COMPUTER NETWORKS LAB

REPORT ASSIGNMENT-2

NAME: SOHAM LAHIRI

CLASS: BCSE UG-III 5TH SEMESTER

ROLL NO: 002210501107

GROUP: A3

SUBMISSION DATE: 30/09/2024

Problem Statement: Design and implement flow control mechanisms of Logical Link Control of Data Link Layer within a simulated network environment.

Sender Program: The Sender program consist of following methods:

● **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

● **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.

FlowControl: implement the following flow control approaches using the Sender and Receiver program
and their methods:

● **Stop and Wait:** The Sender program calls **Send()** method to transmit one frame at a time slot to the receiver. After that Sender will wait for the ACK frame from the receiver as a response to a successful receipt of the data frame. The interval between sending a message and receiving an acknowledgment is known as the sender's waiting time, and the sender is idle during this time.
After receiving an ACK, the sender will send the next data packet to the receiver, and so on, as long as the sender has data to send. If the data frame is lost due to delay or discarded due to error no acknowledgement will be transmitted from the receiver and thus there will be timeout.
After timeout, Sender will retransmit the same packet again.

● **Go-Back-N ARQ:** The **Send()** method of the Sender program will take N as input. The Sender’s window size is N. So that the sender can send N frames which are equal to the window size. The size of the receiver's window is 1. Once the entire window is sent, the sender then waits for a cumulative acknowledgement to send more packets. That is, receiving acknowledgment for frame i means the frames i-1, i-2, and so on are acknowledged as well. You can specify that acknowledgement no. in the ACK frame. Receiver receives only in-order packets and discards out-of-order frames and corrupted frames (checked by CRC or Checksum). In case of packet loss, the entire window would be re-transmitted.

● **Selective Repeat ARQ:** The **Send()** method of the Sender program will take N as input. The Sender’s window size and Receiver’s window size both are N. The sender sends N frames and the receiver acknowledges all frames whether they were received in order or not. Remember that type of acknowledgement here is not cumulative. It uses Independent Acknowledgement to acknowledge the packets. In this case, the receiver maintains a buffer to contain out-of-order packets and sorts them. Receiver discards only corrupted frames (checked by CRC or Checksum). The sender selectively re-transmits the lost packet and moves the window forward. That means Sender retransmits unacknowledged packets after a timeout or upon a NAK (if NAK is employed).

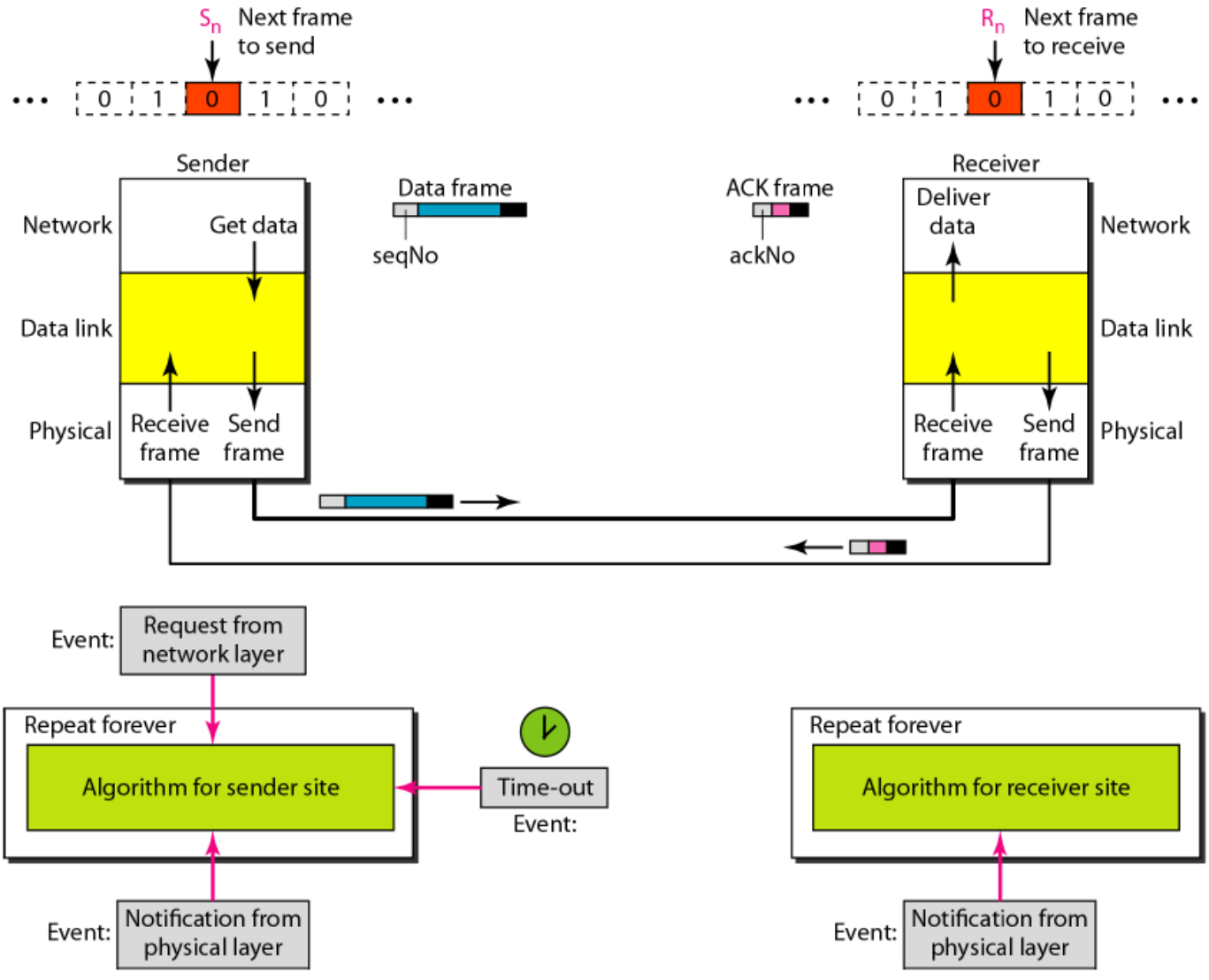
DESIGN

Theory:

Stop-and-Wait ARQ (Automatic Repeat Request):

Stop-and-Wait ARQ is one of the simplest forms of error control in data communication. In this protocol, the sender transmits a single data frame and then waits for an acknowledgment (ACK) from the receiver before sending the next one. The sender essentially "stops" until it "waits" for a response. If the acknowledgment is received, the sender proceeds with the next frame. However, if an acknowledgment is not received within a specified timeout period, the sender assumes the frame was lost or corrupted and retransmits the same frame. This ensures reliable data transmission. While Stop-and-Wait ARQ is easy to implement, its major drawback is inefficiency, particularly over long-distance communication links, as the sender remains idle while waiting for the acknowledgment, underutilizing available bandwidth. This inefficiency grows with increased propagation delays in the network.

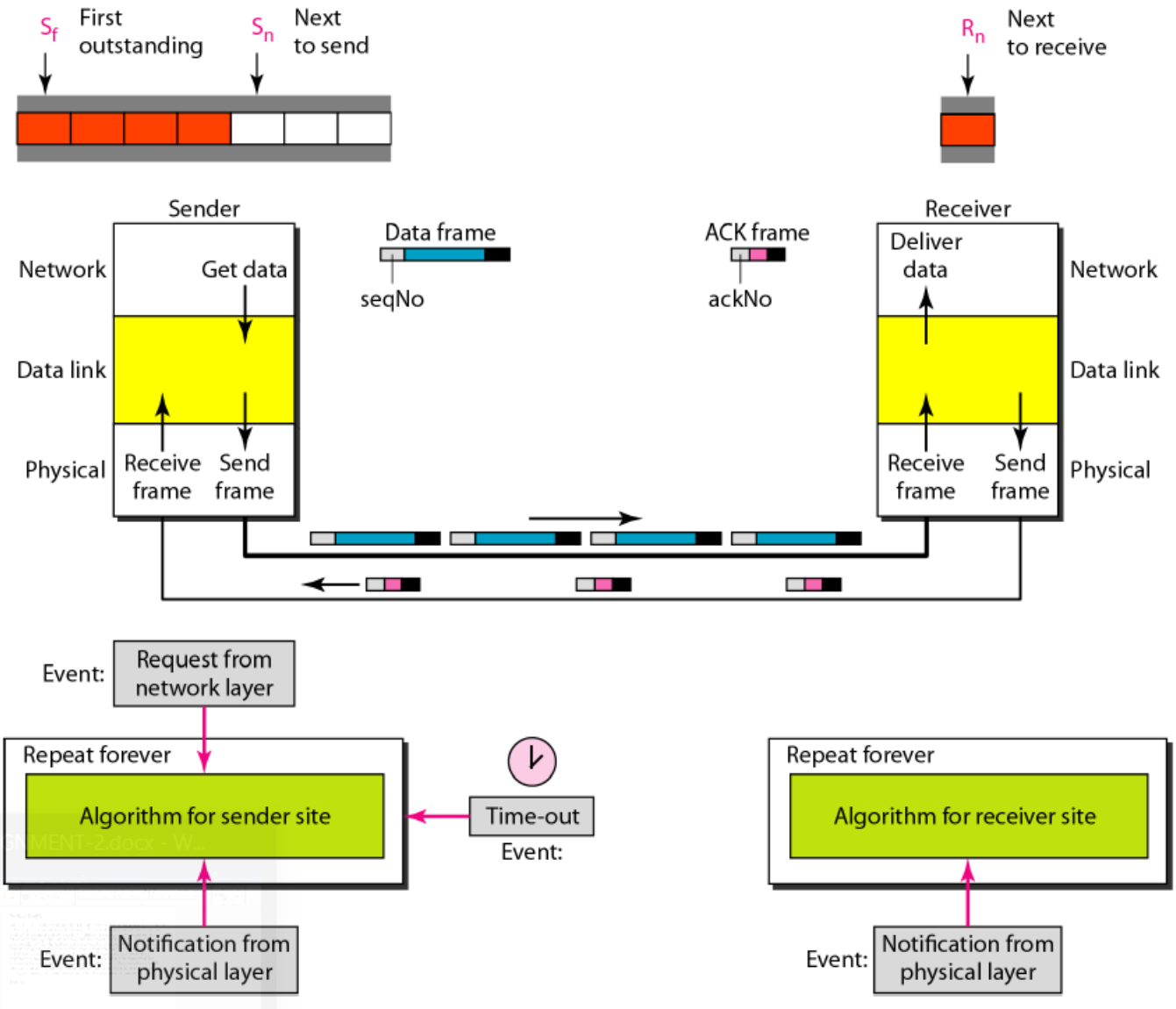
Example:



Go-Back-N ARQ:

Go-Back-N ARQ improves upon the Stop-and-Wait approach by allowing the sender to transmit multiple frames without waiting for individual acknowledgments. In this protocol, the sender can send up to **N frames** in sequence before stopping to wait for an acknowledgment. The receiver checks each incoming frame and sends an acknowledgment for the last correctly received frame in sequence. If an error is detected in a frame, the receiver will discard that frame and all subsequent frames until the erroneous frame is correctly received. The sender, upon receiving a negative acknowledgment (NACK) or timeout, "goes back" and retransmits the erroneous frame along with all subsequent frames, even if they were transmitted correctly earlier. This retransmission process can cause inefficiencies when errors occur, as frames that were transmitted without errors may be needlessly resent.

