COMPUTER NETWORKS

LAB REPORT

ASSIGNMENT 2

DEBJIT DHAR

BCSE UG 3

ROLL:002210501106

GROUP: A3

SUBMISSION: 29/09/2024

Problem Statement: Implement flow control mechanisms for Data Link Layer.

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

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.

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).
Test the above three schemes for the following cases (not limited to).

● Compare time between the propagation of a packet and reception of its ACK.
● Compare efficiency of the above approaches without error or lost frame.
● 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.

DESIGN

Theory:

Stop and Wait ARQ:

Stop-and-wait ARQ, also referred to as alternating bit protocol, is a method in telecommunications to send information between two connected devices. It ensures that information is not lost due to dropped packets and that packets are received in the correct order. It is the simplest automatic repeat-request (ARQ) mechanism. A stop-and-wait ARQ sender sends one frame at a time; it is a special case of the general sliding window protocol with transmit and receive window sizes equal to one in both cases. After sending each frame, the sender doesn't send any further frames until it receives an acknowledgement (ACK) signal. After receiving a valid frame, the receiver sends an ACK. If the ACK does not reach the sender before a certain time, known as the timeout, the sender sends the same frame again. The timeout countdown is reset after each frame transmission. The above behavior is a basic example of Stop-and-Wait. However, real-life implementations vary to address certain issues of design.

Typically the transmitter adds a redundancy check number to the end of each frame. The receiver uses the redundancy check number to check for possible damage. If the receiver sees that the frame is good, it sends an ACK. If the receiver sees that the frame is damaged, the receiver discards it and does not send an ACK—pretending that the frame was completely lost, not merely damaged.

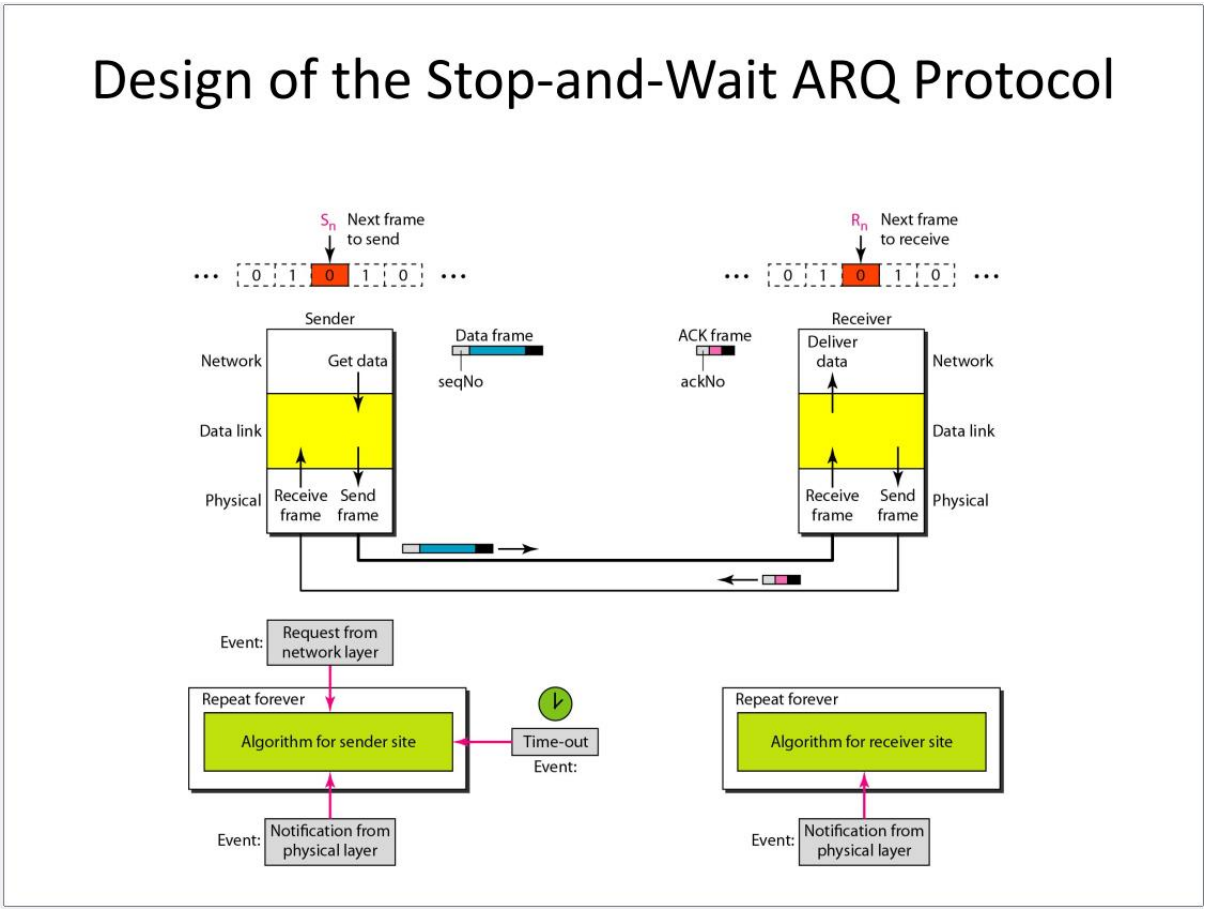
One problem is when the ACK sent by the receiver is damaged or lost. In this case, the sender doesn't receive the ACK, times out, and sends the frame again. Now the receiver has two copies of the same frame, and doesn't know if the second one is a duplicate frame or the next frame of the sequence carrying identical DATA.

Another problem is when the transmission medium has such a long latency that the sender's timeout runs out before the frame reaches the receiver. In this case the sender resends the same packet. Eventually the receiver gets two copies of the same frame, and sends an ACK for each one. The sender, waiting for a single ACK, receives two ACKs, which may cause problems if it assumes that the second ACK is for the next frame in the sequence.

To avoid these problems, the most common solution is to define a 1 bit sequence number in the header of the frame. This sequence number alternates (from 0 to 1) in subsequent frames. When the receiver sends an ACK, it includes the sequence number of the next packet it expects. This way, the receiver can detect duplicated frames by checking if the frame sequence numbers alternate. If two subsequent frames have the same sequence number, they are duplicates, and the second frame is discarded. Similarly, if two subsequent ACKs reference the same sequence number, they are acknowledging the same frame.

Stop-and-wait ARQ is inefficient compared to other ARQs, because the time between packets, if the ACK and the data are received successfully, is twice the transit time (assuming the

turnaround time can be zero). The throughput on the channel is a fraction of what it could be. To solve this problem, one can send more than one packet at a time with a larger sequence number and use one ACK for a set. This is what is done in Go-Back-N ARQ and the Selective Repeat ARQ.



Sender-site algorithm for Stop-and-Wait ARQ

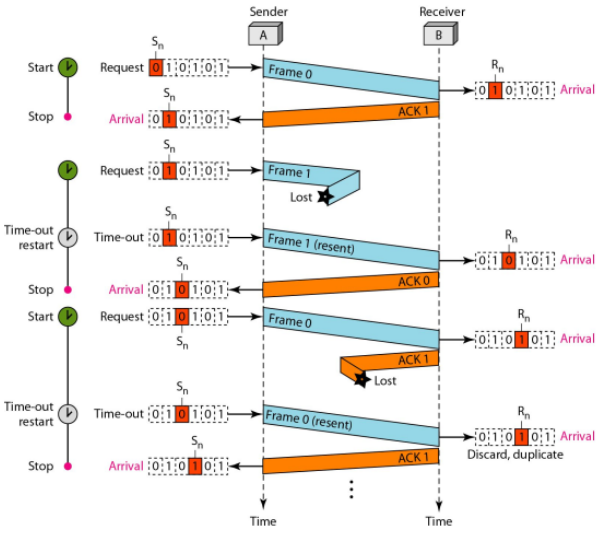
```
1  Sn = 0;                              // Frame 0 should be sent first
2  canSend = true;                      // Allow the first request to go
3  while(true)                          // Repeat forever
4  {
5    WaitForEvent();                    // Sleep until an event occurs

6    if(Event(RequestToSend) AND canSend)
7    {
8       GetData();
9       MakeFrame(Sn);                  //The seqNo is Sn
10      StoreFrame(Sn);                 //Keep copy
11      SendFrame(Sn);
12      StartTimer();
13      Sn = Sn + 1;
14      canSend = false;
15   }
16   WaitForEvent();                    // Sleep

17      if(Event(ArrivalNotification)   // An ACK has arrived
18      {
19        ReceiveFrame(ackNo);          //Receive the ACK frame
20        if(not corrupted AND ackNo == Sn) //Valid ACK
21          {
22            Stoptimer();
23            PurgeFrame(Sn-1);         //Copy is not needed
24            canSend = true;
25          }
26       }
27
28      if(Event(TimeOut)               // The timer expired
29      {
30       StartTimer();
31       ResendFrame(Sn-1);             //Resend a copy check
32      }
33 }
```

