AN ACK
Packet  9  Resent

Packet  13  Sent to channel
Packet  10  Resent
RECEIVED PACKET IS NOT AN ACK
Packet  11  Resent
Packet  12  Resent
Packet  13  resent from NAK
RECEIVED PACKET IS NOT AN ACK
Packet  6  has reached successfully
```

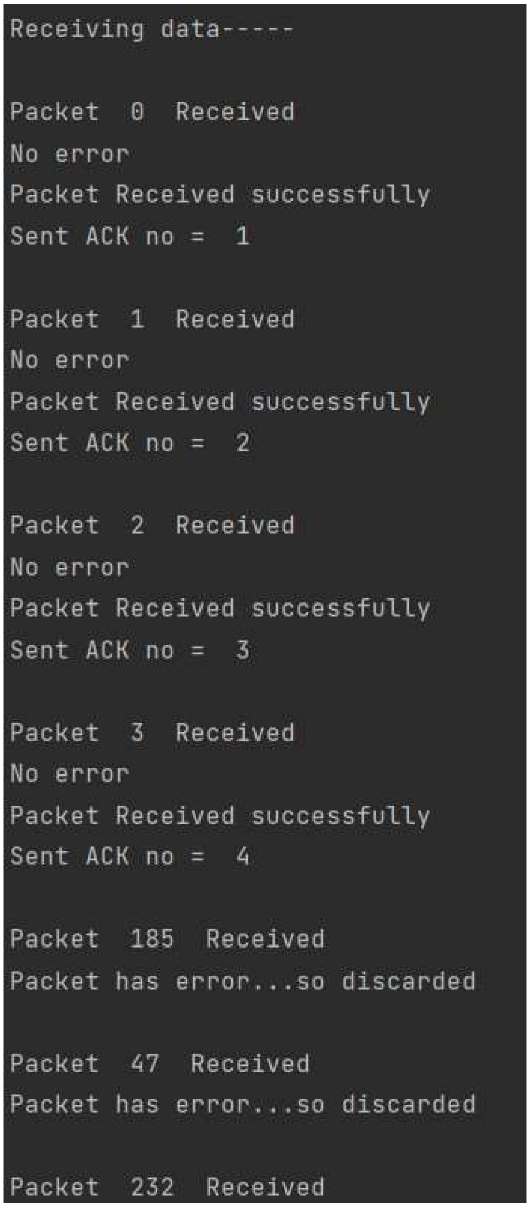