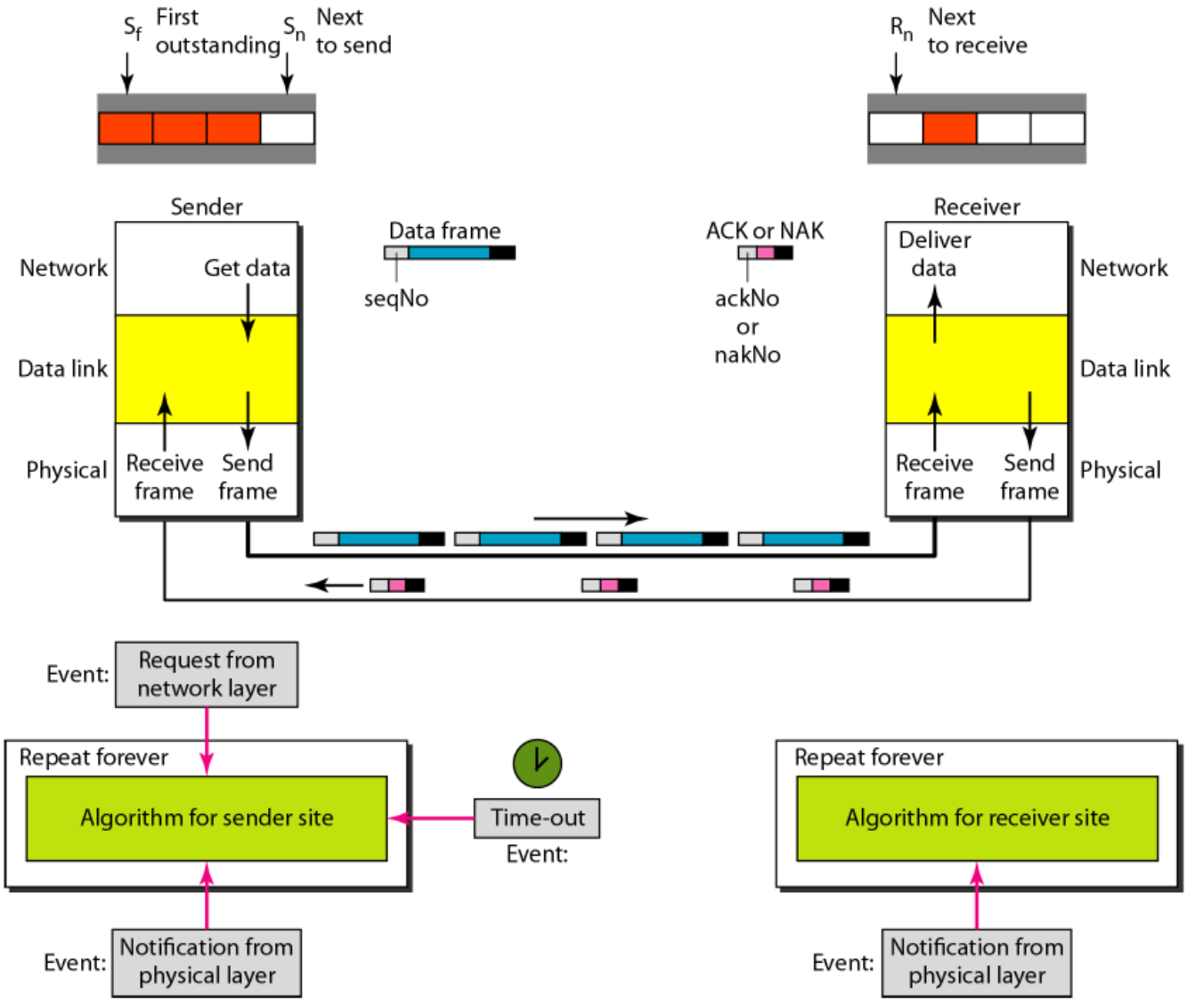
Example:



Selective Repeat ARQ

Selective Repeat ARQ is a more efficient and sophisticated error control method than Go-Back-N. Like Go-Back-N, Selective Repeat allows the sender to send multiple frames before needing acknowledgments. However, in Selective Repeat, the sender retransmits only the specific frames that were detected as erroneous or lost, rather than retransmitting all subsequent frames as in Go-Back-N. The receiver stores frames that arrive out of order and waits for the missing frames to arrive before delivering the data in the correct order to the upper layers. This reduces redundant retransmissions and makes Selective Repeat more efficient, especially in noisy or unreliable networks. The challenge of Selective Repeat lies in its complexity, as both the sender and receiver need more buffer space and mechanisms to manage out-of-order frames and acknowledgments. Despite this, it provides better utilization of bandwidth and minimizes retransmission overhead.

Example:



Sender Program:
This program simulates a **reliable data transmission** system between a sender and a receiver using a sliding window mechanism for controlling the flow of data and handling error recovery. It allows the sender to transmit multiple frames of data at once, ensuring efficient use of network bandwidth. The receiver sends back acknowledgments (ACKs) for successfully received frames, and the sender retransmits any frames that were lost or corrupted during transmission.
The sender reads data from a file, breaks it into frames, and transmits them over a network connection. A sliding window is used to manage the frames in transit and ensure that the sender does not overwhelm the receiver with too many frames at once. The sender waits for acknowledgments from the receiver and retransmits frames that are not acknowledged within a set timeout period. This process ensures that all data is successfully transmitted, even in the presence of errors or packet loss.
The program employs **multi-threading** to handle the simultaneous tasks of sending data frames and receiving acknowledgments. It uses a timeout mechanism to detect when frames need to be resent due to errors or delays in the network.

Process Flow:

- **Sending Data**: The sender reads data from a file, creates frames, and sends them to the receiver. The sender is allowed to send multiple frames without waiting for an acknowledgment for each one, within the limits of a sliding window.
- **Acknowledgment Handling**: The receiver acknowledges each frame that is successfully received. If a frame is lost or corrupted, the sender will retransmit the missing frames when a timeout occurs.
- **Timeout and Retransmission**: If an acknowledgment is not received for a frame within the timeout period, the sender retransmits the unacknowledged frames, ensuring data integrity.

Input Format:

- **File**: The path to the file containing the data to be transmitted.
- **Host and Port**: The IP address and port number for network communication with the receiver.

Output Format:

- **Sent Frames**: Displays the frames being transmitted to the receiver.
- **Acknowledgments**: Logs received acknowledgments for successfully transmitted frames.
- **Resent Frames**: Logs retransmission of frames that did not receive an acknowledgment within the timeout period.

This program demonstrates the basic principles of reliable data transmission using flow control, error detection, and retransmission mechanisms.

Receiver Program:

This program simulates the process of **error detection in data communication** by receiving binary data over a network and verifying its integrity using **Cyclic Redundancy Check (CRC)**. The receiver program establishes a connection with the sender, listens for incoming data frames, and deserializes them for further processing. Upon receiving a frame, it checks for transmission errors by comparing the computed CRC value with the one included in the frame.

The receiver maintains an expected sequence number to ensure that frames are processed in the correct order. If the received data frame is found to be error-free and matches the expected sequence number, the receiver sends back an acknowledgment (ACK) for that frame. In case of a CRC error or an out-of-order frame, the receiver identifies the issue and does not send an acknowledgment, prompting the sender to handle retransmission accordingly.

The program employs **multi-threading** to allow the receiver to continuously listen for incoming data without blocking, enhancing responsiveness and allowing for efficient error handling.

Process Flow:

- **Receiving Data**: The receiver listens for incoming frames, decodes them, and extracts the sequence number, data, and CRC.
- **Error Checking**: The program uses CRC to verify the integrity of the received data. If the data is valid and in the correct order, it proceeds to send an acknowledgment.
- **Acknowledgment Handling**: The receiver sends an ACK frame back to the sender for successfully received frames, ensuring the sender knows which frames have been acknowledged.

Input Format:

- **Host and Port**: The IP address and port number for establishing a connection with the sender.

Output Format:

- **Received Messages**: Displays the sequence number and data of the received frames.
- **CRC Results**: Logs whether the received data passed or failed the CRC check.
- **Acknowledgments**: Logs the acknowledgment sent back to the sender.

This program demonstrates fundamental principles of reliable data transmission, emphasizing the importance of error detection and acknowledgment mechanisms in maintaining data integrity.

How It Works:

1. **Network Communication:**
   - Socket Creation: The receiver program creates a TCP socket using the socket library.
   - Establishing Connection: It connects to the sender program using the specified IP address and port number (st.ADDR).
   - Listening for Data: The receiver waits for incoming data frames to be transmitted by the sender.

2. **Data Reception:**
   - Continuous Listening: The receiver runs in a loop, continuously listening for incoming frames until the transmission is complete.
   - Receiving Frames: When a frame is received, it is decoded to extract the following components:
     - Sequence Number (recv_n): The number that indicates the order of the frame.
     - Payload/Data (data): The actual content being transmitted.
     - CRC Value (crc): The Cyclic Redundancy Check value sent for error detection.

