Name – Jeetesh Abrol
Roll – 002210501021
Sub - Computer Networks

Assignment no. 2
BCSE III
Group –A1

Assignment 2: Implement flow control mechanisms for Data Link Layer

- Framing(): This method will prepare the frame following the structure given below. In the header section, the MAC address of the source and destination are specified. Payload is the data of fixed size (pre decided value within the range 46-1500 bytes e.g., 46 bytes) from the input text file. Frame check Sequence using CRC/Checksum (using the CRC/Checksum module of assignment 1) is appended as a trailer.

● Channel(): The channel method introduces random delay (this will cause packet loss or timeout) and/or bit error (using the error injection module of assignment 1) while transferring frames.

Header <Source Address (6 bytes), Destination Address (6 bytes), Length (2 bytes), Frame seq. no. (1 byte)> 12 bytes	Data <Payload> 46-1500 bytes	Trailer FCS (Frame Check Sequence) <CRC/Checksum> 4 bytes

Fig 1: Data Frame Structure

● Send(): Sender program will send/transmit data frame using socket connection to Receiver program. Sender should decide whether to send a new data frame or retransmit the same frame again due to timeout.

● Timer(): Timer will be associated with each frame transmission. It will be used to check the timeout condition.

● Timeout(): This function should be called to compute the most recent data frame's round-trip time and then re-compute the value of timeout.

● Recv(): This method is invoked by the sender program whenever an ACK packet is received. Need to consider network time when the ACK was received to check the timeout condition.

Receiver Program: The Receiver program consist of following methods:

● Recv(): This method is invoked by the Receiver program whenever a Data frame is received. ● Check(): This method checks (using CRC/Checksum of assignment 1) if there is any error in data. The data frame is discarded if an error is detected otherwise accepted.

● Send(): Receiver program will prepare an acknowledgement frame and send it to the sender as a response to successful receipt of the data frame.

Here is the implementation In python,of the flow control mechanism Stop and wait , Go-back-N ARQ , Selective repeat ARQ.

STOP AND WAIT:

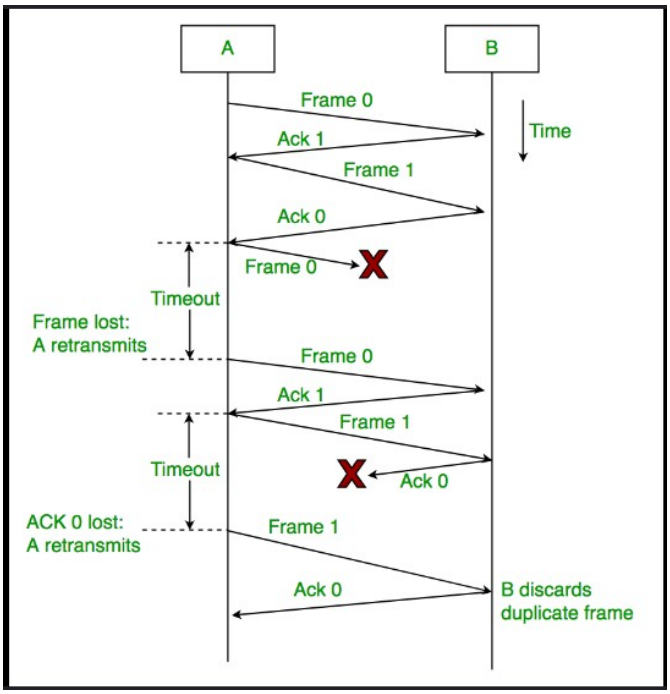
Stop and Wait is a fundamental flow control protocol used in computer networks for reliable data transmission. It operates on a simple principle where the sender transmits a data frame and then waits for an acknowledgment (ACK) from the receiver before sending the next frame. The process can be broken down into these steps:

Frame Transmission: The sender sends one frame at a time and waits for the receiver to confirm successful receipt through an acknowledgment. If no error occurs, the receiver processes the frame and sends an ACK back to the sender.

Acknowledgment: Once the sender receives the ACK, it proceeds to send the next frame in the sequence. If no acknowledgment is received within a specified timeout period, the sender assumes the frame was lost or corrupted and retransmits it.

Error Handling: Errors like lost or damaged frames can occur during transmission. If a frame is lost or an error is detected by the receiver, the absence of an acknowledgment triggers retransmission. Similarly, if an ACK is lost, the sender retransmits the same frame after the timeout.

Drawbacks: The major drawback of Stop and Wait is inefficiency in high-latency or high-bandwidth networks. Since the sender must wait for an acknowledgment after every frame, the utilization of the communication channel is low. The sender remains idle during the waiting period, which limits the throughput.



CODE:

SENDER---

```
import socket
import time
import random

SOURCE_ADDRESS = "011011"                #CAPITAL letters are
constant
DESTINATION_ADDRESS = "110110"
PAYLOAD_SIZE = 8
CHECKSUM_SIZE = 4

TIMEOUT = 2
MAX_RETRIES = 5

RECEIVER_IP = '127.0.0.1'
RECEIVER_PORT = 5005

PACKET_CORRUPTION_PROBABILITY = 0.8
DELAY_RANGE = (0.1, 0.5)              # Delay between 100ms and
500ms

def setWrapSum(sum):
    temp =sum
    if(sum > 0xF):
```

```
            temp = temp & 0xF0
            temp = temp>>4
            sum += temp
            sum = sum & 0x0F
        return sum

    def calculate_checksum(header):
        sum = 0
        for i in range(0,len(header),4):
            byte = header[i:i+4]
            sum += int(byte,2)
        wrapsum = setWrapSum(sum)
        checksum = (~wrapsum & 0xF)
        #print(f"Checksum at sender:{format(checksum,'04b')}")
        return format(checksum,'04b')

    def create_frame(seq_num, payload):

        length = len(payload)
        header =
    SOURCE_ADDRESS+DESTINATION_ADDRESS+"1000"+str(seq_num);
        frame_without_fcs = "0000000"+header+payload
        fcs = calculate_checksum(frame_without_fcs)
        frame = frame_without_fcs + fcs   #using checksum.....
        return frame

    def inject_errors(frame):
        if random.random() < PACKET_CORRUPTION_PROBABILITY:
    #[0,1)
            frame = list(frame)
            char_index = random.randint(0, len(frame) - 1)
            frame[char_index] = "1" if frame[char_index] == "0" else
    "0"
            frame = ''.join(frame)
        return frame

    def delay():
        delay = random.uniform(*DELAY_RANGE)
        time.sleep(delay)

    def send_data(file_path):
        # Initialize TCP socket
        sender_socket = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
        sender_socket.connect((RECEIVER_IP, RECEIVER_PORT))
        sender_socket.settimeout(TIMEOUT)
```

```
    seq_num = 0

    try:
        with open(file_path, 'r') as file:
            while True:
                payload = file.readline()
                if not payload:
                    print("All data has been sent.")
                    break  # EOF
                payload = payload.rstrip('\n')
                #print(f"DATA IS:{payload}")
                binarydata = bin(int(payload, 16))[2:]
                db = ""
                if(8 - len(binarydata)>0):
                    l = 8 - len(binarydata);
                    for i in range(l):
                        db +="0";
                db += (binarydata)

                #print(db)
                frame = create_frame(seq_num,db)
                retries = 0
                Frame = frame
                #print(f"SENDING FRAME : {Frame}")
                while retries < MAX_RETRIES:
                    delay()

                    if(retries > 3):
#after 2 retries the correct data will be sent...
                        corrupted_frame = Frame
                    else:
                        corrupted_frame = inject_errors(frame)

                    print("Sending data is :",corrupted_frame,"
",len(corrupted_frame))

sender_socket.sendall(corrupted_frame.encode('utf-8'))
                    print(f"Sent Frame Seq#: {seq_num}, Payload:
{db}")

                    try:
                        # Wait for ACK
                        ack_packet = sender_socket.recv(1024)
                        ak = ack_packet.decode('utf-8')
                        #print("AK-",ak)
```

```
                        if ak[0] == str(seq_num):
                            print(f"Received ACK for Seq:
{ak[0]}\n")
                            seq_num = 1 - seq_num
                            break
                        else:
                            print(f"Received invalid ACK:
{ack_packet}\n")
                    except socket.timeout:
                        retries += 1
                        print(f"Timeout waiting for ACK for Seq:
{seq_num}. Retrying ({retries}/{MAX_RETRIES})...\n")

                if retries == MAX_RETRIES:
                    print(f"Failed to receive ACK for Seq#:
{seq_num} after {MAX_RETRIES} attempts. Exiting.")
                    return

    except FileNotFoundError:
        print(f"File {file_path} not found.")
    finally:
        sender_socket.close()

if __name__ == "__main__":
    t = time.time()
    send_data("inputdata.txt")
    print("ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!")
    t1 = time.time() - t ;
    print(f"Total Time :{t1}")
```

It reads hexadecimal data from a file, converts it to binary, and encapsulates it into a frame with a header, payload, and checksum for error detection. The frame is then sent to the receiver, with a possibility of introducing errors for testing the retransmission process. If no acknowledgment (ACK) is received within a timeout, the sender retries sending the frame, up to a maximum of 5 attempts. The sequence number alternates between 0 and 1 to identify frames. The program also measures the total transmission time.

RECEIVER---

```
import socket

SOURCE_ADDRESS = "011011"
DESTINATION_ADDRESS = "110110"
PAYLOAD_SIZE = 8
```

```
CHECKSUM_SIZE = 4

RECEIVER_IP = '127.0.0.1'
RECEIVER_PORT = 5005


def setWrapSum(sum):
    temp =sum
    if(sum > 0xF):
        temp = temp & 0xF0
        temp = temp>>4
        sum += temp
        sum = sum & 0x0F
    return sum

def calculate_checksum(data):
    sum =0
    for i in range(0,len(data),4):
        byte = data[i:i+4]
        sum += int(byte,2)
    wrapsum = setWrapSum(sum)
    checksum = (~wrapsum & 0xF)
    #print(f"Checksum at RECV:{format(checksum,'04b')}")
    return format(checksum,'04b')

def receive_data():

    # Initialize TCP socket
    receiver_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    receiver_socket.bind((RECEIVER_IP, RECEIVER_PORT))
    receiver_socket.listen(1)
    print(f"Receiver is listening on {RECEIVER_IP}:
{RECEIVER_PORT}")

    conn, addr = receiver_socket.accept()
    print(f"Connected by {addr}")

    expected_seq_num = 0  # Initial expected sequence number

    while True:
        try:
            frame = conn.recv(1024)
            if not frame:
                continue
```

```
            Frame = frame.decode()
            print("Received data is :",Frame,"  ",len(Frame))
            #padding = Frame[0:7]
            src = Frame[7:13]
            dest = Frame[13:19]
            size = Frame[19:23]
            seq = Frame[23:24]
            data = Frame[24:32]
            fcs = Frame[32:36]

            calculated_checksum =
calculate_checksum(Frame[0:32])
            #print("fcs -----",fcs)
            if calculated_checksum != fcs:
                print("Checksum mismatch. Frame corrupted.
Discarding frame.")
                continue  # Discard the frame and do not send
ACK

            if src != SOURCE_ADDRESS or dest !=
DESTINATION_ADDRESS:
                print("Invalid source or destination address.
Discarding frame.")
                continue  # Invalid addresses

            if seq != str(expected_seq_num):
                print(f"Unexpected sequence number. Expected:
{expected_seq_num}, Received: {seq}. Discarding frame.")
                continue  # Discard out-of-order frame

            print(f"Received Frame Seq: {seq}, Payload: {data}")

            ack_packet = f"{expected_seq_num}ACK".encode()
            conn.sendall(ack_packet)
            print(f"Sent ACK for Seq#: {expected_seq_num}")

            # Update expected sequence number
            expected_seq_num = 1 - expected_seq_num

        except KeyboardInterrupt:
            print("\nReceiver shutting down.")
            break

    conn.close()
    receiver_socket.close()
```

```
if __name__ == "__main__":
    receive_data()
```