```
Packet  7  has reached successfully

Packet  8  has reached successfully

Packet  9  has reached successfully

Packet  10  has reached successfully

Packet  11  has reached successfully

Packet  12  has reached successfully

Packet  13  has reached successfully


Packet  14  Sent to channel

Packet  15  Sent to channel

Packet  0  Sent to channel

Packet  1  Sent to channel
Packet  14  has reached successfully


Packet  2  Sent to channel
Packet  15  has reached successfully


Packet  3  Sent to channel
```

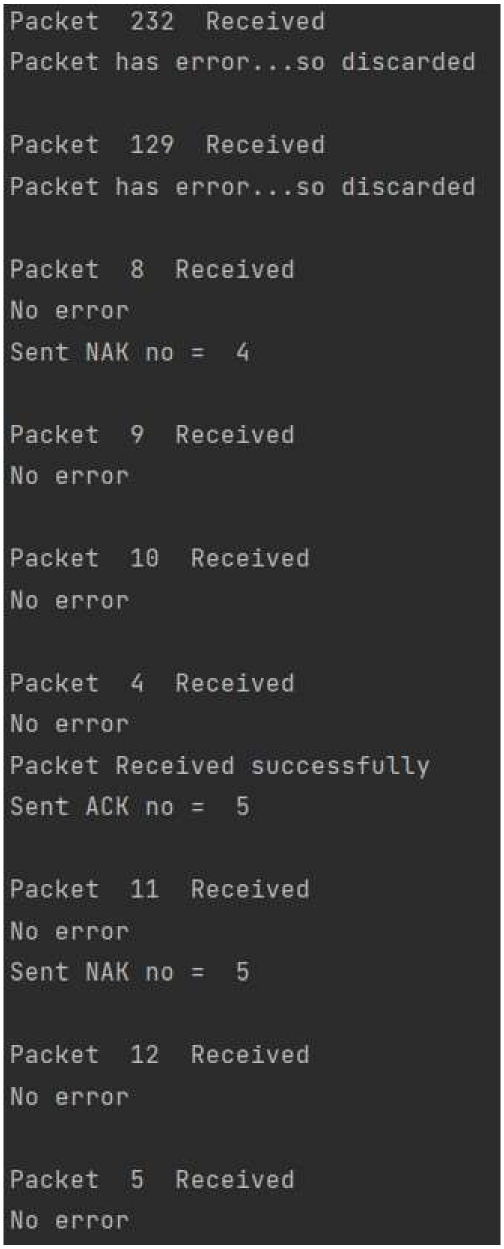


```
Packet  0  has reached successfully