3. **Error Detection:**
   - CRC Computation: The program computes the CRC for the received data using the op.crc4itu(data) function.
   - Comparison: It compares the computed CRC with the received CRC value:
     - If the values match, the data is considered error-free.
     - If they do not match, an error is detected, and the program logs a CRC failure.
   - Sequence Number Check: The program verifies that the received sequence number matches the expected sequence number (rn). This ensures that frames are processed in the correct order.

4. **Acknowledgment Handling:**
   - Sending ACK: If the received frame is valid (no CRC errors and the correct sequence number):
     - The receiver sends an acknowledgment (ACK) frame back to the sender using the send_Ack() function.
     - The ACK frame contains the current expected sequence number, indicating that the sender can safely transmit the next frame.
   - Error Handling: If the data is corrupted or out of order:
     - The receiver does not send an ACK, prompting the sender to potentially retransmit the lost or erroneous frames.

5. **Output Results:**
    - Logging Received Data: The program prints the received sequence number and data to the console.
    - Error Status Output: It logs whether the received data was error-free (indicating successful receipt and verification) or contained errors.

Input Format:

- Host and Port: The IP address and port number where the receiver listens for incoming data frames.

Output Format:

- Received Messages: Displays the sequence number and data of each received frame.
- CRC Results: Logs the results of the CRC check, indicating success or failure.
- Acknowledgment Status: Logs whether an acknowledgment was sent for valid frames.

This receiver program operates in conjunction with the sender program, effectively ensuring the integrity and reliability of data transmitted over the network. It demonstrates the practical application of error detection techniques, particularly the use of CRC, in maintaining robust communication in data transmission systems.

Implementation:

Stop and Wait:

Sender Program:

```
# Assumptions:
# 1. Resending will resend without errors
# 2. Error only occurs in data part

import socket
import time
import random
import threading
import operations as op
import stats as st

sn = 0  # Sequence number

# To calculate timeout
TIMEOUT_LIMIT = 2
time1 = 0
time2 = 0

sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sender.bind(st.ADDR)

canSend = True

# Copy to store incase we need to resend
copy_sn = 0
copy_data = ""

def send(conn, data):

    global canSend, time1, sn, copy_sn, copy_data

    # Sending frames
    if canSend == True:

        # Making the frame
        frame = op.makeFrame(sn, data)
        print(f"[ENCODING] Encoded frame: {frame}")
```

```
        #time.sleep(random.randint()% 6)
        frame = op.noisychannel(frame)
        print(f"[NOISY] Frame after noise: {frame}")

        # Sending frame
        conn.send(frame.encode())

        # Storing Frame
        copy_data = data
        copy_sn = sn

        # Starting timer
        time1 = time.time()

        sn += 1

        canSend = False

    if data == "q":
        return

    # Receing ACK
    recv_Ack(conn)

    # Resending Frame
    while canSend == False:
        frame = op.makeFrame(copy_sn, copy_data)
        frame = op.noisychannel(frame)
        print(f"[RESENDING] Resending frame: {frame}")
        time1 = time.time();
        conn.send(frame.encode())

        # Receving Ack for the resent frame
        recv_Ack(conn)
def recv_Ack(conn):

    global sn, canSend, copy_data, copy_sn, time2
    print("[Reciving ACK].....")

    # Receiving ACK Frame
    try:
        conn.settimeout(0.5)    # Setting timeout time
        ack_frame = conn.recv(20).decode()
    except:
```

```
        # Timeout has occured, so we should resend the frame
        print("---[TIMEOUT OCCURED]----")
        return

    ackNo, _, data,_ = op.receiveFrame(ack_frame)

    # Checking if ACK Valid
    if ackNo == copy_sn and data == '11110000':

        # Stopping timer
        time2 = time.time()

        print(f"[ACK RECV] ACK {ackNo} successfully received.")

        # Purging data
        copy_sn = 0
        copy_data = ""

        canSend = True
        print("[TRANSACTION COMPLETED]")
        return
def start():

    # Listening
    sender.listen()
    print(f"[LISTENING] Server is listening on {st.HOST_IP}")

    # Accepting receiver
    conn, addr = sender.accept()
    print(f"[CONNECTED] Connected to Process Id: {addr}")

    while True:
        data = input('[INPUT] Enter data to send: ')

        send(conn, data)

        if data == 'q':
            print("[CLOSING] Closing the sender....")
            conn.close()
            break

        print("------------------------------------------------")

start()
```

Explanation:

The code represents a simple sender implementation for a socket-based communication system, where data is sent over a network connection with error handling mechanisms (such as acknowledging received data and resending if necessary). Below is a detailed explanation of each part of the code, the assumptions made, and the overall functionality of the sender:

1. Assumptions:

- **Resending will resend without errors:** This implies that the resend mechanism is expected to work correctly if a timeout occurs.
- **Error only occurs in the data part:** This suggests that the error handling will focus primarily on data corruption rather than issues in the network connection itself.

2. Imports and Initial Setup:

- The necessary modules for socket communication (socket), time tracking (time), randomness (random), and threading (threading) are imported.
- The code also imports custom modules operations and stats, which contain functions for handling frame operations and statistics, including address constants.

3. Global Variables:

- sn (sequence number) is initialized to 0 and will be used to keep track of the frames sent.
- TIMEOUT_LIMIT is set to 2 seconds, defining the duration to wait before considering a transmission as failed.
- time1 and time2 will track the sending and acknowledgment times.

4. Socket Initialization:

- A TCP socket (SOCK_STREAM) is created and bound to an address defined in stats module, allowing it to listen for incoming connections.

5. Sending and Acknowledgment Logic:

- canSend is a flag that indicates whether a new frame can be sent.
- copy_sn and copy_data are used to store the sequence number and data of the last sent frame in case a resend is needed.

6. The send Function:

- This function handles the sending of frames and manages acknowledgments and potential resending.

  a. Frame Creation and Sending:
    - If canSend is True, the function constructs a frame using op.makeFrame(sn, data).
    - The frame is then passed through a simulated noisy channel function (op.noisychannel(frame)), which introduces errors or noise to the frame.
    - The modified frame is sent over the socket connection.

  b. Storing Frame for Resend:

    - The original data and sequence number are stored for possible resending if acknowledgment is not received.

  c. Timer Start:

    - The sending time is recorded, and the sequence number is incremented. The canSend flag is set to False, indicating that a frame is in transit.

  d. Acknowledgment Handling:

    - The function waits for an acknowledgment (ACK) from the receiver. If the ACK is not received within the timeout, the sender will attempt to resend the last frame.

  e. Resending Logic:

    - If the ACK is not received, the sender enters a loop to resend the last frame until an acknowledgment is received.

7. The recv_Ack Function:

- This function listens for an acknowledgment frame from the receiver.

  a. ACK Reception:

    - The function sets a timeout of 0.5 seconds for receiving the ACK. If no ACK is received within this period, it indicates a timeout, and the function returns to allow for a resend.

  b. Validating ACK:

- The received ACK frame is processed to extract the acknowledgment number and other information.
- If the ACK number matches the sequence number of the sent frame and the data is valid ('11110000' is presumably a predefined valid acknowledgment format), it confirms successful receipt.

c. Resetting State:

- Upon successful acknowledgment, the function resets the stored values and allows new frames to be sent.

8. The start Function:

This function initializes the listening process for incoming connections.

a. Listening for Connections

- The server begins listening for incoming connections and accepts a connection from a client, storing the connection object in conn.

b. Input Loop:

- A loop prompts the user to enter data to send. This data is passed to the send function for transmission.
- If the input is 'q', the server closes the connection and exits the loop.

Receiver Program:

```
import socket
import time
import operations as op
import stats as st

rn = 0 # Sequence number

receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)
# receiver.settimeout(3)

def isCorrupted(data, crc):
    if op.crc4itu(data) != crc:
        return True
    else:
        return False

def Recv():
    global rn

    while True:

        print("[LISTENING] Receiver is listening....")
        frame = receiver.recv(20).decode()
        recv_n, _, data, crc = op.receiveFrame(frame)
        print(f"[RECV] Received message: {data}")

        # Checking if disconnect statement is there
        if data == 'q':
            print("[CLOSING] Closing the receiver....")
            receiver.close()
            return

        if recv_n == rn:

                if isCorrupted(data, crc) == False: # No crc error
                    print(f"[CRC SUCCESS] {op.crc4itu(data)} and {crc}")
                    send_Ack()
                    rn += 1
                    print(f"[ACK SENT] Sent ACK")
                    print("[TRANSACTION COMPLETED]")
```

```
            else:
                print(f"[CRC FAILURE] {op.crc4itu(data)} and {crc}")
                #receiver.settimeout(3)
        else:
            print("Wrong receiveer")
        print("-----------------------------------------------")

def send_Ack():
    ack_frame = "11110000"
    ack_frame = op.makeFrame(rn, ack_frame)
    receiver.send(ack_frame.encode())

Recv()
```

Explanation:

1. Imports and Initial Setup

The code begins with the importation of necessary modules:

- socket: This module provides access to the BSD socket interface, which is used for network communication.
- time: The time module allows for the tracking of time-related functions, although it is not utilized in this specific code snippet.
- operations (op): This custom module is likely defined elsewhere in the project and includes functions related to frame handling and CRC computation.
- stats (st): This module presumably contains statistical constants, such as the address to which the receiver connects.

After importing the modules, the receiver initializes a sequence number (rn) to zero. This variable is essential for tracking the sequence of received frames and ensuring that they are processed in the correct order.

2. Socket Initialization

The receiver creates a TCP socket using the socket.socket function and establishes a connection to the sender using the address defined in the stats module. This socket is set to operate over IPv4 and uses TCP for reliable data transmission.

3. CRC Corruption Check Function

The isCorrupted function is defined to check whether the received data has been corrupted during transmission. It accepts two parameters: data and crc. The function uses a CRC computation function from the operations module to compare the calculated CRC of the received data with the CRC received in the frame. If they do not match, the function returns True, indicating that the data is corrupted; otherwise, it returns False.