This Python program implements the receiver side of the Stop and Wait ARQ protocol over a TCP connection. It listens for incoming frames, extracts the header, payload, and checksum, and verifies the integrity of the received frame using a checksum function. If the checksum is valid, the frame is further checked for correct source/destination addresses and expected sequence number. Upon successfully receiving a valid frame, the receiver sends an acknowledgment (ACK) back to the sender. If the frame is corrupted or out of order, it is discarded, and no ACK is sent. The receiver updates the expected sequence number after each successful reception.

GO BACK-N ARQ–

Go-Back-N ARQ is a sliding window protocol used for reliable data transmission in computer networks. It allows the sender to transmit multiple frames before receiving an acknowledgment, but with a limitation on the number of unacknowledged frames, defined by the window size.

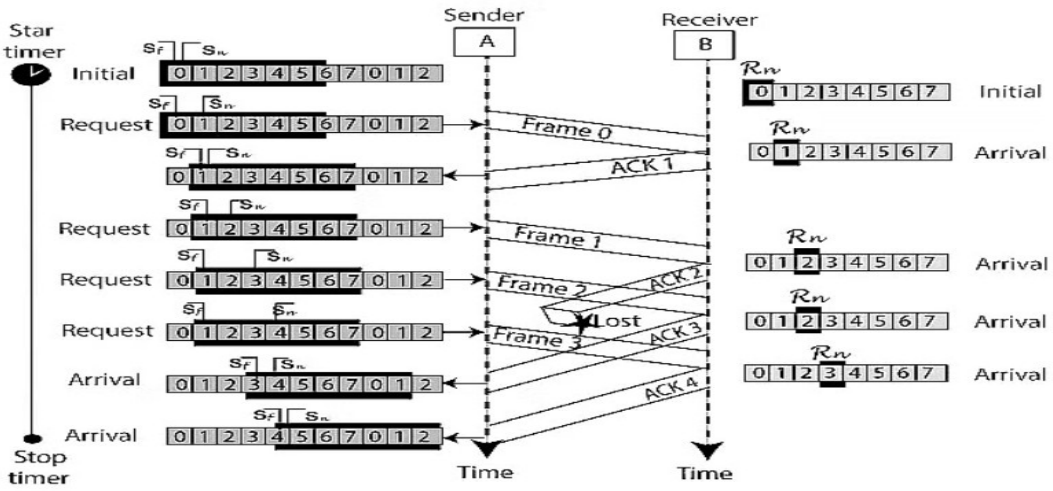
Sliding Window: The sender maintains a window of up to *N* frames (window size), which can be sent without waiting for individual acknowledgments.

Sequential Transmission: Frames are sequentially numbered. The sender can transmit up to *N* frames, but must wait for an acknowledgment (ACK) for the first frame in the window before moving the window forward.

ACK Handling: If an acknowledgment is received for a specific frame, the sender shifts the window forward, allowing new frames to be transmitted.

Error Handling: If a frame is lost or corrupted, the receiver discards that frame and all subsequent frames. The sender, upon timeout or receiving a NACK (negative acknowledgment), retransmits all frames starting from the erroneous frame (hence "Go-Back").

Efficiency: Go-Back-N is more efficient than Stop-and-Wait since it allows multiple frames to be in transit, but less efficient than Selective Repeat ARQ due to retransmission of potentially error-free frames.



CODE:

SENDER—

```
import socket
import threading
import time
import random
from check import *

# Constants
WINDOW_SIZE = 4
TOTAL_FRAMES = 10
TIMEOUT = 3  # seconds
HOST = 'localhost'
PORT = 12345
frameList = makeListOfFrames()  # having the frames list at one
place....
# print("ALL THE FRAMES")
# print(frameList)
lock = threading.Lock()

class Sender:
    def __init__(self):
        self.base = 0  # First unacknowledged frame
        self.next_seq_num = 0  # Next sequence number to send
        self.window = WINDOW_SIZE
        self.acks_received = [False] * TOTAL_FRAMES  # To track
received ACKs
        self.timer = None
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
```

```
        self.sock.settimeout(TIMEOUT)

    def start(self):
        while self.base < TOTAL_FRAMES:
            with lock:
                while self.next_seq_num < self.base +
self.window and self.next_seq_num < TOTAL_FRAMES:
                    self.send_frame(self.next_seq_num)
                    self.next_seq_num += 1

            self.start_timer()

            try:
                # Waiting for ACK
                ack, _ = self.sock.recvfrom(1024)
                ack = int(ack.decode())
                self.handle_ack(ack)
            except socket.timeout:
                print("Timeout! Resending frames...")
                self.resend_frames()

    def send_frame(self, frame_num):
        message = f"Frame {frame_num}"
        time.sleep(1)
        self.sock.sendto(message.encode(), (HOST, PORT))
        print(f"Sent: {message}")

    def handle_ack(self, ack):
        print(f"Received ACK for Frame {ack}")
        self.acks_received[ack] = True

        with lock:
            if ack == self.base:
                while self.base < TOTAL_FRAMES and
self.acks_received[self.base]:
                    self.base += 1
                self.stop_timer()

    def resend_frames(self):
        with lock:
            for i in range(self.base, min(self.base +
self.window, TOTAL_FRAMES)):
                self.send_frame(i)
        self.start_timer()

    def start_timer(self):
```

```
        if self.timer:
            self.timer.cancel()
        self.timer = threading.Timer(TIMEOUT,
self.resend_frames)
        self.timer.start()

    def stop_timer(self):
        if self.timer:
            self.timer.cancel()
            self.timer = None

if __name__ == "__main__":
    sender = Sender()
    t = time.time()
    sender.start()
    print("ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!")
    t1 = time.time() - t ;
    print(f"Total Time :{t1}")
```