Packet  4  Sent to channel

Packet  5  Sent to channel
Packet  1  has reached successfully

RECEIVED PACKET IS NOT AN ACK

Packet  6  Sent to channel
Packet  2  has reached successfully

Packet  3  has reached successfully


Packet  7  Sent to channel

Packet  8  Sent to channel

Packet  9  Sent to channel

Packet  10  Sent to channel

Packet  11  Sent to channel
Packet  4  has reached successfully

Wrong NAK...so discarded

Packet  12  Sent to channel
```

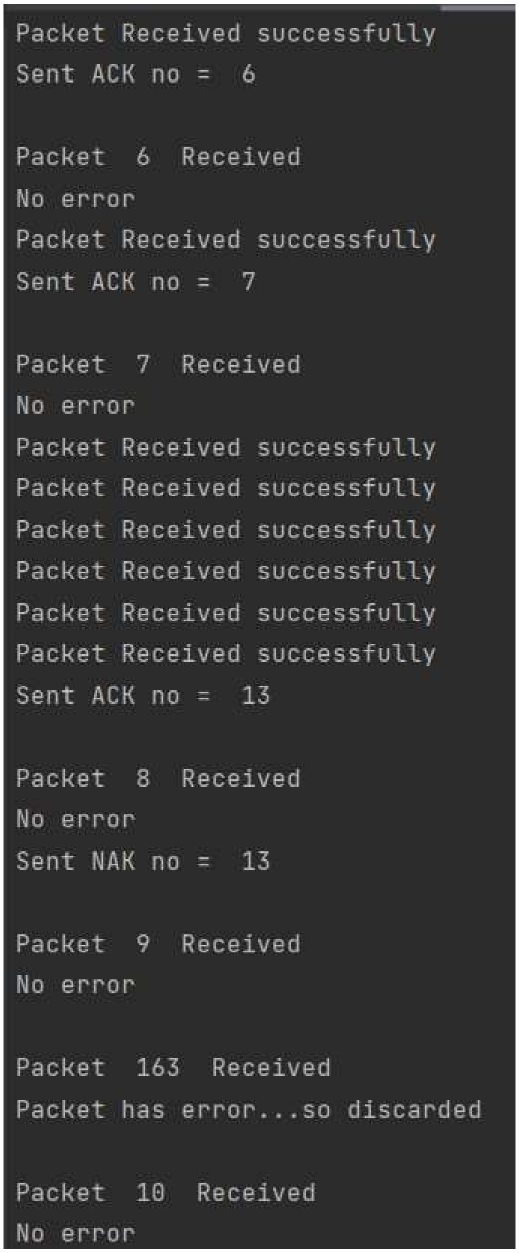
RECEIVER TERMINAL



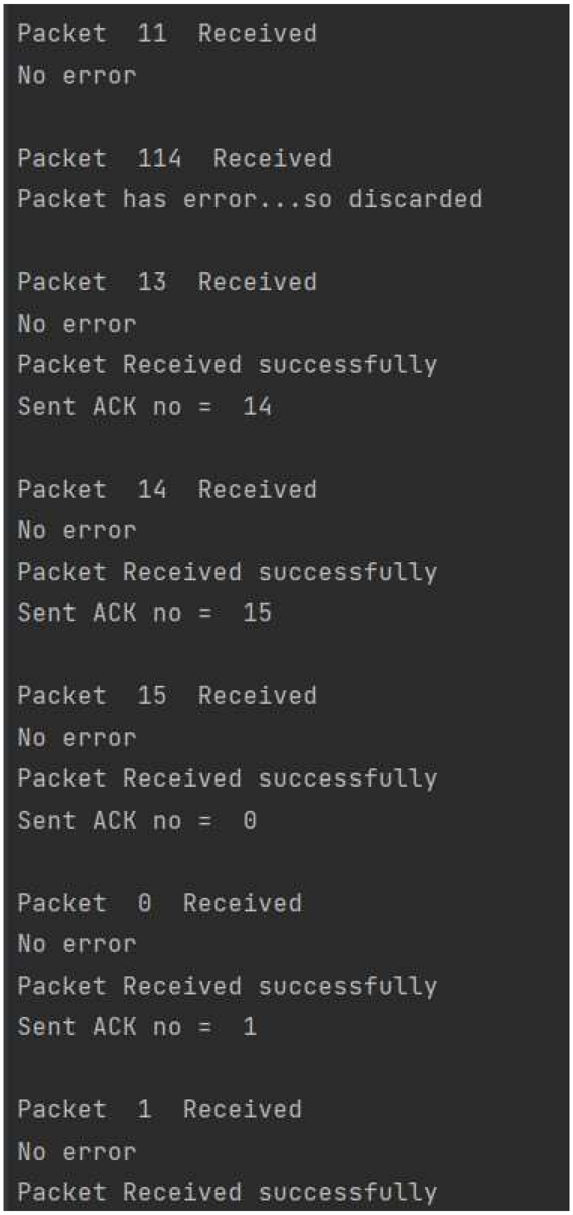
```
Receiving data-----