4. Receiving Loop

The main logic of the receiver is encapsulated in the Recv function. This function continuously listens for incoming frames in an infinite loop. Upon receiving a frame, the following steps are executed:

- Listening for Frames: The receiver prints a message indicating it is ready to listen for incoming messages and waits to receive a frame. The received frame is decoded from bytes to a string format.
- Frame Processing: The function processes the received frame by extracting the sequence number, data, and CRC using a dedicated frame processing function from the operations module.
- Termination Condition: The receiver checks if the received data is a termination command ('q'). If so, it prints a closing message, closes the socket connection, and exits the function.

5. Sequence Number and CRC Validation
If the sequence number of the received frame matches the expected sequence number (rn):

- The receiver calls the isCorrupted function to check for any CRC errors. If the data is found to be uncorrupted:
    - A message is printed confirming successful CRC validation.
    - The send_Ack function is called to send an acknowledgment (ACK) back to the sender.
    - The sequence number (rn) is incremented to expect the next frame in the sequence.
    - A message is printed indicating that the ACK has been sent and the transaction is completed.

If the data is found to be corrupted, the receiver prints a failure message indicating the mismatch between the computed and received CRC values.

6. Handling Out-of-Order Frames
If the sequence number of the received frame does not match the expected sequence number (rn), the receiver prints an error message indicating that an incorrect sequence number was received. This helps in identifying issues related to frame ordering.

7. Acknowledgment Function
The send_Ack function is responsible for sending an acknowledgment frame back to the sender. It constructs an ACK frame with a predetermined format ("11110000") and then encodes it into bytes before sending it through the socket connection. The acknowledgment indicates to the sender that the specific frame has been successfully received.

8. Execution of the Receiver
Finally, the Recv function is called to start the receiver process. This initiates the listening and receiving logic described above.

Operations Program:

```
import random
import stats as st
import time

# For sake of convinience we are only injecting error in the data part

def noisychannel(frame):

    frame_list = list(frame[-st.CRC_SIZE-st.DATA_SIZE:-st.CRC_SIZE])  #
Converting frame in string for to list
    no_of_bits_changed =  random.randint(0,len(frame_list)-1)
    enum = enumerate(frame_list)

    positions = random.sample(list(enum), no_of_bits_changed)

    for _, (index, _) in enumerate(positions):
        if frame_list[index] == '0':
            frame_list[index] = '1'
        elif frame_list[index] == '1':
            frame_list[index] = '0'

    new_frame = frame[0:st.N_SIZE] +
frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE] + ''.join(frame_list) + frame[-
st.CRC_SIZE:]
    return new_frame

def delay():
    time.sleep(random.randint()% 6)

def xor(a, b):

    val = ''
    for i in range(len(b)):
        if(a[i] == b[i]):
            val += '0'
        else:
            val += '1'

    return val

def binary_division(dividend, divisor):
```

```
    rem = dividend[0:len(divisor)]
    i = len(divisor)

    while(len(rem) == len(divisor)):

        if(rem[0] != '0'):
            rem = xor(rem, divisor)[1:]
        else:
            rem = rem[1:]

        if(i == len(dividend)):
            break

        rem += dividend[i]
        i += 1

    return rem


def crc4itu(frame):

    divisor = "10011"
    remainder = binary_division(frame + '0000', divisor)

    return str(remainder)


# Wraps the data into the specified frame format
def makeFrame(n, data):                         # n = nth frame to send
    l = str(len(data))
    rem = crc4itu(data)

    # PADDING
    msg_n = str(n).zfill(st.N_SIZE)
    msg_l = l.zfill(st.LENGTH_SIZE)
    msg_data = data.zfill(st.DATA_SIZE)
    msg_rem = rem.zfill(st.CRC_SIZE)

    frame = msg_n + msg_l + msg_data + msg_rem
    return frame
```

```
def receiveFrame(frame):
    crc = int(frame[-st.CRC_SIZE:])

    n = int(frame[:st.N_SIZE]) # Extracting N

    # Extracting length
    l = int(frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE])

    data = frame[-st.CRC_SIZE-l:-st.CRC_SIZE]
    if data == "q":
        crc = ""
    else:
        # Extracting CRC code
        crc = frame[-st.CRC_SIZE:]
    return n, l, data, crc
```

Explanation:

1. Imports and Setup
The code begins by importing the necessary modules:

- **random:** This module is used to generate random numbers, which are essential for simulating errors in data transmission.
- **stats (st):** This custom module likely contains various constants, including sizes for CRC, data, and frame components.

2. Noisy Channel Simulation
The noisychannel function simulates the introduction of random errors into the data portion of a frame. Here's how it works:

- **Extracting Data:** The function extracts the data portion of the frame by slicing it based on the predefined sizes from the stats module. It converts the data portion into a list for easier manipulation.
- **Changing Bits:** A random number of bits (between 0 and the length of the data) is selected, and the positions of these bits are chosen randomly. For each selected position, the bit is flipped from '0' to '1' or from '1' to '0'. This simulates the effects of noise during transmission.
- **Reconstructing the Frame:** After modifying the bits, the function reconstructs the frame by concatenating the unchanged portions (sequence number, length, and CRC) with the modified data. The new frame is then returned.

3. Delay Simulation
The delay function introduces a random delay in processing. It uses time.sleep combined with random.randint() to create a random pause in execution, simulating network latency. Note that there seems to be an issue with random.randint() as it is incorrectly called without specifying upper and lower bounds.

4. Bitwise XOR Function
The xor function computes the bitwise XOR operation between two binary strings, a and b. It iterates through each bit of the strings:

- If the bits are the same, it appends '0' to the result.
- If the bits are different, it appends '1'. This function is fundamental for the CRC computation, allowing for the binary division of the dividend by the divisor.

5. Binary Division Function
The binary_division function performs the binary division required for CRC calculation. It accepts two binary strings: dividend and divisor. The process is as follows:

- **Initialization:** The function starts by initializing the remainder with the first segment of the dividend equal in length to the divisor.

- **Division Process:** It iterates over the dividend, performing the following steps:
  - If the first bit of the remainder is '1', it performs the XOR operation with the divisor. The remainder is updated by removing the first bit (shifting left).
  - If the first bit is '0', it simply removes the first bit of the remainder.
  - The next bit from the dividend is added to the remainder until all bits of the dividend are processed.
- **Final Remainder:** The function returns the final remainder, which will be used for CRC verification.

6. CRC Calculation

The crc4itu function calculates the CRC for a given frame using a predetermined polynomial divisor ("10011"):

- It appends four zeros to the end of the frame (to account for the CRC size) and then calls the binary_division function to compute the remainder.
- The resulting remainder is returned as the CRC value, which will be appended to the frame for integrity checking.

7. Frame Creation

The makeFrame function is responsible for constructing the data frame that will be sent over the network:

- **Length Calculation:** It calculates the length of the data.
- **CRC Calculation:** It computes the CRC for the data using the crc4itu function.
- **Padding:** The function pads the sequence number, length, data, and CRC with leading zeros to ensure they conform to specified sizes, as defined in the stats module.
- **Frame Assembly:** Finally, it concatenates all parts (sequence number, length, data, CRC) into a single frame string and returns it.

8. Frame Reception and Parsing

The receiveFrame function processes a received frame and extracts its components:

- It retrieves the CRC from the end of the frame.
- It extracts the sequence number (N) from the beginning of the frame.
- The length of the data is obtained from the specified position, followed by extracting the actual data portion of the frame.

- **Special Case Handling:** If the data indicates a termination command ('q'), the CRC variable is set to an empty string. The function returns the extracted components: sequence number, length, data, and CRC.

Stats Program:

```
import socket
import time
import random

# SOCKET VARIABLES
PORT = 8080
HOST_IP = socket.gethostbyname(socket.gethostname())
ADDR = (HOST_IP, PORT)

# DISCONNECT MESSAGE
DISCONNECT = '1111'

# SIZE OF FRAME PARTS
N_SIZE = 2
LENGTH_SIZE = 3
DATA_SIZE = 11
CRC_SIZE = 4

ack_frame = "11110000"

# FRAME FORMAT = [NN][LLL][DDDDDDDDDDDDDDD][CCCCCC]
# where D --> data, L --> length, N --> pckt no., C --> CRC
```

Explanation:

1. Imports
The code begins by importing the necessary modules:

- **socket:** This module provides access to the BSD socket interface, which is used for creating and managing network connections.
- **time:** This module provides various time-related functions, useful for adding delays or timestamps in the program.
- **random:** This module allows for generating random numbers, which can be useful for simulating noise in data transmission.

2. Socket Variables
The socket-related variables are defined to facilitate communication over a network:

- **PORT:** Set to 8080, this variable specifies the port number on which the server will listen for incoming connections. Port 8080 is commonly used for web services and can be used for testing or development purposes.
- **HOST_IP:** This variable retrieves the local host's IP address using the socket.gethostbyname(socket.gethostname()) function. This IP address will be used by clients to connect to the server.
- **ADDR:** A tuple that combines HOST_IP and PORT, representing the address the server will bind to for incoming connections.

3. Disconnect Message

- **DISCONNECT:** This variable is defined as the string '1111', which serves as a special message to indicate a request for disconnection. When this message is received, the server can close the connection gracefully.

4. Size of Frame Parts
The sizes of various components of the data frame are defined as constants:

- **N_SIZE:** This constant is set to 2, representing the size (in bits) for the sequence number of the packet. It allows for up to 4 unique packet numbers (00 to 11).
- **LENGTH_SIZE:** This constant is set to 3, indicating the size (in bits) for the length of the data. This allows for a maximum length of 7 bits, or 2^3 - 1, meaning the maximum data size can be 11 bits (as defined in the next constant).
- **DATA_SIZE:** This constant is set to 11, representing the size (in bits) allocated for the actual data part of the frame. This allows for 2^11 possible values in the data field.
- **CRC_SIZE:** This constant is set to 4, indicating the size (in bits) for the cyclic redundancy check (CRC) portion of the frame. The CRC is used for error detection, ensuring the integrity of the transmitted data.

5. Acknowledgment Frame

- **ack_frame:** This variable is initialized to "11110000", which serves as a predefined acknowledgment frame sent back to the sender to confirm receipt of a data packet. This specific frame format will depend on the implementation and protocol being followed.