Receiver-site algorithm for Stop-and-Wait ARQ Protocol

```
1  Rn = 0;                          // Frame 0 expected to arrive first
2  while(true)
3  {
4    WaitForEvent();                // Sleep until an event occurs
5    if(Event(ArrivalNotification))  //Data frame arrives
6    {
7       ReceiveFrame();
8       if(corrupted(frame));
9          sleep();
10      if(seqNo == Rn)              //Valid data frame
11      {
12       ExtractData();
13        DeliverData();             //Deliver data
14        Rn = Rn + 1;
15      }
16       SendFrame(Rn);             //Send an ACK
17   }
18 }
```

Flow diagram

- *Frame 0 is sent and acknowledged.*
- *Frame 1 is lost and resent after the time-out.*
- *The resent frame 1 is acknowledged and the timer stops.*
- *Frame 0 is sent and acknowledged, but the acknowledgment is lost.*
- *The sender has no idea if the frame or the acknowledgment is lost, so after the time-out, it resends frame 0, which is acknowledged.*



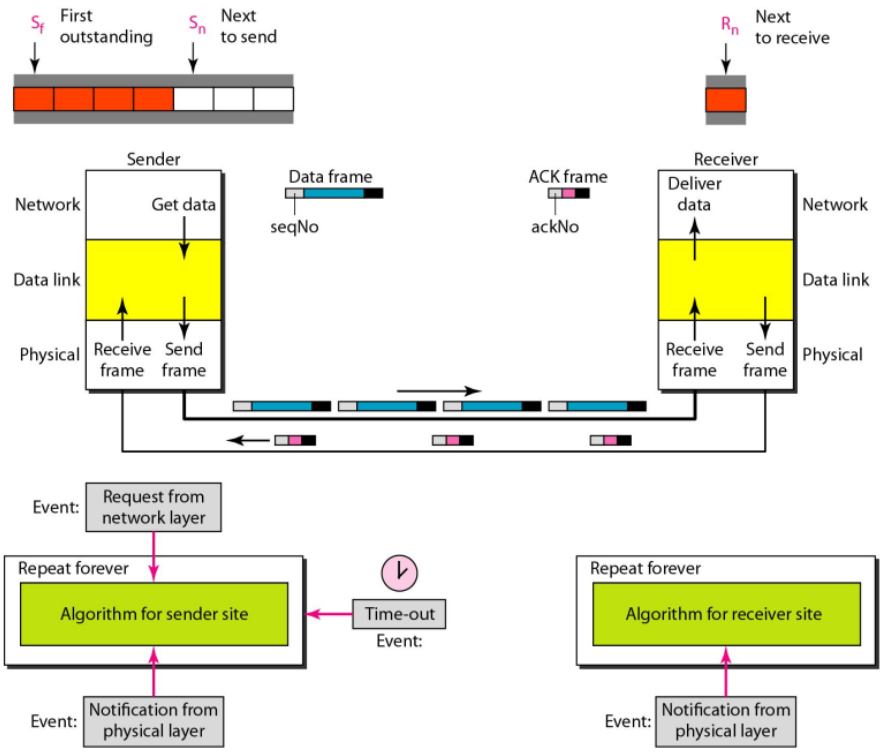
Go Back N ARQ:
Go-Back-N ARQ is a specific instance of the automatic repeat request (ARQ) protocol, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver. It is a special case of the general sliding window protocol with the transmit window size of N and receive window size of 1. It can transmit N frames to the peer before requiring an ACK.

The receiver process keeps track of the sequence number of the next frame it expects to receive. It will discard any frame that does not have the exact sequence number it expects (either a duplicate frame it already acknowledged, or an out-of-order frame it expects to receive later) and will send an ACK for the last correct in-order frame.[1] Once the sender has sent all of the frames in its window, it will detect that all of the frames since the first lost frame are outstanding, and will go back to the sequence number of the last ACK it received from the receiver process and fill its window starting with that frame and continue the process over again.

Go-Back-N ARQ is a more efficient use of a connection than Stop-and-wait ARQ, since unlike waiting for an acknowledgement for each packet, the connection is still being utilized as packets are being sent. In other words, during the time that would otherwise be spent waiting, more packets are being sent. However, this method also results in sending frames multiple times – if any frame was lost or damaged, or the ACK acknowledging them was lost or damaged, then that frame and all following frames in the send window (even if

they were received without error) will be re-sent. To avoid this, Selective Repeat ARQ can be used.

Design of Go-Back-N ARQ



Go-Back-N sender algorithm

```
1  Sw = 2^m - 1;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                          //Repeat forever
6  {
7   WaitForEvent();
8    if(Event(RequestToSend))            //A packet to send
9    {
10      if(Sn-Sf >= Sw)                  //If window is full
11            Sleep();
12      GetData();
13      MakeFrame(Sn);
14      StoreFrame(Sn);
15      SendFrame(Sn);
16      Sn = Sn + 1;
17      if(timer not running)
18           StartTimer();
19   }
20
21   if(Event(ArrivalNotification)) //ACK arrives
22   {
23      Receive(ACK);
24      if(corrupted(ACK))
25           Sleep();
26      if((ackNo>Sf)&&(ackNo<=Sn))  //If a valid ACK
27      While(Sf <= ackNo)
28       {
29        PurgeFrame(Sf);
30        Sf = Sf + 1;
31       }
32       StopTimer();
33   }
34
35   if(Event(TimeOut))                  //The timer expires
36   {
37    StartTimer();
38    Temp = Sf;
39    while(Temp < Sn);
40     {
41      SendFrame(Sf);
42      Sf = Sf + 1;
43     }
44   }
45 }
```

Go-Back-N receiver algorithm

```
1  Rn = 0;
2
3  while (true)                          //Repeat forever
4  {
5    WaitForEvent();
6
7    if(Event(ArrivalNotification)) /Data frame arrives
8    {
9       Receive(Frame);
10      if(corrupted(Frame))
11           Sleep();
12      if(seqNo == Rn)                  //If expected frame
13      {
14        DeliverData();                 //Deliver data
15        Rn = Rn + 1;                   //Slide window
16        SendACK(Rn);
17      }
18   }
19 }
```

Selective Repeat ARQ:
It may be used as a protocol for the delivery and acknowledgement of message units, or it may be used as a protocol for the delivery of subdivided message sub-units.

When used as the protocol for the delivery of messages, the sending process continues to send a number of frames specified by a window size even after a frame loss. Unlike Go-Back-N ARQ, the receiving process will continue to accept and acknowledge frames sent after an initial error; this is the general case of the sliding window protocol with both transmit and receive window sizes greater than 1.

The receiver process keeps track of the sequence number of the earliest frame it has not received, and sends that number with every acknowledgement (ACK) it sends. If a frame from

the sender does not reach the receiver, the sender continues to send subsequent frames until it has emptied its window. The receiver continues to fill its receiving window with the subsequent frames, replying each time with an ACK containing the sequence number of the earliest missing frame. Once the sender has sent all the frames in its window, it re-sends the frame number given by the ACKs, and then continues where it left off.

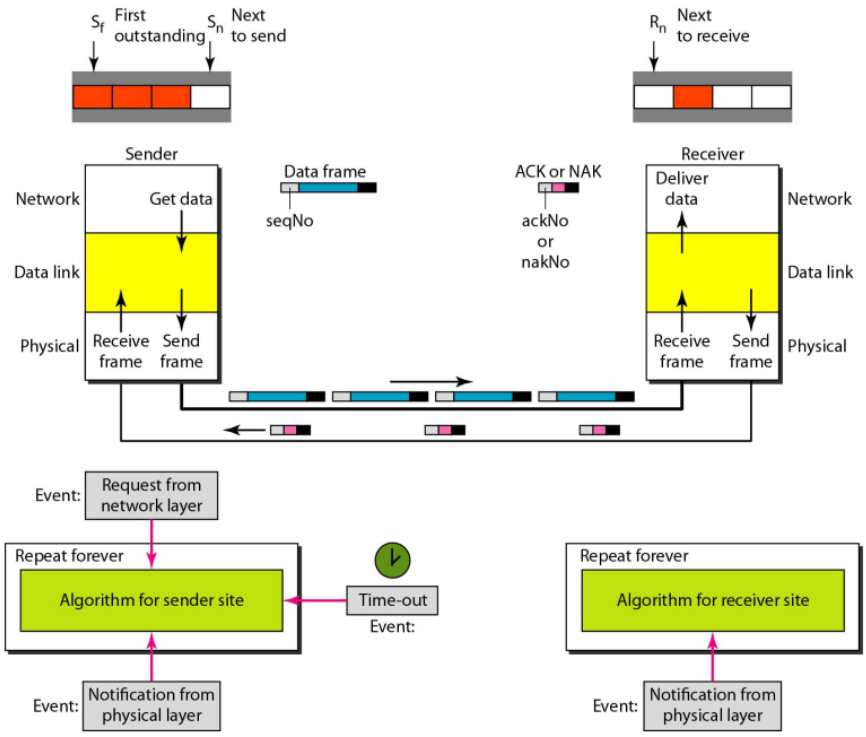
The size of the sending and receiving windows must be equal, and half the maximum sequence number (assuming that sequence numbers are numbered from 0 to n−1) to avoid miscommunication in all cases of packets being dropped. To understand this, consider the case when all ACKs are destroyed. If the receiving window is larger than half the maximum sequence number, some, possibly even all, of the packets that are present after timeouts are duplicates that are not recognized as such. The sender moves its window for every packet that is acknowledged.[1]

When used as the protocol for the delivery of subdivided messages it works somewhat differently. In non-continuous channels where messages may be variable in length, standard ARQ or Hybrid ARQ protocols may treat the message as a single unit. Alternately selective retransmission may be employed in conjunction with the basic ARQ mechanism where the message is first subdivided into sub-blocks (typically of fixed length) in a process called packet segmentation. The original variable length message is thus represented as a concatenation of a variable number of sub-blocks. While in standard ARQ the message as a whole is

either acknowledged (ACKed) or negatively acknowledged (NAKed), in ARQ with selective transmission the ACK response would additionally carry a bit flag indicating the identity of each sub-block successfully received. In ARQ with selective retransmission of sub-divided messages each retransmission diminishes in length, needing to only contain the sub-blocks that were linked.