This code implements the sender side of the **Go-Back-N ARQ** protocol using UDP. The sender transmits frames, up to the window size, while waiting for ACKs. It tracks sent frames using base and next_seq_num, resending frames when a timeout occurs. Each frame is sent by the send_frame method, and received ACKs are handled in handle_ack, where the sender adjusts the base to slide the window forward. If no ACK is received within the TIMEOUT period, it triggers resend_frames for all frames in the window. A timer manages retransmissions, and the process continues until all frames are acknowledged.

RECEIVER—

```
import socket
import random
import time
from check import *

# Constants
TOTAL_FRAMES = 10
HOST = 'localhost'
PORT = 12345
PACKET_CORRUPTED_PROBABILITY = 0.2 # probabilty for having the
corrupted data..

class Receiver:
    def __init__(self):
        self.expected_frame = 0
```

```
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
        self.sock.bind((HOST, PORT))

    def start(self):
        while self.expected_frame < TOTAL_FRAMES:
            frame, addr = self.sock.recvfrom(1024)
            frame_num = int(frame.decode().split()[1])
            print(f"Received: {frame.decode()}")

            if random.random() < 0.1:  # 10% chance of frame
loss
                print(f"Frame {frame_num} is lost!")
                continue

            if checkTheChecksum() <
PACKET_CORRUPTED_PROBABILITY:
                print(f"Frame {frame_num} is wrong!")
                continue
            if frame_num == self.expected_frame:
                print(f"ACK Sent for Frame
{self.expected_frame}")

self.sock.sendto(str(self.expected_frame).encode(), addr)
                self.expected_frame += 1
            else:
                print(f"Discarding frame {frame_num}, waiting
for {self.expected_frame}")

if __name__ == "__main__":
    receiver = Receiver()
    receiver.start()
```

This code implements the **receiver** side of the Go-Back-N ARQ protocol using UDP. The receiver listens for incoming frames and checks if the received frame number matches the expected frame (expected_frame). If a frame is lost (simulated with a 10% probability) or the checksum is incorrect (simulated with PACKET_CORRUPTED_PROBABILITY), the receiver discards the frame. When a correct and expected frame is received, the receiver sends an acknowledgment (ACK) back to the sender and increments the expected frame number. Out-of-order frames are discarded, ensuring that only in-sequence frames are accepted.

I have put all the error injection,making frames etc, important function in a file named check.py---

This is like this---

```
import time
import random

SOURCE_ADDRESS = "011011"                #CAPITAL letters are
constant
DESTINATION_ADDRESS = "110110"
PAYLOAD_SIZE = 8
CHECKSUM_SIZE = 4

TIMEOUT = 2
MAX_RETRIES = 5


PACKET_CORRUPTION_PROBABILITY = 0.6

def setWrapSum(sum):
    temp =sum
    if(sum > 0xF):
        temp = temp & 0xF0
        temp = temp>>4
        sum += temp
        sum = sum & 0x0F
    return sum

def calculate_checksum(header):
    sum =0
    for i in range(0,len(header),4):
        byte = header[i:i+4]
        sum += int(byte,2)
    wrapsum = setWrapSum(sum)
    checksum = (~wrapsum & 0xF)
    #print(f"Checksum at sender:{format(checksum,'04b')}")
    return format(checksum,'04b')
def checkTheChecksum():
    return random.random()
def create_frame(seq_num, payload):

    length = len(payload)
    header =
SOURCE_ADDRESS+DESTINATION_ADDRESS+"1000"+str(seq_num);
    frame_without_fcs = "0000000"+header+payload
    fcs = calculate_checksum(frame_without_fcs)
    frame = frame_without_fcs + fcs   #using checksum.....
    return frame
```

```
def inject_errors(frame):
    if random.random() < PACKET_CORRUPTION_PROBABILITY:
#[0,1)
        frame = list(frame)
        char_index = random.randint(0, len(frame) - 1)
        frame[char_index] = "1" if frame[char_index] == "0" else
"0"
        frame = ''.join(frame)
    return frame

def makeListOfFrames():
    file = open("inputdata.txt","r")
    line = file.readline().strip()
    frameList =[]
    i = 0;
    while(line):
        frameList.append(create_frame(i%4,line))
        line = file.readline().strip()
    return frameList
```

SELECTIVE REPEAT ARQ–

Selective Repeat ARQ is a protocol used in reliable data communication. It improves upon Go-Back-N by allowing the retransmission of only the specific erroneous or lost frames rather than all frames after an error. Here are the key points:

Window-Based: Both sender and receiver use a sliding window mechanism to manage the transmission and acknowledgment of multiple frames.