Packet  0  Received
No error
Packet Received successfully
Sent ACK no =  1

Packet  1  Received
No error
Packet Received successfully
Sent ACK no =  2

Packet  2  Received
No error
Packet Received successfully
Sent ACK no =  3

Packet  3  Received
No error
Packet Received successfully
Sent ACK no =  4

Packet  185  Received
Packet has error...so discarded

Packet  47  Received
Packet has error...so discarded

Packet  232  Received
```



```
Packet  232  Received
Packet has error...so discarded

Packet  129  Received
Packet has error...so discarded

Packet  8  Received
No error
Sent NAK no =  4

Packet  9  Received
No error

Packet  10  Received
No error

Packet  4  Received
No error
Packet Received successfully
Sent ACK no =  5

Packet  11  Received
No error
Sent NAK no =  5

Packet  12  Received
No error

Packet  5  Received
No error
```



```
Packet Received successfully
Sent ACK no =  6

Packet  6  Received
No error
Packet Received successfully
Sent ACK no =  7

Packet  7  Received
No error
Packet Received successfully
Packet Received successfully
Packet Received successfully
Packet Received successfully
Packet Received successfully
Packet Received successfully
Sent ACK no =  13

Packet  8  Received
No error
Sent NAK no =  13

Packet  9  Received
No error

Packet  163  Received
Packet has error...so discarded

Packet  10  Received
No error
```



```
Packet  11  Received
No error

Packet  114  Received
Packet has error...so discarded

Packet  13  Received
No error
Packet Received successfully
Sent ACK no =  14

Packet  14  Received
No error
Packet Received successfully
Sent ACK no =  15

Packet  15  Received
No error
Packet Received successfully
Sent ACK no =  0

Packet  0  Received
No error
Packet Received successfully
Sent ACK no =  1

Packet  1  Received
No error
Packet Received successfully
```

RESULTS

We have to evaluate the performance of the data link layer protocols in comparison to one another. For this, we take a dataset containing random characters and then convert them into binary using their ASCII value. We then break the data set up into packets and restrict the number of packets to be sent at a time from the sender to the receiver so that we can evaluate the three protocols based on the number of packets sent.

Performance metrics for evaluation: We consider the bandwidth of the channel to be 4000 bps.