In most channel models with variable length messages, the probability of error-free reception diminishes in inverse proportion with increasing message length. In other words, it's easier to receive a short message than a longer message. Therefore, standard ARQ techniques involving variable length messages have increased difficulty delivering longer messages, as each repeat is the full length. Selective re-transmission applied to variable length messages completely eliminates the difficulty in delivering longer messages, as successfully delivered sub-blocks are retained after each transmission, and the number of outstanding sub-blocks in following transmissions diminishes. Selective Repeat is implemented in UDP transmission.

Design of Selective Repeat ARQ



Sender-site Selective Repeat algorithm

```
1  Sw = 2^(m-1) ;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                          //Repeat forever
6  {
7    WaitForEvent();
8    if(Event(RequestToSend))            //There is a packet to send
9    {
10     if(Sn-Sf >= Sw)                   //If window is full
11           Sleep();
12     GetData();
13     MakeFrame(Sn);
14     StoreFrame(Sn);
15     SendFrame(Sn);
16     Sn = Sn + 1;
17     StartTimer(Sn);
18   }
19
20   if(Event(ArrivalNotification)) //ACK arrives
21   {
22      Receive(frame);                 //Receive ACK or NAK
23      if(corrupted(frame))
24           Sleep();
25      if (FrameType == NAK)
26         if (nakNo between Sf and Sn)
27         {
28          resend(nakNo);
29          StartTimer(nakNo);
30         }
31      if (FrameType == ACK)
32         if (ackNo between Sf and Sn)
33         {
34           while(sf < ackNo)
35           {
36            Purge(sf);
37            StopTimer(sf);
38            Sf = Sf + 1;
39           }
40         }
41   }
42
43   if(Event(TimeOut(t)))               //The timer expires
44   {
45    StartTimer(t);
46    SendFrame(t);
47   }
48 }
```

Receiver-site Selective Repeat algorithm

```
1  Rn = 0;
2  NakSent = false;
3  AckNeeded = false;
4  Repeat(for all slots)
5      Marked(slot) = false;
6
7  while (true)                                  //Repeat forever
8  {
9    WaitForEvent();
10
11   if(Event(ArrivalNotification))              /Data frame arrives
12   {
13      Receive(Frame);
14      if(corrupted(Frame))&& (NOT NakSent)
15      {
16       SendNAK(Rn);
17       NakSent = true;
18       Sleep();
19      }
20      if(seqNo <> Rn)&& (NOT NakSent)
21      {
22       SendNAK(Rn);
23       NakSent = true;
24       if ((seqNo in window)&&(!Marked(seqNo))
25       {
26        StoreFrame(seqNo)
27        Marked(seqNo)= true;
28        while(Marked(Rn))
29        {
30         DeliverData(Rn);
31         Purge(Rn);
32         Rn = Rn + 1;
33         AckNeeded = true;
34        }
35         if(AckNeeded);
36         {
37         SendAck(Rn);
38         AckNeeded = false;
39         NakSent = false;
40         }
41       }
42      }
43   }
44 }
```

Stop and Wait ARQ Program Overview:

1. General Description:

- The system is divided into three main components:
  - **Sender Program (Sender.ipynb):** Responsible for sending frames with a sequence number and data, managing retransmissions in case of timeout or acknowledgment failure.
  - **Receiver Program (Receiver.ipynb):** Receives the frames, checks for errors using CRC (Cyclic Redundancy Check), and sends an acknowledgment if the frame is correct.
  - **Utility Functions (Utils.ipynb):** Includes helper functions for CRC computation, noise injection, and frame handling.
  - **Statistics Module (Statistics.ipynb):** Defines important constants such as frame structure (N_SIZE, DATA_SIZE, CRC_SIZE), and other network parameters.
- The *Stop and Wait ARQ* protocol operates by sending a single frame, waiting for an acknowledgment before sending the next frame. In case of a timeout or CRC error, the frame is resent.

2. Input/Output Format:

- **Sender Input:**
  - Data input by the user to be transmitted.
  - The data frame format is as follows: [NN][LLL][DDDDDDDDDDD][CCCC] where:
    - NN: Sequence number (2 digits).
    - LLL: Length of the data (3 digits).
    - DDDDDDDDDDD: Data part of fixed size (11 characters).

        - CCCC: CRC checksum (4 digits).
- **Sender Output:**
    - The program prints the frames being sent, any retransmissions due to timeout, and acknowledgment received.
- **Receiver Input:**
    - Frames received from the sender.
    - The frame is analyzed for sequence number, data integrity (via CRC), and errors.
- **Receiver Output:**
    - Prints whether the received frame was successfully received (correct CRC) or whether retransmission is required (CRC failure or out-of-order frame).

3. How to Execute:

- **Step 1: Run the Receiver.ipynb program first.**
    - The receiver starts listening for incoming frames. This will print messages as it waits for data from the sender.
- **Step 2: Run the Sender.ipynb program.**
    - The sender will prompt you to enter the data that you wish to send.
    - The program will attempt to send the frame, handling timeouts or errors (due to the noisy channel simulation) and waiting for acknowledgment from the receiver.
- **Step 3: Communicate between the sender and receiver.**
    - Both programs must run simultaneously in different terminals or environments (or on different systems connected via sockets).

    - The sender will continuously send frames and wait for acknowledgment, while the receiver listens, validates frames, and sends acknowledgments back.
- **Step 4: Terminate communication by typing 'q' in the sender input.**
    - This signals the receiver to close the connection and terminate the session.

Go Back N ARQ Program Overview:

The four main components of this system are:

1. **Utils.ipynb**: Provides utility functions for CRC (Cyclic Redundancy Check) generation, simulating a noisy channel, and frame creation/processing.
2. **Sender.ipynb**: Implements the Go-Back-N sender logic, which sends frames, manages the sliding window, and handles acknowledgment (ACK) reception and retransmission upon timeouts.
3. **Receiver.ipynb**: Implements the receiver logic, checks frame integrity using CRC, and acknowledges correctly received frames.
4. **Statistics.ipynb**: Contains network settings, constants such as CRC size, frame size, and configurations like socket addresses.

Input/Output Format

- **Input**:
    - The sender reads data from a file (data.txt) to be transmitted to the receiver.
- **Output**:

- **Sender** prints the status of the frames being sent, ACKs received, and frames being resent in case of timeouts or errors.
- **Receiver** prints the details of received frames, checks for errors using CRC, and sends ACKs for correctly received frames.

Execution of the Programs

1. **Sender Program Execution**:
   - Run Sender.ipynb, which reads the data from a file, transmits frames to the receiver in a sliding window manner, and handles timeouts/resends if ACKs are not received in time.
2. **Receiver Program Execution**:
   - Run Receiver.ipynb, which listens for incoming frames, checks for CRC errors, and sends acknowledgments for successfully received frames.

Selective Repeat ARQ Program Overview

Overview of the Programs

The implementation consists of four main files:

1. **Utils.ipynb**: Contains utility functions for manipulating frames, including creating frames, adding noise, calculating CRC, and checking received frames for errors.
2. **Sender.ipynb**: Implements the sender side of the protocol. It sends frames to the receiver, handles acknowledgments (ACKs) and negative acknowledgments (NAKs), and can resend frames if necessary.

3. **Receiver.ipynb**: Implements the receiver side. It receives frames from the sender, checks for corruption using CRC, and sends back appropriate ACKs or NAKs based on the received data.
4. **Statistics.ipynb**: Contains constants and global variables used across the other files, such as frame sizes and the host address.

Input/Output Format

Input Format:

- The sender reads data from a file named data.txt. Each line of this file is treated as a separate data packet to be sent.
- The connection is established over a socket using TCP/IP.

Output Format:

- The sender outputs frames being sent, ACKs received, and any NAKs sent back.
- The receiver outputs the frames it receives, whether they were corrupted, and the corresponding ACK or NAK responses.