6. Frame Format

The code includes a comment that describes the overall format of the data frame that will be sent over the socket:

- **Frame Format:** [NN][LLL][DDDDDDDDDD][CCCC]
  - **NN:** The packet number (sequence number), represented by N_SIZE bits.
  - **LLL:** The length of the data, represented by LENGTH_SIZE bits.
  - **DDDDDDDDDD:** The actual data being sent, represented by DATA_SIZE bits.
  - **CCCC:** The CRC for error checking, represented by CRC_SIZE bits

TEST CASES

Sender Output:

```
[LISTENING] Server is listening on 192.168.29.31
[CONNECTED] Connected to Process Id: ('192.168.29.31', 63830)
[INPUT] Enter data to send: 1010
[ENCODING] Encoded frame: 00004000000010101101
[NOISY] Frame after noise: 00004010101001111101
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00004111111000011101
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00004101100111111101
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00004111111100011101
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00004000000010101101
[Reciving ACK]......
[ACK RECV] ACK 0 successfully received.
[TRANSACTION COMPLETED]
```

Receiver Output:

```
[LISTENING] Receiver is listening....
[RECV] Received message: 0111
[CRC FAILURE] 1001 and 1101
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 0001
[CRC FAILURE] 0011 and 1101
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 1111
[CRC FAILURE] 0010 and 1101
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 0001
[CRC FAILURE] 0011 and 1101
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 1010
[CRC SUCCESS] 1101 and 1101
[ACK SENT] Sent ACK
[TRANSACTION COMPLETED]
```

Implementation:

Go-Back-N ARQ:

Sender Program:

```
import socket, threading, time, operations as op, stats as st

sf = 0
sn = 0
sw = 3

text = []
length = 0
flag2 = False   # used to terminate the recv_Ack thread

sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sender.bind(st.ADDR)

data_queue = [None]*sw

flag = False

def send(conn, f):

    global sn, sf, sw, data_queue, flag, text

    while True:
        if (sn - sf) < sw:

            try:
                data = text.pop(0)
            except:
                print("[FINISHED] All output read.")
                flag = True
                return

            frame = op.noisychannel(op.makeFrame(sn, data))
            print(f"[SENDING] Sending frame: {frame}")
            conn.send(frame.encode())

            data_queue[sn % sw] = data

            sn += 1
```

```
def recv_Ack(conn):

    global sn, sf, sw, data_queue, text, length

    # Receiving ACK Frame
    while True:
        global sf, sn, sw, data_queue, flag2
        try:
            conn.settimeout(0.5)
            recv_frame = conn.recv(20).decode()

            # Extracting frame details
            ackNo, _, data,_ = op.receiveFrame(recv_frame)

        except:
            resend_thread = threading.Thread(target=timeout_steps, args=(conn,))
            resend_thread.start()
            resend_thread.join()
            if flag2 == True:
                print("[FINISHED] All ACK received.")
                return
            continue

        if ackNo >= sf and ackNo <= sn and data == '11110000': # if valid ACK
            while(sf <= ackNo):
                print(f"[ACK RECV] ACK {ackNo} successfully received.")
                data_queue[sf % sw] = ""
                sf += 1

        if ackNo == (length - 1):  # All ACK received
            flag2 = True

def timeout_steps(conn):
    global sf, sn, sw, data_queue
    temp = sf
    while(temp < sn):
        data = data_queue[temp % sw]
        frame = op.makeFrame(temp, data)
        print(f"[RE SENDING] Resending frame: {frame}")
        conn.send(frame.encode())
        temp += 1
```

```
def start():
    # Listening
    global text, length

    sender.listen()
    print(f"[LISTENING] Server is listening on {st.HOST_IP}")

    # Accepting receiver
    conn, addr = sender.accept()
    print(f"[CONNECTED] Connected to Process Id: {addr}")

    f = open("data.txt", "r")

    text = f.readlines()
    for i in range(len(text)):
        text[i] = text[i].strip()

    length = len(text)

    sender_thread = threading.Thread(target=send, args=(conn, f))
    receiver_thread = threading.Thread(target=recv_Ack, args=(conn,))

    sender_thread.start()
    receiver_thread.start()
    sender_thread.join()
    receiver_thread.join()

    print("[CLOSING] Closing sender...")
    conn.close()

    f.close()


start()
```

Explanation:

1. Imports

The code begins by importing necessary modules:

- **socket:** For creating network connections.
- **threading:** To enable concurrent execution of functions, allowing the sender to send data and receive acknowledgments simultaneously.
- **time:** Provides time-related functions, though it's not heavily utilized in the provided snippet.
- **operations (op):** A custom module likely containing functions for data frame operations like making frames and introducing noise.
- **stats (st):** A custom module likely containing constant values or configurations, such as socket addresses.

2. Global Variables

Several global variables are initialized:

- **sf (send frame):** Tracks the sequence number of the last acknowledged frame.
- **sn (send next):** Represents the sequence number of the next frame to send.
- **sw (sliding window size):** Defines the maximum number of frames that can be sent before requiring an acknowledgment, set to 3.
- **text:** An empty list that will hold the data read from a file.
- **length:** An integer representing the total number of data items to send.
- **flag2:** A boolean used to signal the termination of the acknowledgment receiving thread.

3. Socket Setup

The sender socket is created and bound to the address defined in the stats module:
python
Copy code

```
sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sender.bind(st.ADDR)
```

4. Data Queue

- **data_queue:** An array initialized to hold None values, with a size equal to the sliding window (sw). It stores data that has been sent but not yet acknowledged.

5. Send Function
The send function is responsible for sending frames of data:

- It enters a loop where it checks if the number of frames sent (sn - sf) is less than the window size (sw).
- If there is data to send, it pops the next data item from the text list, creates a frame using the makeFrame function from the operations module, and applies noise to simulate transmission errors using the noisychannel function.
- The frame is then sent over the established connection.
- The corresponding data is stored in the data_queue, and the send sequence number (sn) is incremented.

6. Receive Acknowledgment Function
The recv_Ack function listens for acknowledgment frames:

- It runs in a loop and attempts to receive acknowledgment frames from the connection.
- If an acknowledgment is received, it extracts the acknowledgment number and checks if it falls within valid bounds.
- For valid acknowledgments, it updates the sf variable and clears the corresponding entry in the data_queue.
- If all acknowledgments are received, flag2 is set to True.

7. Timeout Steps Function
The timeout_steps function is executed in a separate thread to handle retransmission:

- It checks for frames that need to be resent if an acknowledgment is not received in a timely manner.
- It iterates through the data_queue and resends frames for any that are still outstanding.

8. Start Function
The start function sets up the server:

- It listens for incoming connections and accepts a connection from a client (receiver).
- It reads data from a file (data.txt) and stores it in the text list.
- A sender_thread is created to run the send function, and a receiver_thread is created to run the recv_Ack function.
- Both threads are started, allowing them to run concurrently, and the main thread waits for both to complete using join.

9. Closing Operations
Once both threads finish execution, the connection is closed, and the file is also closed

Receiver Program:

```
from ast import arg
import socket, time, threading, operations as op, stats as st

rn = 0

receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)
# receiver.settimeout(3)

def isCorrupted(data, crc):
    if op.crc4itu(data) != crc:
        return True
    else:
        return False

def Recv():
    global rn

    while True:

        # print("[LISTENING] Receiver is listening....")
        try:
            frame = receiver.recv(20).decode()
            recv_n, _, data, crc = op.receiveFrame(frame)
        except:
            # When everything is done, ValueError will be raised due to receiver
receiving empty string
            # so we use it to terminate this thread
            return
        print(f"[RECV] Received message: {recv_n}, {data}")

        # Checking if disconnect statement is there

        if isCorrupted(data, crc) == False and recv_n == rn : # No crc error
            print(f"[CRC SUCCESS] {op.crc4itu(data)} and {crc}")
            send_Ack()
            rn += 1
            print(f"[ACK SENT] Sent ACK")
        else:
            print(f"[CRC FAILURE] {op.crc4itu(data)} and {crc}")

        #print("-----------------------------------------------")
```

```
def send_Ack():
    ack_frame = "11110000"
    ack_frame = op.makeFrame(rn, ack_frame)
    receiver.send(ack_frame.encode())

receiver_thread = threading.Thread(target=Recv)

receiver_thread.start()
receiver_thread.join()

print("[CLOSING] Closing receiver....")
receiver.close()
```

Explanation:

1. Imports and Initializations

The code begins with the necessary imports:

- **socket:** For creating and managing network connections.
- **time:** Although imported, it is not utilized in this snippet.
- **threading:** To allow concurrent execution of the receiver functionality.
- **operations (op):** A custom module that likely contains functions related to frame operations and CRC calculations.
- **stats (st):** A custom module that likely contains configurations, such as socket addresses.

A global variable rn (receive number) is initialized to 0, which keeps track of the sequence number of the last received and acknowledged frame.

2. Socket Setup

A socket object is created and connected to the address defined in the stats module:
python
Copy code
receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)

3. Corruption Check Function

The isCorrupted function checks whether the received data is corrupted:

- It takes the data and its corresponding CRC as arguments and compares the calculated CRC (using op.crc4itu(data)) with the received CRC.
- Returns True if they do not match, indicating that the data is corrupted, and False otherwise.

4. Receive Function

The Recv function runs in a loop to listen for incoming frames:

- It tries to receive data from the socket in a loop.
- Upon receiving a frame, it decodes the frame and extracts the sequence number (recv_n), data, and CRC.
- It then prints the received message for debugging purposes.
- If the data is not corrupted and the received sequence number matches the expected sequence number (rn), it indicates successful reception. The function sends an ACK back to the sender, increments the expected sequence number (rn), and prints a confirmation message.
- If there is a corruption (the CRC does not match), it prints a failure message.

5. Sending ACK Function

The send_Ack function constructs and sends an acknowledgment frame:

- An ACK frame is created with a predefined value (in this case, "11110000").
- The function uses op.makeFrame(rn, ack_frame) to create a proper frame format for sending and then sends it back to the sender.

6. Threading and Execution

A thread is created to run the Recv function:

python

Copy code

```
receiver_thread = threading.Thread(target=Recv)
receiver_thread.start()
receiver_thread.join()
```

This allows the receiver to operate concurrently and listen for incoming frames while potentially freeing up the main thread for other tasks.