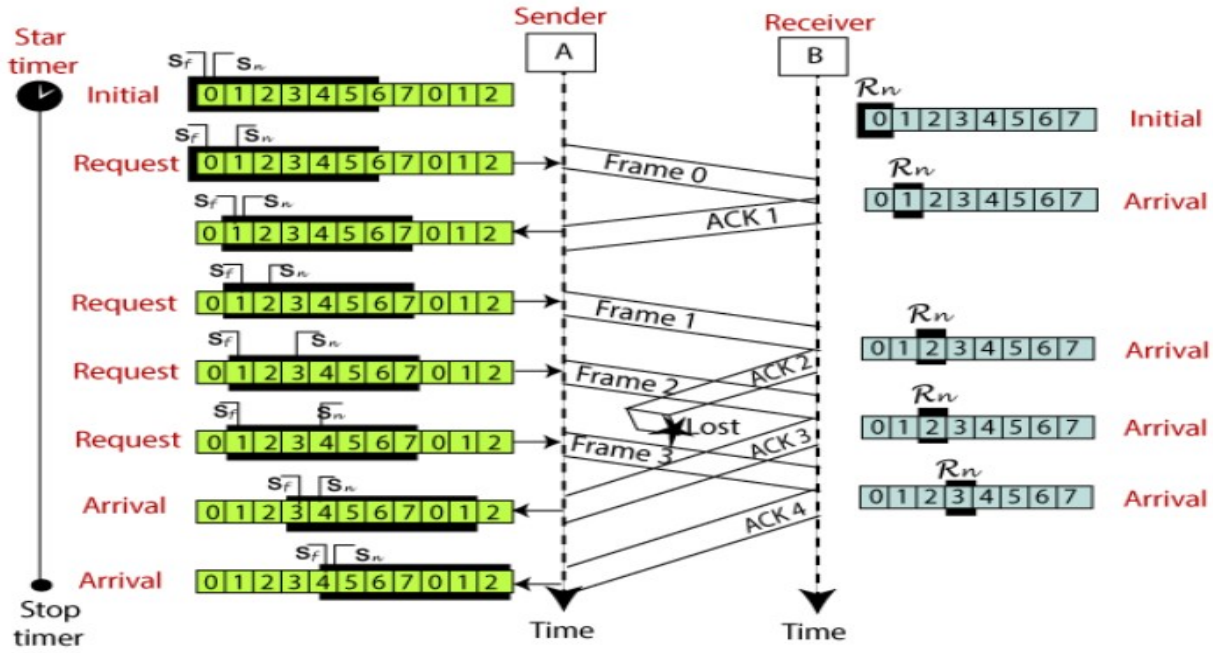
Selective Retransmission: Only frames that are detected as erroneous or lost are retransmitted, rather than all subsequent frames, as in Go-Back-N.

Receiver Buffering: The receiver can buffer out-of-order frames, allowing it to accept frames that arrive in a different order and then deliver them in sequence to the application.

Acknowledgment: The receiver sends an acknowledgment (ACK) for each frame individually. It can also send a negative acknowledgment (NACK) for frames that need retransmission.

Efficiency: By retransmitting only the affected frames, Selective Repeat reduces the overhead and improves efficiency compared to Go-Back-N, especially in environments with higher error rates.

Complexity: It is more complex to implement than Go-Back-N due to the need for maintaining a buffer for out-of-order frames and managing individual acknowledgments.



CODE—

SENDER—

```
import socket
import threading
import random
import time

WINDOW_SIZE = 4
PROBABILITY_CORRUPTION = 0.2  # Probability of frame corruption
TIMEOUT = 5  # seconds

class Frame:
    def __init__(self, seq_num, data):
        self.seq_num = seq_num
        self.data = data

class Sender:
```

```
    def __init__(self, receiver_ip, receiver_port):
        self.receiver_ip = receiver_ip
        self.receiver_port = receiver_port
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
        self.sock.settimeout(TIMEOUT)
        self.window = []
        self.next_seq_num = 0
        self.base = 0
        self.frames = self.readfromfile("inputdata.txt")
        self.TOTAL_FRAMES = len(self.frames)
        self.ack_received = [False] * self.TOTAL_FRAMES

    def readfromfile(self, frame_file):

        frames = []
        with open(frame_file, 'r') as f:
            lines = f.readlines()
            for i, line in enumerate(lines):
                frames.append(Frame(i, line.strip()))
        return frames

    def send_frame(self, frame):
        if random.random() < PROBABILITY_CORRUPTION:
            print(f"Frame {frame.seq_num} corrupted")
            corrupted_frame = Frame(frame.seq_num, "CORRUPT")
            time.sleep(1)
            self.sock.sendto(f"{corrupted_frame.seq_num}:
{corrupted_frame.data}".encode(), (self.receiver_ip,
self.receiver_port))
        else:
            print(f"Sending frame {frame.seq_num}:
{frame.data}")
            time.sleep(1)
            self.sock.sendto(f"{frame.seq_num}:
{frame.data}".encode(), (self.receiver_ip, self.receiver_port))

    def resend_frame(self, seq_num):
        for frame in self.window:
            if frame.seq_num == seq_num:
                print(f"Resending frame {seq_num}")
                self.send_frame(frame)

    def receive_ack(self):
        while self.base < self.TOTAL_FRAMES:
            try:
```

```
                ack, _ = self.sock.recvfrom(1024)
                ack = ack.decode()
                print(f"Received {ack}")
                ack_num, status = ack.split(":")
                ack_num = int(ack_num)

                if status == "ACK":
                    self.ack_received[ack_num] = True
                    if ack_num == self.base:
                        while self.base < self.TOTAL_FRAMES and
self.ack_received[self.base]:
                            self.base += 1
                            if self.next_seq_num <
self.TOTAL_FRAMES:
                                frame =
self.frames[self.next_seq_num]
                                self.window.append(frame)
                                self.send_frame(frame)
                                self.next_seq_num += 1
                            self.window = self.window[1:]#Slide
window
                elif status == "NACK":
                    print(f"Received NACK for frame {ack_num}")
                    self.resend_frame(ack_num)

            except socket.timeout:
                print("Timeout, resending unacknowledged
frames")
                for i in range(self.base, self.base +
WINDOW_SIZE):
                    if i < self.TOTAL_FRAMES and not
self.ack_received[i]:
                        self.resend_frame(i)

    def start(self):
        for i in range(min(WINDOW_SIZE, self.TOTAL_FRAMES)):
            frame = self.frames[i]
            self.window.append(frame)
            self.send_frame(frame)
            self.next_seq_num += 1

        ack_thread = threading.Thread(target=self.receive_ack)
        ack_thread.start()
        ack_thread.join()

if __name__ == "__main__":
```

```
    sender = Sender("localhost", 12345)
    t = time.time()
    sender.start()
    print("ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!")
    t1 = time.time() - t ;
    print(f"Total Time :{t1}")
```