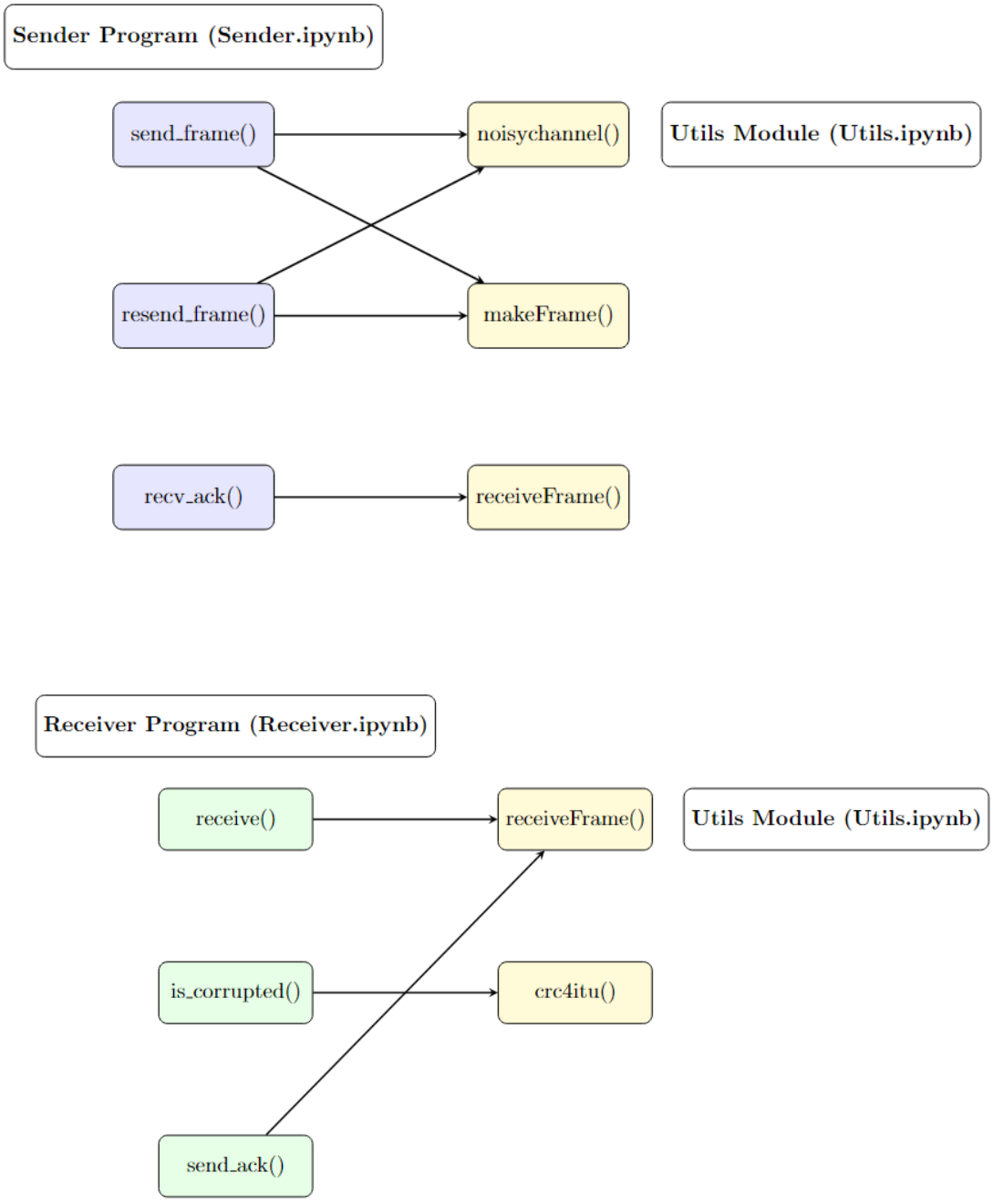
Execution of the Entire Set of Programs

1. **Set Up the Environment**:
    - Ensure Python is installed on your machine.
    - Install necessary packages (if any).
2. **Create data.txt**:
    - Populate a data.txt file with sample data to be sent. Each line will be a separate data packet.
3. **Run the Programs**:
    - Start the sender program (Sender.ipynb).
    - Start the receiver program (Receiver.ipynb).

- The programs will communicate over a socket, sending and receiving frames as defined by the protocol.

4. **Monitoring**:
   - Monitor the console output of both the sender and receiver to see the status of frames, ACKs, and NAKs.

Structural Diagram



Implementation

Implementable code at: https://github.com/Debjit-Dhar/Networks

STOP AND WAIT ARQ

Utils.ipynb

```
import random
import time
import Statistics as st


# Inject noise into the data part of the frame
def noisychannel(frame):
    data_part = list(frame[-st.CRC_SIZE - st.DATA_SIZE: -st.CRC_SIZE])
    # Flip random bits in the data part
    for index in random.sample(range(len(data_part)), random.randint(0, len(data_part) - 1)):
        data_part[index] = '1' if data_part[index] == '0' else '0'
    # Rebuild the frame
    return frame[:st.N_SIZE + st.LENGTH_SIZE] + ''.join(data_part) + frame[-st.CRC_SIZE:]


# Introduce random delay (up to 5 seconds)
def delay():
    time.sleep(random.randint(0, 5))


# XOR operation between two binary strings
def xor(a, b):
    return ''.join('0' if x == y else '1' for x, y in zip(a, b))


# Perform binary division for CRC calculation
def binary_division(dividend, divisor):
```

```
    rem = dividend[:len(divisor)]
    for bit in dividend[len(divisor):]:
        rem = xor(rem, divisor)[1:] if rem[0] == '1' else rem[1:]
        rem += bit
    return rem

# CRC-4 ITU calculation
def crc4itu(frame):
    return binary_division(frame + '0000', "10011")

# Create a frame with sequence number and data
def makeFrame(n, data):
    return
(f"{str(n).zfill(st.N_SIZE)}{str(len(data)).zfill(st.LENGTH_SIZE)}"

f"{data.zfill(st.DATA_SIZE)}{crc4itu(data).zfill(st.CRC_SIZE)}")

# Extract details from received frame
def receiveFrame(frame):
    n = int(frame[:st.N_SIZE])
    l = int(frame[st.N_SIZE:st.N_SIZE + st.LENGTH_SIZE])
    data = frame[-st.CRC_SIZE - l: -st.CRC_SIZE]
    crc = frame[-st.CRC_SIZE:] if data != 'q' else ""
    return n, l, data, crc
```

Explanation

1. noisychannel(frame)

- **Purpose**: To simulate transmission errors by injecting noise (flipping random bits) into the data part of the frame.
- **Parameters**:
    - frame (string): The frame containing data and CRC that is being transmitted.
- **Process**:

1. Extracts the data part of the frame.
2. Randomly selects bits within the data part and flips them (i.e., changes 0 to 1 and vice versa).
3. Rebuilds the frame with the modified (noisy) data part.

4. Returns the frame with the added noise.

2. delay()

- **Purpose**: To introduce a random delay in transmission, simulating network latency.
- **Parameters**: None.
- **Process**:
    1. Introduces a random sleep (pause) for up to 5 seconds (random.randint(0, 5)).
    2. This is used to simulate delays in the transmission process.

3. xor(a, b)

- **Purpose**: To perform a bitwise XOR operation between two binary strings.
- **Parameters**:
    - a (string): The first binary string.
    - b (string): The second binary string (should be of the same length as a).
- **Process**:

1. Compares corresponding bits of a and b and performs XOR (i.e., returns '0' if bits are the same, '1' if they are different).

2. Returns the resulting binary string after XOR.

4. binary_division(dividend, divisor)

- **Purpose**: To perform the binary division used for CRC error-checking.
- **Parameters**:
    - dividend (string): The binary string to be divided (the frame data with extra 0000 bits).
    - divisor (string): The CRC-4 ITU divisor, which is a fixed polynomial (10011).
- **Process**:

1. Takes the initial portion of the dividend (equal to the length of the divisor) as the remainder.

2. Iteratively performs XOR (if the remainder starts with 1) or just removes the leading bit (if it starts with 0) while appending the next bit from the dividend.

3. Returns the remainder, which is the result of the binary division and represents the CRC.

5. crc4itu(frame)

- **Purpose**: To compute the CRC (Cyclic Redundancy Check) for a given frame using the CRC-4 ITU standard.
- **Parameters**:

- frame (string): The binary string (data part of the frame) for which CRC needs to be calculated.

- **Process**:

1. Appends four 0 bits to the frame.

2. Calls binary_division() to divide the modified frame by the CRC-4 ITU polynomial (10011).

3. Returns the remainder of the division, which is the CRC value.

6. makeFrame(n, data)

- **Purpose**: To create a frame that includes the sequence number, length of data, the data itself, and the CRC.

- **Parameters**:

  - n (integer): The sequence number of the frame (used in sliding window protocols).

  - data (string): The data to be transmitted in the frame.

- **Process**:

1. Converts the sequence number n to a zero-padded string based on st.N_SIZE.

2. Converts the length of the data to a zero-padded string based on st.LENGTH_SIZE.

3. Zero-pads the data to fit the st.DATA_SIZE.

4. Computes the CRC for the data using crc4itu().

5. Combines the sequence number, data length, data, and CRC into a single frame string and returns it.

7. receiveFrame(frame)

- **Purpose**: To extract and decode information (sequence number, data length, data, and CRC) from a received frame.