We calculate the receiver throughput as (number of packets sent successfully/delay), delay as the total time taken for sending the entire dataset from the sender to the receiver, channel utilization as throughput/bandwidth and efficiency (or performance) as the (number of packets sent successfully/total number of packets sent, including retransmitted packets) * 100 %. The flow control mechanisms ensure that all the packets to be sent are sent successfully, the difference is the time or the number of retransmitted packets it requires in the process. The throughput is correct to the nearest integer.

Observation Tables:

Each observation percentage is correct to two decimal places.

NUMBER OF PACKETS TO BE SENT: 10

PROTOCOL	DELAY	THROUGHPUT	UTILIZATION	PERFORMANCE
STOP-AND-WAIT	5.94	425	10.63	66.67
GO BACK N	4.80	1161	29.04	45.45
SELECTIVE REPEAT	13.5	964	24.11	47.62

NUMBER OF PACKETS TO BE SENT: 20

PROTOCOL	DELAY	THROUGHPUT	UTILIZATION	PERFORMANCE
STOP-AND-WAIT	48.16	239	5.98	48.78
GO BACK N	8.03	1433	35.84	51.28
SELECTIVE REPEAT	8.88	1296	32.42	47.61

NUMBER OF PACKETS TO BE SENT: 30

PROTOCOL	DELAY	THROUGHPUT	UTILIZATION	PERFORMANCE
STOP-AND-WAIT	41.99	411	10.29	63.82
GO BACK N	24.69	699	17.49	29.12
SELECTIVE REPEAT	9.43	1830	45.77	56.60

NUMBER OF PACKETS TO BE SENT: 50

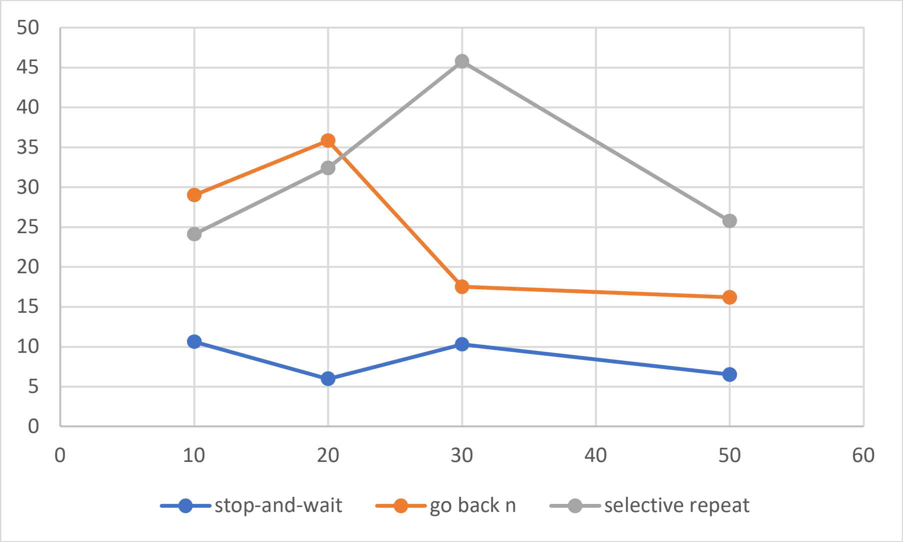
PROTOCOL	DELAY	THROUGHPUT	UTILIZATION	PERFORMANCE
STOP-AND-WAIT	110.54	260	6.51	51.02
GO BACK N	44.45	647	16.20	26.45
SELECTIVE REPEAT	27.92	1031	25.78	48.37

AVERAGE VALUES FOR THE PARAMETERS

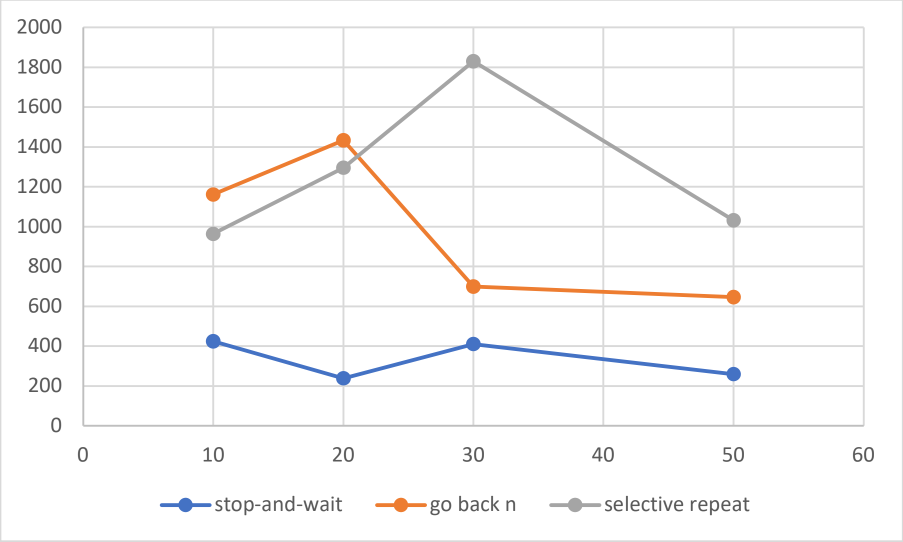
PROTOCOL	DELAY	THROUGHPUT	UTILIZATION
STOP-AND-WAIT	51.66	333.75	8.35
GO BACK N	20.49	985	24.64
SELECTIVE REPEAT	14.93	1280.25	32.02

Graphs:

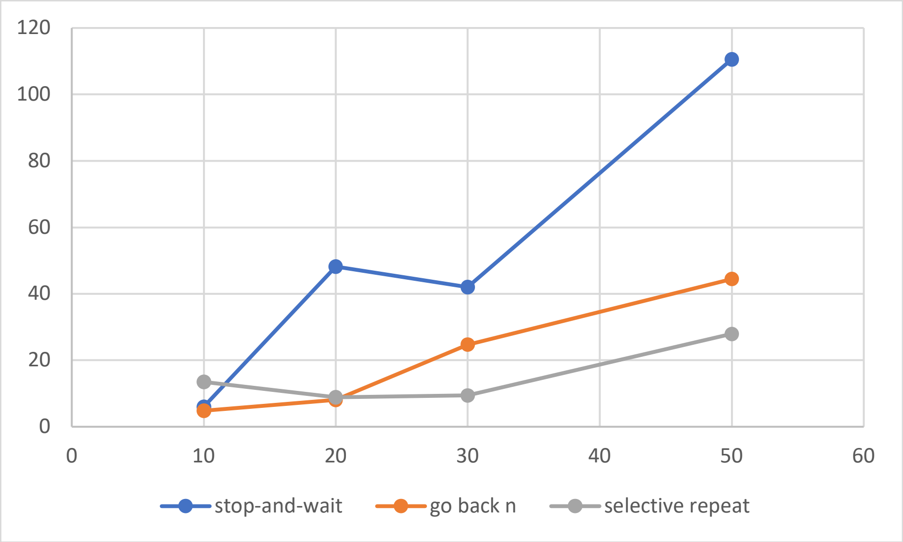
COMPARING THE CHANNEL UTILIZATION FOR THE DIFFERENT PROTOCOLS



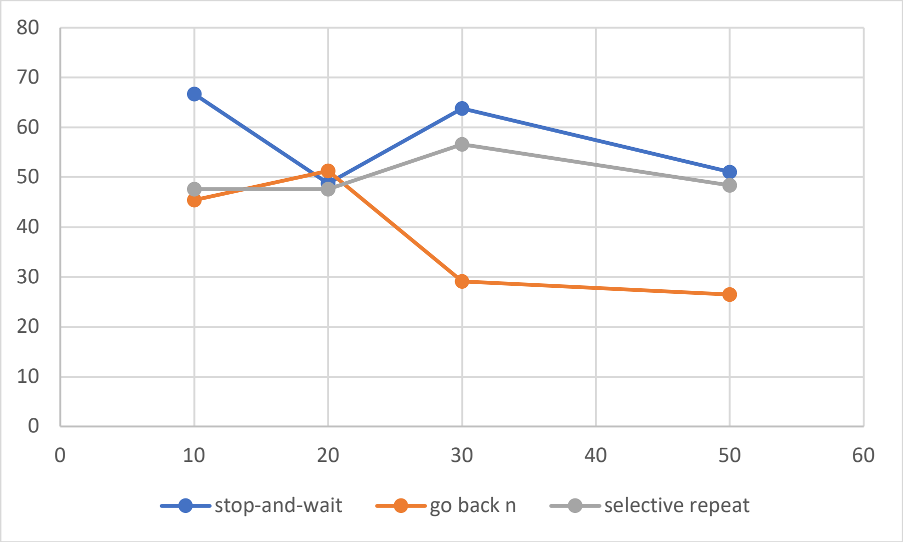
COMPARING THE THROUGHPUT FOR THE DIFFERENT PROTOCOLS



COMPARING THE DELAY (in seconds) FOR THE DIFFERENT PROTOCOLS



COMPARING THE PERFORMANCE (EFFICIENCY, IN %) OF THE PROTOCOLS



ANALYSIS

From the observations, we conclude that selective repeat protocol is the most efficient protocol as it has the least average delay, the most average throughput and good channel utilization. The stop-and-wait protocol has the least channel utilization and the most delay for large number of packets. If the number of packets is less, then using go back N or selective repeat protocols may become too complicated to use or have more