This Python code implements a **sender** for a protocol with **Selective Repeat ARQ** using UDP. The sender reads frames from a file, simulates frame corruption with a specified probability, and manages a sliding window to send frames. It uses a separate thread to handle ACKs and NACKs from the receiver. If a frame is acknowledged (ACK), the sender slides the window forward and sends the next frame. If a NACK is received, the sender resends the specified frame. The sender also handles timeouts by resending unacknowledged frames. The process continues until all frames are sent and acknowledged.

RECEIVER—

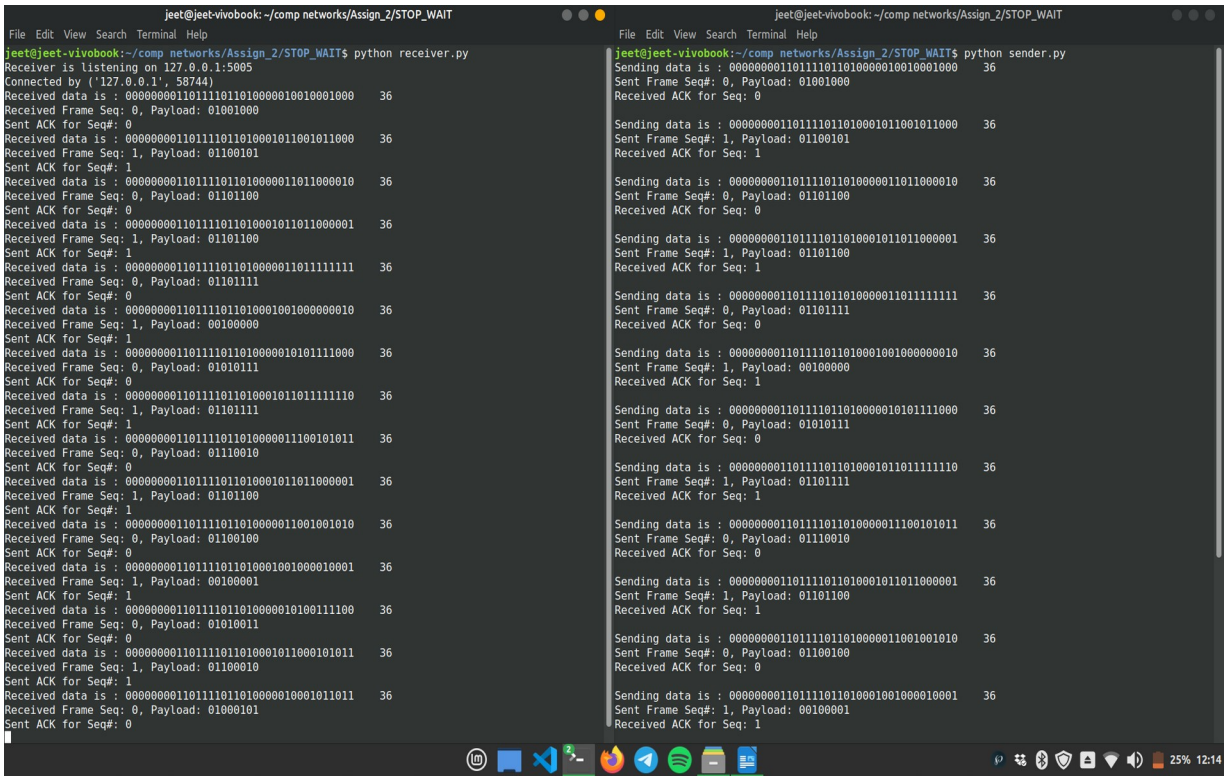
```
import socket
import random

WINDOW_SIZE = 4
PROBABILITY_CORRUPTION = 0.2  # Probability of corrupted
ACK/NACK

class Receiver:
    def __init__(self, ip, port):
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
        self.sock.bind((ip, port))
        self.window = [-1] * WINDOW_SIZE
        self.expected_seq_num = 0

    def send_ack(self, addr, seq_num, status):
        print(f"Sending {status} for frame {seq_num}")
        if random.random() < PROBABILITY_CORRUPTION:
            print(f"ACK {seq_num} corrupted")
            return
        self.sock.sendto(f"{seq_num}:{status}".encode(), addr)

    def receive_frame(self):
        while True:
```

```
            frame, addr = self.sock.recvfrom(1024)
            frame = frame.decode()
            seq_num, data = frame.split(":")
            seq_num = int(seq_num)

            if data == "CORRUPT":
                print(f"Frame {seq_num} is corrupt, sending
NACK")
                self.send_ack(addr, seq_num, "NACK")
            elif seq_num == self.expected_seq_num:
                print(f"Received correct frame {seq_num},
sending ACK")
                self.send_ack(addr, seq_num, "ACK")
                self.expected_seq_num += 1
            else:
                print(f"Out of order frame {seq_num}, sending
ACK")
                self.send_ack(addr, seq_num, "ACK")

if __name__ == "__main__":
    receiver = Receiver("localhost", 12345)
    receiver.receive_frame()
```

This code implements a **Selective Repeat ARQ** receiver using UDP. The receiver listens for incoming frames and maintains a window to manage frame sequence numbers. It handles frames based on their sequence number: if the frame is correct and in order, it sends an ACK (acknowledgment) and increments the expected sequence number. If a frame is detected as corrupt, it sends a NACK (negative acknowledgment). Frames arriving out of order receive an ACK but are not processed further. ACKs and NACKs can be randomly corrupted based on a defined probability.