- **Parameters**:

  - frame (string): The frame string that was received over the network.

- **Process**:

1. Extracts the sequence number from the frame by reading the first st.N_SIZE characters.

2. Extracts the length of the data from the next st.LENGTH_SIZE characters.

3. Extracts the actual data from the frame (based on the length extracted).

4. Extracts the CRC from the end of the frame (unless the data indicates termination with 'q').

5. Returns the sequence number, data length, data, and CRC as separate variables.

Sender.ipynb

```
import socket
import time
import random
import threading
import Utils as ut
import Statistics as st

sn = 0  # Sequence number
TIMEOUT_LIMIT = 2  # Timeout for resending frames
copy_sn, copy_data, can_send = 0, "", True

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

def send_frame(conn, data):
    global sn, can_send, copy_sn, copy_data, time1

    # Send frame if allowed
    if can_send:
        frame = ut.noisychannel(ut.makeFrame(sn, data))
        print(f"[SENDING] Frame: {frame}")
        conn.send(frame.encode())

        # Store frame for resending
        copy_sn, copy_data, can_send = sn, data, False
        time1 = time.time()
        sn += 1

        recv_ack(conn)

    # Resend on timeout
```

```
    while not can_send:
        resend_frame(conn)

def resend_frame(conn):
    global time1
    frame = ut.noisychannel(ut.makeFrame(copy_sn, copy_data))
    print(f"[RESENDING] Frame: {frame}")
    time1 = time.time()
    conn.send(frame.encode())
    recv_ack(conn)

def recv_ack(conn):
    global can_send
    try:
        conn.settimeout(TIMEOUT_LIMIT)
        ack_frame = conn.recv(20).decode()
        ack_no, _, _, data = ut.receiveFrame(ack_frame)

        if ack_no == copy_sn and data == '11110000':
            print(f"[ACK RECEIVED] ACK {ack_no}")
            can_send = True
    except socket.timeout:
        print("---[TIMEOUT]---")

def start():
    sender.listen()
    print(f"[LISTENING] Server is listening on {st.HOST_IP}")
    conn, addr = sender.accept()
    print(f"[CONNECTED] Connected to {addr}")

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

        if data == 'q':
```

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

start()
```

Explanation

1. send_frame(conn, data)

- Purpose: To create and send a frame containing the specified data and manage acknowledgments for successful transmission.
- Parameters:
  - conn (socket): The socket connection through which the frame is sent.
  - data (string): The actual data to be transmitted in the frame.
- Process:

1. Checks if sending is allowed (can_send).
2. If allowed, creates a frame using ut.makeFrame(sn, data) and introduces noise using ut.noisychannel().
3. Sends the encoded frame to the receiver.
4. Stores the sequence number (sn) and data for potential resending.
5. Increments the sequence number and sets can_send to False.
6. Calls recv_ack(conn) to wait for an acknowledgment.
7. If not allowed to send (can_send is False), enters a loop to check for timeouts and resend the frame if necessary.

---

2. resend_frame(conn)

- Purpose: To resend the previously sent frame if an acknowledgment is not received in a timely manner.
- Parameters:
  - conn (socket): The socket connection used for resending the frame.
- Process:

1. Creates a new frame using ut.makeFrame(copy_sn, copy_data) and introduces noise using ut.noisychannel().
2. Sends the newly created frame to the receiver.
3. Updates the timer (time1) for tracking when the frame was resent.

4. Calls recv_ack(conn) to wait for an acknowledgment from the receiver.

---

3. recv_ack(conn)

- Purpose: To receive and validate an acknowledgment frame from the receiver.
- Parameters:
  - conn (socket): The socket connection from which the acknowledgment frame is received.
- Process:

1. Sets a timeout for receiving the acknowledgment frame (TIMEOUT_LIMIT).
2. Waits to receive an acknowledgment frame from the connection.
3. If received, extracts the acknowledgment number and checks it against the stored sequence number (copy_sn).
4. If the acknowledgment is valid (correct sequence number and ACK signal), sets can_send to True to allow sending new frames.
5. If a timeout occurs, prints a timeout message.

---

4. start()

- Purpose: To initiate the server socket, listen for connections, and handle user input for sending data.
- Parameters: None.
- Process:
  1. Starts the server to listen for incoming connections.
  2. Accepts a connection from a client and prints the connected address.
  3. Enters a loop to accept user input for data to send.
  4. Calls send_frame(conn, data) to send the entered data.
  5. If the user enters 'q', prints a closing message and terminates the connection.

Receiver.ipynb

```
import socket
import Utils as ut
import Statistics as st

rn = 0  # Sequence number
receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)

def is_corrupted(data, crc):
    return ut.crc4itu(data) != crc

def send_ack():
    ack_frame = ut.makeFrame(rn, "11110000")
    receiver.send(ack_frame.encode())
    print(f"[ACK SENT] ACK {rn}")

def receive():
    global rn
    while True:
        print("[LISTENING] Receiver is listening...")
        frame = receiver.recv(20).decode()
        recv_n, _, data, crc = ut.receiveFrame(frame)

        if data == 'q':  # Check for termination
            print("[CLOSING] Closing receiver...")
            receiver.close()
            break

        print(f"[RECEIVED] Frame: {recv_n}, Data: {data}")

        if recv_n == rn:
            if not is_corrupted(data, crc):  # Check CRC
                print(f"[CRC SUCCESS] CRC {crc} matches.")
                send_ack()
```

```
                rn += 1
            else:
                print(f"[CRC FAILURE] Mismatch: Calculated
{ut.crc4itu(data)}, Received {crc}")
        else:
            print("[OUT OF ORDER] Received wrong frame.")
        print("-----------------------------------------------")

receive()
```

Explanation

1. is_corrupted(data, crc)

- **Purpose**: To determine if the received data has been corrupted by comparing the calculated CRC with the received CRC.
- **Parameters**:
  - data (string): The data part of the received frame.
  - crc (string): The CRC value received with the frame.
- **Process**:

1. Calls ut.crc4itu(data) to calculate the CRC for the received data.
2. Compares the calculated CRC with the received CRC.
3. Returns True if they do not match (indicating corruption), otherwise returns False.

---

2. send_ack()

- **Purpose**: To send an acknowledgment (ACK) frame back to the sender indicating successful receipt of a frame.
- **Parameters**: None.
- **Process**:
  1. Creates an acknowledgment frame using ut.makeFrame(rn, "11110000"), where rn is the current sequence number.
  2. Sends the encoded acknowledgment frame back to the sender.
  3. Prints a message indicating that the acknowledgment has been sent.

---

3. receive()

- **Purpose**: To continuously listen for incoming frames from the sender, validate them, and manage acknowledgments.
- **Parameters**: None.

- **Process**:
    1. Enters an infinite loop to listen for incoming frames.
    2. Prints a message indicating that the receiver is listening.
    3. Receives a frame from the socket and decodes it.
    4. Extracts the sequence number (recv_n), length, data, and CRC from the received frame using ut.receiveFrame(frame).
    5. Checks if the received data is a termination signal ('q'):
        - If so, closes the receiver socket and breaks the loop.
    6. Prints the received frame information.
    7. Checks if the received sequence number matches the expected sequence number (rn):
        - If they match, checks if the data is not corrupted using is_corrupted(data, crc).
            - If not corrupted, sends an acknowledgment and increments the sequence number (rn).
            - If corrupted, prints a mismatch message with the calculated and received CRC values.
        - If they do not match, prints an out-of-order message.
    8. Ends with a separator line for clarity in output.

GO BACK N ARQ

Utils.ipynb

```
import random, Statistics as st, time

# Simulate a noisy channel by flipping random bits in the data section of
the frame

def noisychannel(frame):

    data_start = -st.CRC_SIZE - st.DATA_SIZE

    frame_list = list(frame[data_start:-st.CRC_SIZE])  # Extract data
section as list

    num_bits_to_flip = random.randint(0, len(frame_list) - 1)


    positions = random.sample(range(len(frame_list)), num_bits_to_flip)

    for pos in positions:

        frame_list[pos] = '1' if frame_list[pos] == '0' else '0'  # Flip
bits


    return frame[:data_start] + ''.join(frame_list) + frame[-st.CRC_SIZE:]


# Simulate a random delay of up to 5 seconds

def delay():

    time.sleep(random.randint(0, 5))


# XOR operation between two binary strings

def xor(a, b):

    return ''.join('0' if x == y else '1' for x, y in zip(a, b))


# Perform binary division for CRC calculation

def binary_division(dividend, divisor):

    rem = dividend[:len(divisor)]

    for i in range(len(divisor), len(dividend)):
```

```
        if rem[0] == '1':
            rem = xor(rem, divisor)[1:] + dividend[i]
        else:
            rem = rem[1:] + dividend[i]
    return rem[1:]  # Return remainder after division