delay. However, delay for stop and wait is significant for large number of packets so stop and wait protocol cannot be used for a large number of packets when the delay is to be minimized. The delay increases as the number of packets increases for each protocol; however, this increase is less for selective repeat protocol. The throughput is least for stop and wait ARQ. Go back N protocol has a greater throughput for small number of packets but selective repeat protocol has a greater throughput for large number of packets. The graphs for the channel utilization and for the throughput have the same shape as throughput = channel utilization*bandwidth and bandwidth is a constant for the channel (in this case 4000 bps). The performance for the stop and wait ARQ is better, at least for small number of packets, when the number of packets become larger, the efficiency of stop and wait ARQ and Selective Repeat ARQ become comparable. The efficiency of go-back-n is less, probably because if a packet arrives at the receiver out of order, then it is discarded whereas it is stored in selective repeat ARQ as long as it is in the receiving window and not already received. So, in go back n, the sending window with outstanding packets has to be retransmitted if a packet is lost or tainted in the transmission. However, stop and wait does not have reliable performance as the range of observations for the efficiency is large. Selective Repeat, on the other hand, has reliable performance as the range of observations is less (approx. 9% as opposed to 18% for stop and wait). We can always expect an efficiency around 50% for selective repeat.

Possible Improvements: We can include a mechanism for choosing the sender which has access to the channel at a given time. This becomes particularly important if more than one sender is ready to send at the same time.

COMMENTS

The lab assignment was interesting as it allowed us to see the different advantages and limitations of each data link layer protocol in a practical manner and allowed us to verify the results we have read about in theory. I found the analysis of the different observations very engaging. Writing the programs was very difficult, as there were synchronization issues with threads at first. Then I used different sockets and used locks. Comparing the performance of different values of window size for Go Back-N and Selective Repeat protocols is a possibility.