7. Closing Operations

Once the receiving thread completes its execution (indicated by termination of the while loop), the socket connection is closed, and a message indicating that the receiver is closing is printed.

Operations Program:

```
import random
import stats as st
import time

# For sake of convinience we are only injecting error in the data part

def noisychannel(frame):

    frame_list = list(frame[-st.CRC_SIZE-st.DATA_SIZE:-st.CRC_SIZE])  #
Converting frame in string for to list
    no_of_bits_changed =  random.randint(0,len(frame_list)-1)
    enum = enumerate(frame_list)

    positions = random.sample(list(enum), no_of_bits_changed)

    for _, (index, _) in enumerate(positions):
        if frame_list[index] == '0':
            frame_list[index] = '1'
        elif frame_list[index] == '1':
            frame_list[index] = '0'

    new_frame = frame[0:st.N_SIZE] +
frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE] + ''.join(frame_list) + frame[-
st.CRC_SIZE:]
    return new_frame

def delay():
    time.sleep(random.randint()% 6)

def xor(a, b):

    val = ''
    for i in range(len(b)):
        if(a[i] == b[i]):
            val += '0'
        else:
            val += '1'

    return val

def binary_division(dividend, divisor):
```

```
    rem = dividend[0:len(divisor)]
    i = len(divisor)

    while(len(rem) == len(divisor)):

        if(rem[0] != '0'):
            rem = xor(rem, divisor)[1:]
        else:
            rem = rem[1:]

        if(i == len(dividend)):
            break

        rem += dividend[i]
        i += 1

    return rem


def crc4itu(frame):

    divisor = "10011"
    remainder = binary_division(frame + '0000', divisor)

    return str(remainder)


# Wraps the data into the specified frame format
def makeFrame(n, data):                         # n = nth frame to send
    l = str(len(data))
    rem = crc4itu(data)

    # PADDING
    msg_n = str(n).zfill(st.N_SIZE)
    msg_l = l.zfill(st.LENGTH_SIZE)
    msg_data = data.zfill(st.DATA_SIZE)
    msg_rem = rem.zfill(st.CRC_SIZE)

    frame = msg_n + msg_l + msg_data + msg_rem
    return frame

def receiveFrame(frame):
    crc = int(frame[-st.CRC_SIZE:])
```

```
n = int(frame[:st.N_SIZE]) # Extracting N

# Extracting length
l = int(frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE])

data = frame[-st.CRC_SIZE-l:-st.CRC_SIZE]
if data == "q":
    crc = ""
else:
    # Extracting CRC code
    crc = frame[-st.CRC_SIZE:]
return n, l, data, crc
```

Explanation:

1. Imports and Module Initialization
The code imports necessary modules:

- **random:** Used for simulating noise in the channel by randomly altering bits in the transmitted data.
- **stats (st):** A custom module that presumably defines constants such as sizes for different parts of the frame (like CRC size, data size, etc.).
- **time:** Used to introduce delays in the simulation.

2. Function: noisychannel(frame)
This function simulates a noisy communication channel by randomly flipping bits in the data portion of a frame.

- **Extracting Data:** It converts the relevant part of the frame into a list of bits for manipulation.
- **Random Bit Flipping:** It selects a random number of bits to change (between 0 and the length of the data portion) and flips them from 0 to 1 or from 1 to 0.
- **Reconstructing the Frame:** After flipping the bits, it reconstructs the frame by concatenating the unchanged parts and the modified data.

3. Function: delay()
This function introduces a random delay (up to 6 seconds) but currently contains a syntax error (random.randint()% 6) which should be corrected. It should be random.randint(0, 5) to specify both the minimum and maximum bounds.

4. Function: xor(a, b)
This function performs a bitwise XOR operation on two binary strings.

- It iterates through the characters of both strings, appending 0 to the result if the bits are the same and 1 if they differ.

5. Function: binary_division(dividend, divisor)
This function performs binary division (similar to polynomial division in coding theory) to calculate the remainder when dividing a binary string.

- It initializes the remainder with the first bits of the dividend.
- It iterates over the bits of the dividend, updating the remainder by XORing it with the divisor when appropriate.

6. Function: crc4itu(frame)
This function calculates the CRC (Cyclic Redundancy Check) for a given frame using a specified polynomial (in this case, 10011).

- It appends zeros to the frame and calls binary_division to get the remainder, which serves as the CRC.

7. Function: makeFrame(n, data)

This function constructs a data frame according to the defined protocol format.

- It calculates the length of the data and CRC, pads the necessary parts, and concatenates them to create the complete frame.
- The result is a string representing the full frame ready to be sent over the network.

8. Function: receiveFrame(frame)

This function extracts the components of a received frame:

- It retrieves the sequence number, length, data, and CRC from the frame.
- If the received data is "q" (possibly a disconnect signal), it resets the CRC.

Stats Program:

```
import socket
import time
import random

# SOCKET VARIABLES
PORT = 8081
HOST_IP = socket.gethostbyname(socket.gethostname())
ADDR = (HOST_IP, PORT)

# DISCONNECT MESSAGE
DISCONNECT = '1111'

# SIZE OF FRAME PARTS
N_SIZE = 2
LENGTH_SIZE = 3
DATA_SIZE = 11
CRC_SIZE = 4

ack_frame = "11110000"

# FRAME FORMAT = [NN][LLL][DDDDDDDDDDDDDDD][CCCCCC]
# where D --> data, L --> length, N --> pckt no., C --> CRC
```

Explanation:

1. Socket Variables
These variables define the communication parameters for the socket connection:

- **PORT**: The port number on which the server will listen for incoming connections. In this case, it's set to 8081.
- **HOST_IP**: Retrieves the local host IP address using the gethostbyname method from the socket library, which resolves the hostname of the machine to its corresponding IP address.
- **ADDR**: A tuple that combines HOST_IP and PORT to define the address that the server will bind to.

2. Disconnect Message

- **DISCONNECT**: A predefined message ('1111') that likely indicates to the receiver that the sender intends to close the connection. It can be used in the protocol to signal termination.

3. Sizes of Frame Parts
These constants define the size of various components of the data frame:

- **N_SIZE**: The number of bits allocated for the packet number. Here, it's set to 2 bits.
- **LENGTH_SIZE**: The number of bits allocated for the length of the data. It's set to 3 bits.
- **DATA_SIZE**: The size of the data field within the frame, set to 11 bits.
- **CRC_SIZE**: The size of the CRC (Cyclic Redundancy Check) portion of the frame, set to 4 bits.

4. Acknowledgment Frame

- **ack_frame**: This is a predefined string ("11110000") that likely represents a generic acknowledgment frame to be sent back to the sender after successfully receiving data. It indicates that the receiver is acknowledging the receipt of data.

5. Frame Format

- **Frame Structure**: The comment describes the structure of the frame that will be used in communication. Each frame consists of:
  - **NN**: The packet number (2 bits).
  - **LLL**: The length of the data (3 bits).
  - **DDDDDDDDDDD**: The data itself (11 bits).
  - **CCCCCC**: The CRC (4 bits).

The overall format is designed to encapsulate all necessary information for reliable communication, including error detection through CRC

TEST CASES

Sender Output:

```
[LISTENING] Server is listening on 192.168.29.31
[CONNECTED] Connected to Process Id: ('192.168.29.31', 64082)
[SENDING] Sending frame: 00006111100011111111
[SENDING] Sending frame: 01006111010011111100
[SENDING] Sending frame: 02006100010101111001
[RE SENDING] Resending frame: 00006000001100001111
[RE SENDING] Resending frame: 01006000001100011100
[RE SENDING] Resending frame: 02006000001100101001
[ACK RECV] ACK 0 successfully received.
[SENDING] Sending frame: 03006011110011001111
[ACK RECV] ACK 1 successfully received.
[ACK RECV] ACK 2 successfully received.
[SENDING] Sending frame: 04006000001101000011
[SENDING] Sending frame: 05006110110010100101
[RE SENDING] Resending frame: 03006000001000111111
[RE SENDING] Resending frame: 04006000001101000011
[RE SENDING] Resending frame: 05006000001001010101
[ACK RECV] ACK 3 successfully received.
[SENDING] Sending frame: 06006111110000001110
[ACK RECV] ACK 4 successfully received.
[ACK RECV] ACK 5 successfully received.
[SENDING] Sending frame: 07006011110011110011
[SENDING] Sending frame: 08006000001111111101
[RE SENDING] Resending frame: 06006000001111101110
[RE SENDING] Resending frame: 07006000001001110011
[RE SENDING] Resending frame: 08006000001111111101
[ACK RECV] ACK 6 successfully received.
[ACK RECV] ACK 7 successfully received.
[FINISHED] All output read.
[ACK RECV] ACK 8 successfully received.
[FINISHED] All ACK received.
[CLOSING] Closing sender...
```

Receiver Output:

```
[RECV] Received message: 0, 001111
[CRC FAILURE] 0010 and 1111
[RECV] Received message: 1, 001111
[CRC FAILURE] 0010 and 1100
[RECV] Received message: 2, 010111
[CRC FAILURE] 1100 and 1001
[RECV] Received message: 0, 110000
[CRC SUCCESS] 1111 and 1111
[ACK SENT] Sent ACK
[RECV] Received message: 1, 110001
[CRC SUCCESS] 1100 and 1100
[ACK SENT] Sent ACK
[RECV] Received message: 2, 110010
[CRC SUCCESS] 1001 and 1001
[ACK SENT] Sent ACK
[RECV] Received message: 3, 001100
[CRC FAILURE] 0111 and 1111
[RECV] Received message: 4, 110100
[CRC FAILURE] 0011 and 0011
[RECV] Received message: 5, 001010
[CRC FAILURE] 1101 and 0101
[RECV] Received message: 3, 100011
[CRC SUCCESS] 1111 and 1111
[ACK SENT] Sent ACK
[RECV] Received message: 4, 110100
[CRC SUCCESS] 0011 and 0011
[ACK SENT] Sent ACK
[RECV] Received message: 5, 100101
[CRC SUCCESS] 0101 and 0101
[ACK SENT] Sent ACK
[RECV] Received message: 6, 000000
[CRC FAILURE] 0000 and 1110
[RECV] Received message: 7, 001111
[CRC FAILURE] 0010 and 0011
[RECV] Received message: 8, 111111
[CRC FAILURE] 1101 and 1101
[RECV] Received message: 6, 111110
[CRC SUCCESS] 1110 and 1110
[ACK SENT] Sent ACK
[RECV] Received message: 7, 100111
[CRC SUCCESS] 0011 and 0011
[ACK SENT] Sent ACK
[RECV] Received message: 8, 111111
[CRC SUCCESS] 1101 and 1101
[ACK SENT] Sent ACK
[CLOSING] Closing receiver....
```