# Calculate CRC using the CRC-4-ITU polynomial
def crc4itu(data):
    divisor = "10011"
    return binary_division(data + '0000', divisor)

# Create a frame with sequence number, length, data, and CRC
def makeFrame(n, data):
    frame = (
        str(n).zfill(st.N_SIZE) +  # Sequence number
        str(len(data)).zfill(st.LENGTH_SIZE) +  # Length
        data.zfill(st.DATA_SIZE) +  # Data (padded)
        crc4itu(data).zfill(st.CRC_SIZE)  # CRC
    )
    return frame

# Extract information from the received frame
def receiveFrame(frame):
    n = int(frame[:st.N_SIZE])  # Sequence number
    l = int(frame[st.N_SIZE:st.N_SIZE + st.LENGTH_SIZE])  # Length
    data = frame[-st.CRC_SIZE - l:-st.CRC_SIZE]  # Data
    crc = frame[-st.CRC_SIZE:] if data != 'q' else ''
    return n, l, data, crc
```

Explanation

1. noisychannel(frame)

Purpose: Simulates a noisy communication channel by randomly flipping bits in the data section of a given frame.

Parameters:

- frame (str): A binary string representing the frame, which includes sequence number, length, data, and CRC.

Process:

- Extracts the data section of the frame.
- Randomly selects bits in the data section and flips them.
- Returns the modified frame with the flipped bits.

---

2. delay()

Purpose: Simulates random transmission delay by pausing execution for a random duration.

Parameters:

- None.

Process:

- Pauses the execution for a random duration between 0 and 5 seconds.

---

3. xor(a, b)

Purpose: Performs a bitwise XOR operation between two binary strings.

Parameters:

- a (str): First binary string.
- b (str): Second binary string.

Process:

- Compares the corresponding bits of a and b.
- Produces a new string with each bit being '0' if the bits are the same, otherwise '1'.
- Returns the XOR result as a binary string.

---

4. binary_division(dividend, divisor)

Purpose: Performs binary division between the dividend and the divisor to calculate the remainder, which is used in CRC calculations.

Parameters:

- dividend (str): The binary string that is divided (data + padding).
- divisor (str): The binary string representing the CRC polynomial.

Process:

- Performs bitwise division by XOR between dividend and divisor.
- Returns the remainder (excluding the first bit) as the result of the division.

---

5. crc4itu(data)

Purpose: Calculates the CRC (Cyclic Redundancy Check) value using the CRC-4-ITU polynomial.

Parameters:

- data (str): The binary string for which the CRC value is calculated.

Process:

- Appends four zero bits to the data (as padding).
- Performs binary division between the padded data and the CRC-4-ITU polynomial ("10011").
- Returns the remainder of the division, which serves as the CRC value.

---

6. makeFrame(n, data)

Purpose: Constructs a communication frame that includes the sequence number, length, data, and CRC.

Parameters:

- n (int): The sequence number of the frame.
- data (str): The binary data to be sent in the frame.

Process:

- Pads the sequence number and data to fixed lengths.
- Calculates the length of the data and appends it to the frame.
- Computes the CRC for the data and adds it to the frame.
- Returns the complete frame as a string.

---

7. receiveFrame(frame)

Purpose: Extracts the sequence number, data length, data, and CRC from a received frame.

Parameters:

- frame (str): The received frame, containing sequence number, length, data, and CRC.

Process:

- Extracts the sequence number, data length, data, and CRC from the frame.
- If the data contains a termination signal ('q'), the CRC is ignored.
- Returns the extracted sequence number, length, data, and CRC.

Receiver.ipynb

```
import socket, threading, utils as ut, Statistics as st


rn = 0

receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

receiver.connect(st.ADDR)


# Check if the received frame is corrupted

def isCorrupted(data, crc):

    return op.crc4itu(data) != crc


# Function to handle receiving frames
```

```
def recvFrames():
    global rn
    while True:
        try:
            frame = receiver.recv(20).decode()
            recv_n, _, data, crc = op.receiveFrame(frame)
        except ValueError:  # Handles empty string when connection ends
            return

        print(f"[RECV] Received frame {recv_n}, data: {data}")

        if data == 'q':  # Disconnect signal
            break

        if not isCorrupted(data, crc) and recv_n == rn:
            print(f"[CRC SUCCESS] Data CRC: {op.crc4itu(data)}, Frame CRC:
{crc}")
            sendAck()
            rn += 1
            print("[ACK SENT] Acknowledgement sent")
        else:
            print(f"[CRC FAILURE] Data CRC: {op.crc4itu(data)}, Frame CRC:
{crc}")

# Function to send an acknowledgment
def sendAck():
    ack_frame = op.makeFrame(rn, "11110000")  # Acknowledgment frame
    receiver.send(ack_frame.encode())

# Start receiver thread
receiver_thread = threading.Thread(target=recvFrames)
```

```
receiver_thread.start()

receiver_thread.join()


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

receiver.close()
```

Explanation

1. isCorrupted(data, crc)

Purpose: Determines whether the received frame is corrupted by checking if the calculated CRC matches the received CRC.

Parameters:

- data (str): The data portion of the received frame.
- crc (str): The CRC received with the frame.

Process:

- Uses the op.crc4itu(data) function to calculate the CRC of the data.
- Compares the calculated CRC with the received crc.
- Returns True if they don't match (indicating corruption), otherwise returns False.

---

2. recvFrames()

Purpose: Continuously listens for incoming frames, checks their integrity, and sends an acknowledgment if the frame is correctly received and in order.

Parameters:

- None.

Process:

- The function runs in a loop, receiving frames using receiver.recv(20) and decoding them.
- Extracts the sequence number (recv_n), data, and CRC from the received frame.
- If the data equals 'q', it terminates (disconnect signal).

- If the frame is not corrupted (verified by isCorrupted) and the sequence number matches the expected rn, it sends an acknowledgment using sendAck(), increments rn, and logs success.
- If the frame is corrupted, logs a CRC failure.

---

3. sendAck()

Purpose: Sends an acknowledgment frame back to the sender.

Parameters:

- None.

Process:

- Creates an acknowledgment frame using the current sequence number (rn) and the special acknowledgment signal "11110000".
- Sends the frame to the sender using receiver.send().

---

4. receiver_thread = threading.Thread(target=recvFrames)

Purpose: Starts a separate thread to handle the reception of frames concurrently.

Parameters:

- None (the target function recvFrames is passed to the thread).

Process:

- The thread runs the recvFrames function, allowing the receiver to handle incoming frames without blocking other tasks.
- After the thread completes, it joins back to the main thread, and the receiver socket is closed.

Sender.ipynb

```
import socket, threading, utils as ut, Statistics as st


sf, sn, sw = 0, 0, 4  # Sliding window variables

flag, flag2 = False, False  # Flags for control flow
```

```
data_queue = [None] * sw

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

def send_frames(conn):
    global sn, sf, sw, flag, data_queue
    while not flag:
        if sn - sf < sw:
            try:
                data = text.pop(0)  # Get the next data to send
            except IndexError:
                print("[FINISHED] All data sent.")
                flag = True
                return

            frame = op.noisychannel(op.makeFrame(sn, data))
            print(f"[SENDING] Frame {sn}: {frame}")
            conn.send(frame.encode())
            data_queue[sn % sw] = data
            sn += 1

def recv_ack(conn):
    global sf, sn, sw, data_queue, flag2
    while not flag2:
        try:
            conn.settimeout(0.5)
            recv_frame = conn.recv(20).decode()
            ackNo, _, data, _ = op.receiveFrame(recv_frame)
```

```
            except socket.timeout:
                resend_frames(conn)
                continue

            if sf <= ackNo < sn and data == '11110000':  # Valid ACK
                while sf <= ackNo:
                    print(f"[ACK RECEIVED] ACK {ackNo}")
                    data_queue[sf % sw] = None
                    sf += 1

            if ackNo == length - 1:  # All ACKs received
                flag2 = True

    def resend_frames(conn):
        for i in range(sf, sn):
            data = data_queue[i % sw]
            frame = op.makeFrame(i, data)
            print(f"[RESENDING] Frame {i}: {frame}")
            conn.send(frame.encode())

    def start_server():
        global text, length
        sender.listen()
        print(f"[LISTENING] Server listening on {st.HOST_IP}")

        conn, addr = sender.accept()
        print(f"[CONNECTED] Connected to {addr}")

        with open("data.txt", "r") as f:
            text = [line.strip() for line in f.readlines()]
```

```
    length = len(text)

    sender_thread = threading.Thread(target=send_frames, args=(conn,))
    receiver_thread = threading.Thread(target=recv_ack, args=(conn,))