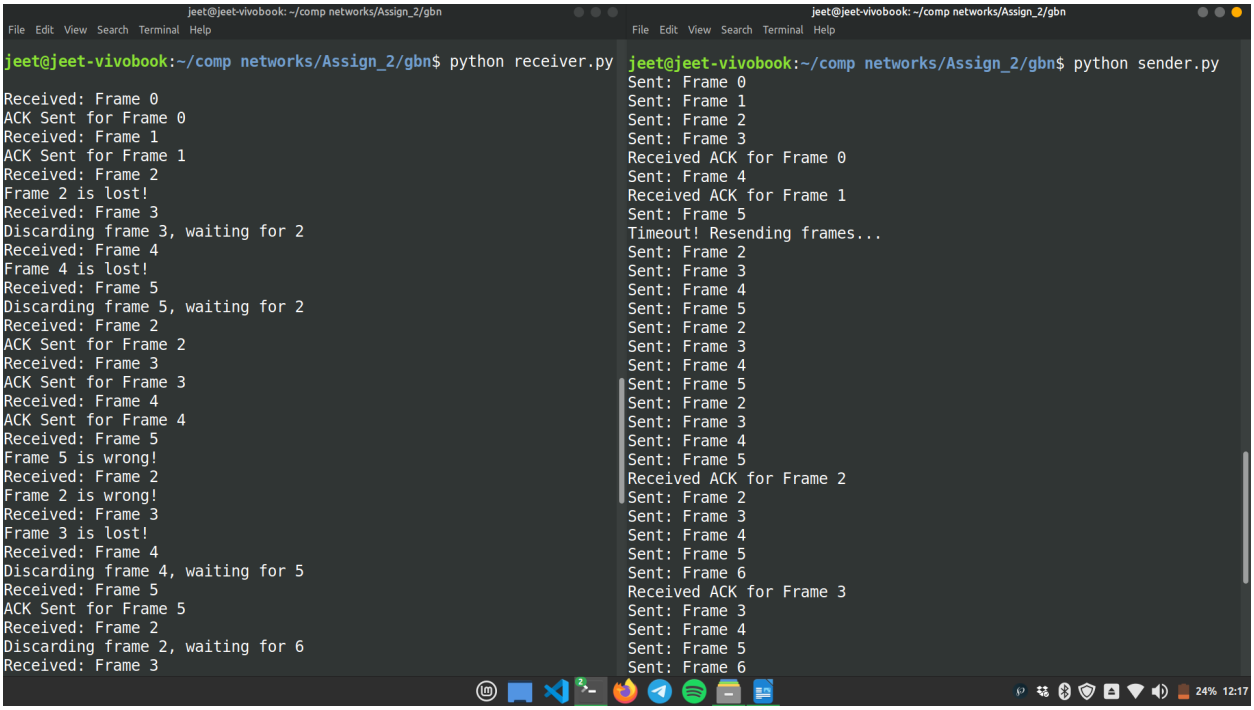
Compare time betweenthepropagation of a packet and reception of its ACK.Compare efficiency of the above approaches for different probability (0.1- 0.5) of an error or delay in the transmission of a packet or in its acknowledgment.