Implementation:

Selective Repeat ARQ:

Sender Program:

```
import socket, threading, time, operations as op, stats as st

sf = 0
sn = 0
sw = 4          # 0 based indexing

text = []
length = 0
flag2 = False   # used to terminate the recv_Ack thread

sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sender.bind(st.ADDR)

data_queue = [None]*sw

flag = False

def send(conn, f):

    global sn, sf, sw, data_queue, flag, text

    while True:
        if (sn - sf) < sw:

            try:
                data = text.pop(0)
            except:
                print("[FINISHED] All output read.")
                flag = True
                return

            frame = op.noisychannel(op.makeFrame(sn, data))
            print(f"[SENDING] Sending frame: no. {sn}, {frame}")
            conn.send(frame.encode())

            data_queue[sn % sw] = data
```

```
            sn += 1

def recv_Ack(conn):

    global sn, sf, sw, data_queue, text, length

    # Receiving ACK Frame
    while True:
        global sf, sn, sw, data_queue, flag2

        recv_frame = conn.recv(20).decode()

        ackNo, _, data,_ = op.receiveFrame(recv_frame)

        if data == '00001111':
            if ackNo >= sf and ackNo <= sn: # if NAK
                print(f"[NAK RECV] NAK {ackNo} successfully received.")
                resend_thread = threading.Thread(target=timeout_steps, args=(conn,
ackNo))
                resend_thread.start()
                resend_thread.join()

        if data == '11110000':
            if ackNo >= sf and ackNo <= sn: # if ACK
                print(f"[ACK RECV] ACK {ackNo} successfully received.")
            while(sf <= ackNo):
                data_queue[sf % sw] = ""
                sf += 1
            if ackNo == 7:
                return

def timeout_steps(conn, ackNo):
    global sf, sn, sw, data_queue
    data = data_queue[ackNo % sw]
    frame = op.makeFrame(ackNo, data)
    print(f"[RE SENDING] Resending frame: {ackNo}, {frame}")
    conn.send(frame.encode())

def start():
    # Listening
    global text, length

    sender.listen()
    print(f"[LISTENING] Server is listening on {st.HOST_IP}")
```

```
    # Accepting receiver
    conn, addr = sender.accept()
    print(f"[CONNECTED] Connected to Process Id: {addr}")

    f = open("data.txt", "r")

    text = f.readlines()
    for i in range(len(text)):
        text[i] = text[i].strip()

    length = len(text)

    sender_thread = threading.Thread(target=send, args=(conn, f))
    receiver_thread = threading.Thread(target=recv_Ack, args=(conn,))

    sender_thread.start()
    receiver_thread.start()
    sender_thread.join()
    receiver_thread.join()

    print("[CLOSING] Closing sender...")
    conn.close()

    f.close()


start()
```

Explanation:

1. Socket Setup

- **Port**: 8081
- **Host IP**: Automatically fetched via socket.gethostbyname(socket.gethostname()).
- **Address**: Tuple defined as (HOST_IP, PORT).

2. Frame Structure

- **Frame Format**: [NN][LLL][DDDDDDDDDDD][CCCC]
  - **NN**: Packet Number (N_SIZE = 2 bits)
  - **LLL**: Length of Data (LENGTH_SIZE = 3 bits)
  - **DDDDDDDDDDD**: Data (DATA_SIZE = 11 bits)
  - **CCCC**: CRC (CRC_SIZE = 4 bits)
- **Disconnect Message**: '1111'
- **Window Size**: sw = 4 (Allows 4 unacknowledged frames)

3. Functions

a. send(conn, f)

- Continuously sends frames until the number of unacknowledged frames is less than sw.
- Constructs frames using op.makeFrame(sn, data).
- Introduces simulated transmission errors via op.noisychannel().

b. recv_Ack(conn)

- Listens for acknowledgment frames from the receiver.
- Checks for ACK and NAK:
  - **ACK**: Updates the sender's frame count and clears the acknowledged frames from data_queue.
  - **NAK**: Triggers retransmission of the specified frame using timeout_steps.

c. timeout_steps(conn, ackNo)

- Resends the frame corresponding to the negative acknowledgment (NAK).
- Constructs the frame and sends it to the receiver.

d. start()

- Initializes the server and listens for incoming connections.
- Reads data from "data.txt" and stores it in the text list.
- Spawns threads for sending data and receiving acknowledgments.

Receiver Program:

```
import socket, time, threading, operations as op, stats as st

rf = 0
rn = 0
rw = 4

receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)

def isCorrupted(data, crc):
    if op.crc4itu(data) != crc:
        return True
    else:
        return False

def Recv():
    global rn, nak_sent, bool_slot, data_slot, rw

    while True:

        # print("[LISTENING] Receiver is listening....")
        try:
            frame = receiver.recv(20).decode()
            recv_n, _, data, crc = op.receiveFrame(frame)
        except:
            # When everything is done, ValueError will be raised due to receiver
receiving empty string
            # so we use it to terminate this thread
            return
        print(f"[RECV] Received message: {recv_n}, {data}")

        if isCorrupted(data, crc) == True:
            print(f"[CRC FAILURE] {op.crc4itu(data)} and {crc}")
            send_NAK(recv_n)
            print(f"[NAK SENT] Sent NAK {recv_n}")
        else:
            print(f"[CRC SUCESS] {op.crc4itu(data)} and {crc}")
            send_Ack(recv_n)
            print(f"[ACK SENT] Sent ACK {recv_n}")

        print("-----------------------------------------------")
```

```
def send_Ack(seq):
    ack_frame = "11110000"
    ack_frame = op.makeFrame(seq, ack_frame)
    receiver.send(ack_frame.encode())

def send_NAK(recv_n):
    nak_frame = "00001111"
    nak_frame = op.makeFrame(recv_n, nak_frame)
    receiver.send(nak_frame.encode())

receiver_thread = threading.Thread(target=Recv)

receiver_thread.start()
receiver_thread.join()

print("[CLOSING] Closing receiver....")
receiver.close()
```

Explanation:

1. Socket Setup

- **Socket**: Created using socket.socket(socket.AF_INET, socket.SOCK_STREAM).
- **Connection**: Established with the server using receiver.connect(st.ADDR).

2. Frame Structure

- **ACK Frame**: '11110000'
- **NAK Frame**: '00001111'
- **Frame Format**: Utilizes the op.makeFrame function to format frames sent to the sender.

3. Functions

a. isCorrupted(data, crc)

- **Purpose**: Checks if the received data has been corrupted by comparing the computed CRC with the received CRC.
- **Return Value**:
  - True if the data is corrupted.
  - False if the data is valid.

b. Recv()

- Continuously listens for incoming frames.
- Processes each received frame:
  - Calls op.receiveFrame(frame) to decode the frame.
  - Verifies the data integrity using isCorrupted().
  - Sends ACK if the frame is valid, or NAK if corrupted.
- **Loop Termination**: Catches exceptions (e.g., empty frames) to terminate the thread gracefully.

c. send_Ack(seq)

- Constructs and sends an acknowledgment frame for the received sequence number using op.makeFrame(seq, ack_frame).

d. send_NAK(recv_n)

- Constructs and sends a negative acknowledgment frame for the received sequence number using op.makeFrame(recv_n, nak_frame).

4. Multithreading

- The receiver runs on a separate thread using threading.Thread(target=Recv) to handle incoming frames without blocking the main execution flow

Operations Program:

```
import random
import stats as st
import time

# For sake of convinience we are only injecting error in the data part

def noisychannel(frame):

    frame_list = list(frame[-st.CRC_SIZE-st.DATA_SIZE:-st.CRC_SIZE])  #
Converting frame in string for to list
    no_of_bits_changed =  random.randint(0,len(frame_list)-1)
    enum = enumerate(frame_list)

    positions = random.sample(list(enum), no_of_bits_changed)

    for _, (index, _) in enumerate(positions):
        if frame_list[index] == '0':
            frame_list[index] = '1'
        elif frame_list[index] == '1':
            frame_list[index] = '0'

    new_frame = frame[0:st.N_SIZE] +
frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE] + ''.join(frame_list) + frame[-
st.CRC_SIZE:]
    return new_frame

def delay():
    time.sleep(random.randint()% 6)

def xor(a, b):

    val = ''
    for i in range(len(b)):
        if(a[i] == b[i]):
            val += '0'
        else:
            val += '1'

    return val

def binary_division(dividend, divisor):
```

```
    rem = dividend[0:len(divisor)]
    i = len(divisor)

    while(len(rem) == len(divisor)):

        if(rem[0] != '0'):
            rem = xor(rem, divisor)[1:]
        else:
            rem = rem[1:]

        if(i == len(dividend)):
            break

        rem += dividend[i]
        i += 1

    return rem


def crc4itu(frame):

    divisor = "10011"
    remainder = binary_division(frame + '0000', divisor)

    return str(remainder)


# Wraps the data into the specified frame format
def makeFrame(n, data):                         # n = nth frame to send
    l = str(len(data))
    rem = crc4itu(data)

    # PADDING
    msg_n = str(n).zfill(st.N_SIZE)
    msg_l = l.zfill(st.LENGTH_SIZE)
    msg_data = data.zfill(st.DATA_SIZE)
    msg_rem = rem.zfill(st.CRC_SIZE)

    frame = msg_n + msg_l + msg_data + msg_rem
    return frame

def receiveFrame(frame):
    crc = int(frame[-st.CRC_SIZE:])
```

```
n = int(frame[:st.N_SIZE]) # Extracting N

# Extracting length
l = int(frame[st.N_SIZE:st.N_SIZE+st.LENGTH_SIZE])

data = frame[-st.CRC_SIZE-l:-st.CRC_SIZE]
if data == "q":
    crc = ""
else:
    # Extracting CRC code
    crc = frame[-st.CRC_SIZE:]
return n, l, data, crc
```