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

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

start_server()
```

Explanation

1. send_frames(conn)

Purpose: Sends frames (data) to the receiver within the sliding window.

Parameters:

- conn (socket): The connection object to send data over the network.

Process:

- Checks if the sliding window limit (sw) is not exceeded by comparing sn - sf.
- Retrieves the next data from the text list. If no more data exists, it sets the flag to True and stops.
- Uses op.noisychannel() to simulate a noisy channel before sending the frame.
- Sends the frame via conn.send() and stores the data in data_queue for potential resending.
- Increments the sequence number sn after each frame is sent.

---

2. recv_ack(conn)

Purpose: Receives acknowledgment (ACK) frames from the receiver and handles valid ACKs.

Parameters:

- conn (socket): The connection object to receive acknowledgments from the network.

Process:

- Runs in a loop until all acknowledgments are received (flag2 is True).
- Attempts to receive an acknowledgment frame using conn.recv() within a timeout window (0.5 seconds). If a timeout occurs, it triggers the resend logic.
- Parses the received ACK and checks if it's within the valid range (sf <= ackNo < sn).
- Marks the frame as acknowledged by updating data_queue and increments sf.
- If the last ACK (matching length - 1) is received, it sets flag2 to True, ending the process.

---

3. resend_frames(conn)

Purpose: Resends frames that haven't been acknowledged yet.

Parameters:

- conn (socket): The connection object to resend frames over the network.

Process:

- Iterates through the range of unacknowledged frames (sf to sn).
- Retrieves each frame from the data_queue and constructs a frame using op.makeFrame().
- Sends the frame again via conn.send().

---

4. start_server()

Purpose: Initializes the server, reads data from a file, and starts the sender and receiver threads.

Parameters:

- None.

Process:

- Sets up the server to listen for incoming connections (sender.listen()).
- Accepts a connection from a receiver and prints the connection details.
- Reads the data from data.txt and prepares the text list.

- Starts two threads: one for sending frames (send_frames()) and one for receiving acknowledgments (recv_ack()).
- Waits for both threads to finish using join().
- Once finished, it closes the connection and terminates the server.

Selective Repeat ARQ

Utils.ipynb

```
import random
import time
import Statistics as st


# Inject noise into the data part of the frame
def noisychannel(frame):
    frame_list = list(frame[-st.CRC_SIZE-st.DATA_SIZE:-st.CRC_SIZE])
    # Randomly select positions to flip bits
    positions = random.sample(range(len(frame_list)), random.randint(0, len(frame_list) - 1))
    for index in positions:
        frame_list[index] = '1' if frame_list[index] == '0' else '0'


    # Rebuild the frame with the modified data
    return frame[:st.N_SIZE + st.LENGTH_SIZE] + ''.join(frame_list) + frame[-st.CRC_SIZE:]


# Introduce random delay up to 5 seconds
def delay():
    time.sleep(random.randint(0, 5))


# XOR operation between two binary strings
def xor(a, b):
    return ''.join('0' if a[i] == b[i] else '1' for i in range(len(b)))


# Perform binary division for CRC calculation
def binary_division(dividend, divisor):
    rem = dividend[:len(divisor)]
```

```
    for i in range(len(divisor), len(dividend)):
        rem = xor(rem, divisor)[1:] if rem[0] == '1' else rem[1:]
        rem += dividend[i]
    return rem

# CRC-4 ITU calculation
def crc4itu(frame):
    return binary_division(frame + '0000', "10011")

# Create a frame with given nth frame number and data
def makeFrame(n, data):
    frame =
f"{str(n).zfill(st.N_SIZE)}{str(len(data)).zfill(st.LENGTH_SIZE)}{data.zfi
ll(st.DATA_SIZE)}{crc4itu(data).zfill(st.CRC_SIZE)}"
    return frame

# Extract details from received frame
def receiveFrame(frame):
    n = int(frame[:st.N_SIZE])
    l = int(frame[st.N_SIZE:st.N_SIZE + st.LENGTH_SIZE])
    data = frame[-st.CRC_SIZE-l:-st.CRC_SIZE]
    crc = frame[-st.CRC_SIZE:] if data != "q" else ""
    return n, l, data, crc
```

Explanation

1. noisychannel(frame)

Purpose:
To simulate the introduction of random errors (noise) into the data portion of a communication frame.

Parameters:

- frame: A string representing the complete frame (including data and CRC bits) to which noise is applied.

Process:

- The data portion of the frame (excluding CRC bits) is extracted.
- Random positions within the data are selected, and the bits at those positions are flipped (0 becomes 1 and vice versa).

- The modified data portion is then combined with the unchanged portions of the frame to produce the noisy frame, which is returned.

---

2. delay()

Purpose:
To simulate a random delay (network latency) before sending or receiving data.

Parameters:

- None

Process:

- The function causes a delay by randomly pausing the execution for up to 5 seconds using time.sleep(), where the time is chosen randomly from 0 to 5 seconds.

---

3. xor(a, b)

Purpose:
To perform a bitwise XOR operation between two binary strings.

Parameters:

- a: A binary string.
- b: A binary string of the same length as a.

Process:

- The function compares corresponding bits in the two binary strings a and b. If the bits are the same, the result is '0'; if they are different, the result is '1'.
- The resulting binary string after XOR is returned.

---

4. binary_division(dividend, divisor)

Purpose:
To perform binary division for calculating the remainder (CRC) when dividing the dividend by the divisor in a bitwise manner, similar to polynomial division.

Parameters:

- dividend: A binary string (representing the data with appended zeros).
- divisor: A binary string representing the generator polynomial for CRC.

Process:

- The function performs binary division by comparing the most significant bit of the dividend to the divisor. If the bit is '1',

it XORs the current section of the dividend with the divisor, shifting to the next bit after each operation.

- This continues until the end of the dividend is reached, leaving the remainder, which is returned as the result.

---

5. crc4itu(frame)

Purpose:
To compute the 4-bit Cyclic Redundancy Check (CRC-4) using the ITU standard for error detection.

Parameters:

- frame: A binary string representing the data to be transmitted.

Process:

- The function appends four zeros ('0000') to the input frame to make space for the CRC bits.
- It then performs binary division using a fixed CRC-4 generator polynomial (10011) via the binary_division function.
- The remainder from this division (which is the CRC) is returned.

---

6. makeFrame(n, data)

Purpose:
To create a communication frame that includes a frame number, the length of the data, the data itself, and a CRC for error detection.

Parameters:

- n: The frame number (sequence number).
- data: The binary string representing the data payload to be sent in the frame.

Process:

- The function converts the frame number n and the length of data to fixed-width binary strings using zero padding based on the frame format specifications (st.N_SIZE for frame number, st.LENGTH_SIZE for length).
- The data is padded to a fixed size (st.DATA_SIZE).
- The CRC is computed for the data using the crc4itu() function.
- The frame is then constructed by concatenating the frame number, length, padded data, and CRC, which is returned.

---

7. receiveFrame(frame)

Purpose:
To extract and interpret the components of a received frame, such as the frame number, data length, the actual data, and the CRC.

Parameters:

- frame: A binary string representing the complete frame received.

Process:

- The frame number is extracted from the first part of the frame (st.N_SIZE bits).
- The length of the data is extracted next (st.LENGTH_SIZE bits).
- Using the length, the data is extracted from the frame (excluding the CRC).
- The CRC is also extracted from the frame.
- If the data is a special termination signal ('q'), the CRC is set to an empty string.
- The function returns the extracted frame number (n), data length (l), data, and CRC.

Sender.ipynb

```
import socket, threading, time, Utils as ut, Statistics as st


sf, sn, sw = 0, 0, 4  # sf = send frame, sn = next frame, sw = sliding window (0-based index)
text, data_queue = [], [None] * sw
flag, flag2, length = False, False, 0


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


def send(conn):
    global sn, sf, text, flag


    while sn - sf < sw and text:
        data = text.pop(0)
        frame = ut.noisychannel(ut.makeFrame(sn, data))
        print(f"[SENDING] Frame: {sn}, {frame}")
        conn.send(frame.encode())
        data_queue[sn % sw] = data
```

```
            sn += 1

        if not text:
            print("[FINISHED] All output read.")
            flag = True

    def recv_Ack(conn):
        global sf, sn, sw, data_queue, flag2

        while True:
            ack_frame = conn.recv(20).decode()
            ack_no, _, data, _ = ut.receiveFrame(ack_frame)

            if data == '00001111' and sf <= ack_no <= sn:  # NAK
                print(f"[NAK RECEIVED] NAK {ack_no}")
                resend_frame(conn, ack_no)

            elif data == '11110000' and sf <= ack_no <= sn:  # ACK
                print(f"[ACK RECEIVED] ACK {ack_no}")
                while sf <= ack_no:
                    data_queue[sf % sw] = None
                    sf += 1
                if ack_no == 7:  # Termination check
                    break

    def resend_frame(conn, ack_no):
        frame = ut.makeFrame(ack_no, data_queue[ack_no % sw])
        print(f"[RESENDING] Frame: {ack_no}, {frame}")
        conn.send(frame.encode())

    def start():
        global text, length

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

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

    with open("data.txt", "r") as f:
        text = [line.strip() for line in f.readlines()]
        length = len(text)

    threading.Thread(target=send, args=(conn,)).start()
    threading.Thread(target=recv_Ack, args=(conn,)).start()