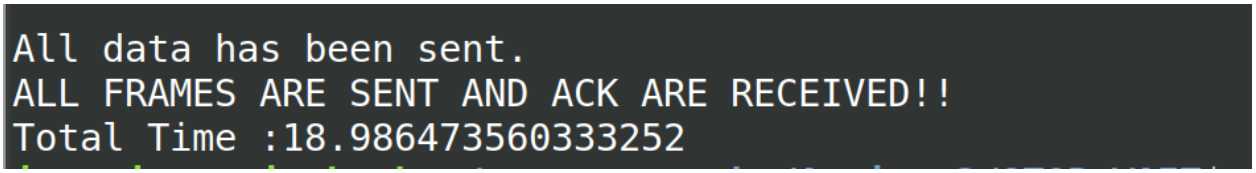
STOP AND WAIT OUTPUT



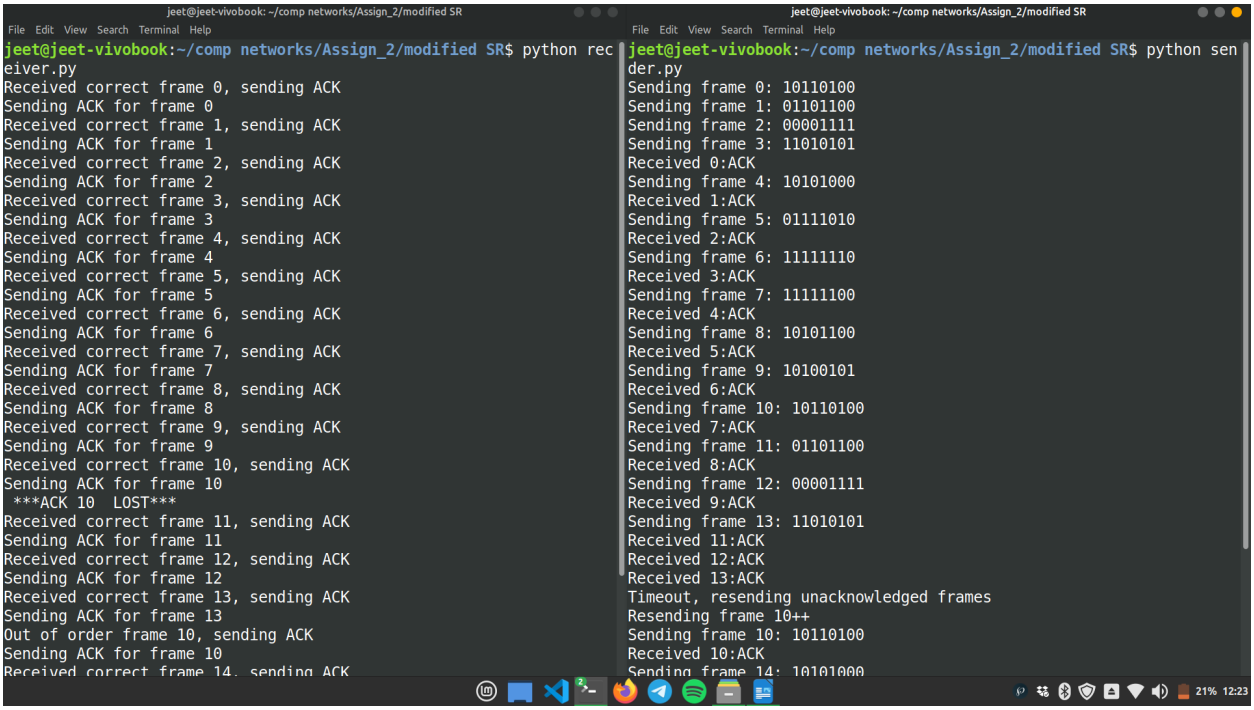

```
Received ACK for Frame 8
Received ACK for Frame 9
ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!
Total Time :44.05464315414429
```

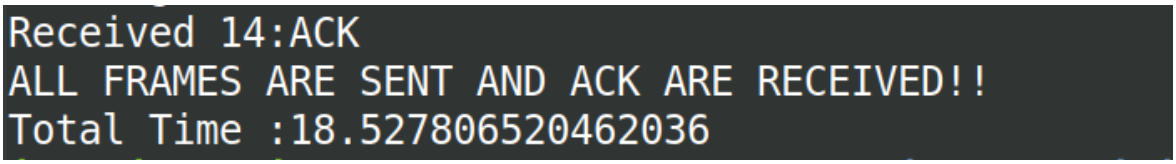
GO –BACK N ARQ


jeet@jeet-vivobook: ~/comp networks/Assign_2/gbn
File Edit View Search Terminal Help
jeet@jeet-vivobook:~/comp networks/Assign_2/gbn$ python receiver.py
Received: Frame 0
ACK Sent for Frame 0
Received: Frame 1
ACK Sent for Frame 1
Received: Frame 2
Frame 2 is lost!
Received: Frame 3
Discarding frame 3, waiting for 2
Received: Frame 4
Frame 4 is lost!
Received: Frame 5
Discarding frame 5, waiting for 2
Received: Frame 2
ACK Sent for Frame 2
Received: Frame 3
ACK Sent for Frame 3
Received: Frame 4
ACK Sent for Frame 4
Received: Frame 5
Frame 5 is wrong!
Received: Frame 2
Frame 2 is wrong!
Received: Frame 3
Frame 3 is lost!
Received: Frame 4
Discarding frame 4, waiting for 5
Received: Frame 5
ACK Sent for Frame 5
Received: Frame 2
Discarding frame 2, waiting for 6
Received: Frame 3
jeet@jeet-vivobook: ~/comp networks/Assign_2/gbn
File Edit View Search Terminal Help
jeet@jeet-vivobook:~/comp networks/Assign_2/gbn$ python sender.py
Sent: Frame 0
Sent: Frame 1
Sent: Frame 2
Sent: Frame 3
Received ACK for Frame 0
Sent: Frame 4
Received ACK for Frame 1
Sent: Frame 5
Timeout! Resending frames...
Sent: Frame 2
Sent: Frame 3
Sent: Frame 4
Sent: Frame 5
Sent: Frame 2
Sent: Frame 3
Sent: Frame 4
Sent: Frame 5
Sent: Frame 2
Sent: Frame 3
Sent: Frame 4
Sent: Frame 5
Received ACK for Frame 2
Sent: Frame 2
Sent: Frame 3
Sent: Frame 4
Sent: Frame 5
Sent: Frame 6
Received ACK for Frame 3
Sent: Frame 3
Sent: Frame 4
Sent: Frame 5
Sent: Frame 6
24% 12:17


All data has been sent.
ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!
Total Time :18.986473560333252

SELECTIVE REPEAT ARQ





● *Compare efficiency of the above approaches without error or lost frame*.

STOP AND WAIT

```
Received ACK for Frame 8
Received ACK for Frame 9
ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!
Total Time :44.05464315414429
```

GO –BACK N ARQ

```
All data has been sent.
ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!
Total Time :18.986473560333252
```

SELECTIVE REPEAT ARQ

```
Received 14:ACK
ALL FRAMES ARE SENT AND ACK ARE RECEIVED!!
Total Time :18.527806520462036
```

Comments:

We can see that the most efficient mechanism is selective repeat as it takes least time for complete transmission. All mechanism is sending about the 15 (8 bit)data.

Also is the channel is ideal(no error) **selective repeat** is fastest and **stop and wait** is slowest.