Explanation:

1. Functions

a. noisychannel(frame)

- **Purpose**: Simulates a noisy channel by randomly flipping bits in the data part of a frame.
- **Parameters**:
  - frame: The original frame string that contains the data and CRC.
- **Process**:
  - Converts the data part of the frame to a list of bits.
  - Randomly selects a number of bits to change (flip from 0 to 1 or vice versa).
  - Constructs and returns a new frame with altered data.
- **Returns**: A new frame string with injected errors.

b. delay()

- **Purpose**: Introduces a random delay in the processing (not directly utilized in the current implementation).
- **Functionality**:
  - Utilizes time.sleep() to pause execution for a random duration between 0 and 6 seconds.

c. xor(a, b)

- **Purpose**: Performs bitwise XOR operation between two binary strings.
- **Parameters**:
  - a, b: Binary strings of equal length.
- **Returns**: A new binary string resulting from the XOR operation.

d. binary_division(dividend, divisor)

- **Purpose**: Implements binary division for CRC calculation.
- **Parameters**:
  - dividend: The binary string to be divided.
  - divisor: The binary string used for division (the CRC polynomial).
- **Returns**: The remainder after the division.

e. crc4itu(frame)

- **Purpose**: Calculates the CRC for a given frame.
- **Parameters**:
  - frame: The input frame string (data part).
- **Process**:
  - Appends four zeros to the frame for CRC calculation.
  - Calls binary_division() to compute the remainder using the specified divisor ("10011").
- **Returns**: The computed CRC as a binary string.

f. makeFrame(n, data)

- **Purpose**: Constructs a frame based on the specified format including sequence number, length, data, and CRC.
- **Parameters**:
  - n: The sequence number of the frame.
  - data: The data to be sent.
- **Process**:
  - Calculates the length of the data.
  - Computes the CRC for the data using crc4itu().
  - Pads the sequence number, length, data, and CRC to their respective sizes.
- **Returns**: The complete formatted frame as a string.

g. receiveFrame(frame)

- **Purpose**: Extracts components from a received frame.
- **Parameters**:
  - frame: The received frame string.
- **Process**:
  - Extracts the sequence number, length, data, and CRC from the frame.
- **Returns**: A tuple containing the sequence number, length, data, and CRC.

3. Error Handling

- The noisychannel() function specifically targets the data part of the frame for error injection, simulating a real-world scenario where data corruption may occur during transmission.

Stats Program:

```
import socket
import time
import random

# SOCKET VARIABLES
PORT = 8080
HOST_IP = socket.gethostbyname(socket.gethostname())
ADDR = (HOST_IP, PORT)

# DISCONNECT MESSAGE
DISCONNECT = '1111'

# SIZE OF FRAME PARTS
N_SIZE = 2
LENGTH_SIZE = 3
DATA_SIZE = 11
CRC_SIZE = 4

ack_frame = "11110000"

# FRAME FORMAT = [NN][LLL][DDDDDDDDDDDDDDD][CCCCCC]
# where D --> data, L --> length, N --> pckt no., C --> CRC
```

Explanation:

1. Socket Variables

a. Port and Host Configuration

- **PORT**:
  - Set to 8080, a commonly used port for socket communication.
- **HOST_IP**:
  - Dynamically retrieves the local host's IP address using socket.gethostbyname(socket.gethostname()).
- **ADDR**:
  - A tuple combining HOST_IP and PORT, representing the address for socket communication.

b. Disconnect Message

- **DISCONNECT**:
  - A predefined string '1111' that signals disconnection from the socket communication.

2. Frame Structure

a. Size Definitions

- **N_SIZE**:
  - Size of the sequence number, set to 2 bits.
- **LENGTH_SIZE**:
  - Size of the length field, set to 3 bits.
- **DATA_SIZE**:
  - Size of the data field, set to 11 bits.
- **CRC_SIZE**:
  - Size of the CRC field, set to 4 bits.

b. Frame Format

- **Format Description**:
  - The frame follows a structured format:
    [NN][LLL][DDDDDDDDDD][CCCC]
    Where:
    - **NN**: Sequence number (2 bits)
    - **LLL**: Length of the data (3 bits)
    - **DDDDDDDDDD**: Actual data (11 bits)
    - **CCCC**: CRC for error checking (4 bits)

c. Acknowledgment Frame

- **ack_frame**:
  - A predefined acknowledgment frame set to "11110000"

TEST CASES Sender Output:

```
[LISTENING] Server is listening on 192.168.29.31
[CONNECTED] Connected to Process Id: ('192.168.29.31', 64254)
[SENDING] Sending frame: no. 0, 00006111110110000101
[SENDING] Sending frame: no. 1, 01006100110010000110
[SENDING] Sending frame: no. 2, 02006001001101000101
[SENDING] Sending frame: no. 3, 03006100001001110101
[NAK RECV] NAK 0 successfully received.
[RE SENDING] Resending frame: 0, 00006000001101100101
[NAK RECV] NAK 1 successfully received.
[RE SENDING] Resending frame: 1, 01006000001101110110
[NAK RECV] NAK 2 successfully received.
[RE SENDING] Resending frame: 2, 02006000001101100101
[NAK RECV] NAK 3 successfully received.
[RE SENDING] Resending frame: 3, 03006000001001010101
[ACK RECV] ACK 0 successfully received.
[ACK RECV] ACK 1 successfully received.
[ACK RECV] ACK 2 successfully received.
[ACK RECV] ACK 3 successfully received.
[SENDING] Sending frame: no. 4, 04006000001100011010
[SENDING] Sending frame: no. 5, 05006011000001110000
[SENDING] Sending frame: no. 6, 06006000000110010111
[SENDING] Sending frame: no. 7, 07006011100111110000
[NAK RECV] NAK 4 successfully received.
[RE SENDING] Resending frame: 4, 04006000001100111010
[NAK RECV] NAK 5 successfully received.
[RE SENDING] Resending frame: 5, 05006000001001100000
[NAK RECV] NAK 6 successfully received.
[RE SENDING] Resending frame: 6, 06006000001110010111
[NAK RECV] NAK 7 successfully received.
[RE SENDING] Resending frame: 7, 07006000001001100000
[ACK RECV] ACK 4 successfully received.
[ACK RECV] ACK 5 successfully received.
[FINISHED] All output read.
[ACK RECV] ACK 6 successfully received.
[ACK RECV] ACK 7 successfully received.
[CLOSING] Closing sender...
```

Receiver Output:

```
[RECV] Received message: 0, 011000
[CRC FAILURE] 1110 and 0101
[NAK SENT] Sent NAK 0
------------------------------------------------
[RECV] Received message: 1, 001000
[CRC FAILURE] 1011 and 0110
[NAK SENT] Sent NAK 1
------------------------------------------------
[RECV] Received message: 2, 110100
[CRC FAILURE] 0011 and 0101
[NAK SENT] Sent NAK 2
------------------------------------------------
[RECV] Received message: 3, 100111
[CRC FAILURE] 0011 and 0101
[NAK SENT] Sent NAK 3
------------------------------------------------
[RECV] Received message: 0, 110110
[CRC SUCESS] 0101 and 0101
[ACK SENT] Sent ACK 0
------------------------------------------------
[RECV] Received message: 1, 110111
[CRC SUCESS] 0110 and 0110
[ACK SENT] Sent ACK 1
------------------------------------------------
[RECV] Received message: 2, 110110
[CRC SUCESS] 0101 and 0101
[ACK SENT] Sent ACK 2
------------------------------------------------
```

```
[RECV] Received message: 3, 100101
[CRC SUCESS] 0101 and 0101
[ACK SENT] Sent ACK 3
------------------------------------------------
[RECV] Received message: 4, 110001
[CRC FAILURE] 1100 and 1010
[NAK SENT] Sent NAK 4
------------------------------------------------
[RECV] Received message: 5, 000111
[CRC FAILURE] 1001 and 0000
[NAK SENT] Sent NAK 5
------------------------------------------------
[RECV] Received message: 6, 011001
[CRC FAILURE] 1101 and 0111
[NAK SENT] Sent NAK 6
------------------------------------------------
[RECV] Received message: 7, 011111
[CRC FAILURE] 0111 and 0000
[NAK SENT] Sent NAK 7
------------------------------------------------
[RECV] Received message: 4, 110011
[CRC SUCESS] 1010 and 1010
[ACK SENT] Sent ACK 4
------------------------------------------------
[RECV] Received message: 5, 100110
[CRC SUCESS] 0000 and 0000
[ACK SENT] Sent ACK 5
------------------------------------------------
```

```
------------------------------------------------
[RECV] Received message: 6, 111001
[CRC SUCESS] 0111 and 0111
[ACK SENT] Sent ACK 6
------------------------------------------------
[RECV] Received message: 7, 100110
[CRC SUCESS] 0000 and 0000
[ACK SENT] Sent ACK 7
------------------------------------------------
[CLOSING] Closing receiver....
```

Results

Packet Size: 1200 bits

- Ideal Environment(No error/Packet loss):
    - Stop and Wait ARQ: 4.20s
    - Go Back N ARQ: 3.58s
    - Selective Repeat ARQ: 3.51s

- Error-Injection/Packet loss possible environment:
    - Stop and Wait ARQ: 13.86s
    - Go Back N ARQ: 8.96s
    - Selective Repeat ARQ: 6.54s

The value of packet loss probability and error loss probability were both kept at 10%. All the protocols are simulated for 100 packets and average time is taken.

Selective Repeat ARQ does better than Go Back N ARQ and both of them do significantly better than Stop And Wait ARQ.

COMMENTS:

This assignment has significantly deepened my understanding of various flow control mechanisms through both research and practical implementation. By exploring and applying these techniques, I gained valuable insights into their strengths and limitations. I learned how the shortcomings of one flow control method can be effectively addressed by alternative approaches, enhancing the overall reliability and efficiency of data transmission.

I would like to extend my sincere gratitude to our teacher for guiding us through this learning process. Their support and encouragement have been instrumental in helping me grasp these concepts more thoroughly.