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

start()
```

Explanation

1. send(conn)

Purpose:
To send frames of data from the text list to the receiver while maintaining the sliding window protocol.

Parameters:

- conn: The socket connection object to send frames over.

Process:

- The function checks whether the number of sent frames (sn - sf) is within the sliding window size (sw) and if there is still data left to send.
- The next data from the text list is popped and wrapped into a frame using ut.makeFrame(), followed by the introduction of noise using ut.noisychannel().
- The constructed frame is sent over the connection, and the corresponding data is stored in the data_queue.
- The sequence number (sn) is incremented after each frame.
- If there is no more data left to send, the function sets the flag to True and terminates.

---

2. recv_Ack(conn)

Purpose:
To receive acknowledgments (ACK) and negative acknowledgments (NAK) from the receiver, and handle them appropriately by sliding the window forward or resending frames if necessary.

Parameters:

- conn: The socket connection object to receive acknowledgments from.

Process:

- The function listens for acknowledgment frames from the receiver.
- It decodes each received frame using ut.receiveFrame() to extract the sequence number (ack_no), data type, and CRC.
- If the frame contains a NAK ('00001111'), it calls resend_frame() to resend the frame corresponding to ack_no.
- If the frame contains an ACK ('11110000'), it slides the window forward by incrementing sf and clearing the sent data from the data_queue for every frame from sf to ack_no.
- The function terminates once it receives an ACK for frame 7 (used as an indication of completion).

---

3. resend_frame(conn, ack_no)

Purpose:
To resend a frame in case a negative acknowledgment (NAK) is received.

Parameters:

- conn: The socket connection object to resend the frame over.
- ack_no: The sequence number of the frame that needs to be resent.

Process:

- The function retrieves the data corresponding to the frame number ack_no from the data_queue.
- It rebuilds the frame using ut.makeFrame() and sends the frame over the connection.
- A message is printed to indicate that the frame has been resent.

---

4. start()

Purpose:
To initiate the server-side operations, including accepting connections, reading data from the file, and starting the sending and acknowledgment-receiving threads.

Parameters:

- None

Process:

- The function starts by setting the server to listen for incoming connections on the address defined in st.ADDR.
- Once a connection is established, it reads the data from the "data.txt" file into the text list.
- The length of the data (number of lines) is stored in length.
- It starts two separate threads: one to handle sending frames (send()) and the other to handle receiving acknowledgments (recv_Ack()).
- After the operation is complete, it closes the connection and prints a closing message.

General Explanation:

- This code uses a **Sliding Window Protocol** for communication between sender and receiver. It ensures efficient data transmission by sending multiple frames before waiting for acknowledgments.
- **NAKs** indicate that a specific frame was corrupted, prompting a retransmission.
- **ACKs** acknowledge correct receipt of frames, allowing the sliding window to move forward.

Receiver.ipynb

```
import socket, threading, Utils as ut, Statistics as st


rf, rn, rw = 0, 0, 4  # Frame-related variables
receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
receiver.connect(st.ADDR)


def isCorrupted(data, crc):
    return ut.crc4itu(data) != crc


def recv():
    global rn


    while True:
        try:
            frame = receiver.recv(20).decode()
            recv_n, _, data, crc = ut.receiveFrame(frame)
```

```
        except ValueError:
            break  # Terminate thread when no more data is received

        print(f"[RECV] Frame: {recv_n}, Data: {data}")

        if isCorrupted(data, crc):
            print(f"[CRC FAILURE] Calculated: {ut.crc4itu(data)},
Received: {crc}")
            send_frame(recv_n, "00001111", "NAK")
        else:
            print(f"[CRC SUCCESS] Calculated: {ut.crc4itu(data)},
Received: {crc}")
            send_frame(recv_n, "11110000", "ACK")

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

def send_frame(seq, msg, frame_type):
    frame = ut.makeFrame(seq, msg)
    receiver.send(frame.encode())
    print(f"[{frame_type} SENT] Frame: {seq}")

receiver_thread = threading.Thread(target=recv)
receiver_thread.start()
receiver_thread.join()

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

Explanation

1. isCorrupted(data, crc)

Purpose:
To check whether the received data has been corrupted during transmission

by comparing its calculated CRC (Cyclic Redundancy Check) value with the received CRC.

Parameters:

- data: The data part of the received frame that needs to be checked for corruption.
- crc: The CRC value received along with the data, which is supposed to match the calculated CRC of the data.

Process:

- The function uses ut.crc4itu() to calculate the CRC for the given data.
- It returns True if the calculated CRC does not match the received crc (indicating corruption), and False if they match (no corruption).

---

2. recv()

Purpose:
To receive frames from the sender, check for corruption, and send an acknowledgment (ACK) or negative acknowledgment (NAK) accordingly.

Parameters:

- None

Process:

- The function continuously listens for incoming frames using receiver.recv() and decodes them into a readable format.
- It extracts the sequence number (recv_n), data, and CRC using ut.receiveFrame().
- If the frame data is corrupted (determined by calling isCorrupted()), it sends a NAK using send_frame() and prints a CRC failure message.
- If the data is not corrupted, it sends an ACK using send_frame() and prints a success message.
- The loop continues until a ValueError occurs (which happens when no more data is received), at which point the function terminates.

---

3. send_frame(seq, msg, frame_type)

Purpose:
To send a frame (ACK or NAK) to the sender, indicating whether the previous frame was received correctly or was corrupted.

Parameters:

- seq: The sequence number of the frame being acknowledged or rejected.

- msg: The message to send as the frame content, either "11110000" for an ACK or "00001111" for a NAK.
- frame_type: A string indicating the type of frame being sent ("ACK" or "NAK"), used for logging.

Process:

- The function creates a new frame using ut.makeFrame() by combining the sequence number and the message (msg).
- It sends the constructed frame to the sender using receiver.send().
- A log message is printed to indicate whether an ACK or NAK was sent, along with the sequence number.

General Explanation:

- The **receiver** is responsible for listening to incoming frames from the sender, verifying their integrity using CRC checks, and responding with an ACK (if the frame is valid) or a NAK (if the frame is corrupted).
- The function isCorrupted() handles the error detection, while recv() manages the reception of frames and calls send_frame() to send appropriate responses based on the result of the CRC check

OUTPUTS:

Stop and Wait:

Sender:

```
[LISTENING] Server is listening on 172.24.128.1
[CONNECTED] Connected to Process Id: ('172.24.128.1', 57439)
[INPUT] Enter data to send: 100111
[ENCODING] Encoded frame: 00006000001001110011
[NOISY] Frame after noise: 00006011000101000011
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00006011100100010011
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 00006100001001110011
[Reciving ACK]......
[ACK RECV] ACK 0 successfully received.
[TRANSACTION COMPLETED]
-----------------------------------------------------
[INPUT] Enter data to send: 101110
[ENCODING] Encoded frame: 01006000001011101011
[NOISY] Frame after noise: 01006000001010001011
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 01006010110000011011
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 01006101000101111011
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 01006000001011101011
[Reciving ACK]......
[ACK RECV] ACK 1 successfully received.
[TRANSACTION COMPLETED]
-----------------------------------------------------
[INPUT] Enter data to send: 10110
[ENCODING] Encoded frame: 02005000000101101111
[NOISY] Frame after noise: 02005110111001101111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005110111001001111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005001001001101111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005111101010001111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005000010111111111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005100011000101111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005000000110101111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005111111010001111
[Reciving ACK]......
---[TIMEOUT OCCURED]----
[RESENDING] Resending frame: 02005010111101101111
[Reciving ACK]......
[ACK RECV] ACK 2 successfully received.
[TRANSACTION COMPLETED]
-----------------------------------------------------
```

Receiver:

```
[LISTENING] Receiver is listening....
[RECV] Received message: 010100
[CRC FAILURE] 1001 and 0011
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 010001
[CRC FAILURE] 0110 and 0011
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 100111
[CRC SUCCESS] 0011 and 0011
[ACK SENT] Sent ACK
[TRANSACTION COMPLETED]
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 101000
[CRC FAILURE] 0001 and 1011
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 000001
[CRC FAILURE] 0011 and 1011
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 010111
[CRC FAILURE] 1100 and 1011
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 101110
[CRC SUCCESS] 1011 and 1011
[ACK SENT] Sent ACK
[TRANSACTION COMPLETED]
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 00110
[CRC FAILURE] 1010 and 1111
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 00100
[CRC FAILURE] 1100 and 1111
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 00110
[CRC FAILURE] 1010 and 1111
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 01000
[CRC FAILURE] 1011 and 1111
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 11111
[CRC FAILURE] 0111 and 1111
--------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 00010
```

```
[CRC FAILURE] 0110 and 1111
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 11010
[CRC FAILURE] 1000 and 1111
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 01010
[CRC FAILURE] 1101 and 1111
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 01000
[CRC FAILURE] 1011 and 1111
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 10110
[CRC SUCCESS] 1111 and 1111
[ACK SENT] Sent ACK
[TRANSACTION COMPLETED]
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 101100
[CRC FAILURE] 1101 and 1010
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 111110
[CRC FAILURE] 1110 and 1010
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 100001
[CRC FAILURE] 1001 and 1010
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 101111
[CRC FAILURE] 1000 and 1010
-------------------------------------------------
[LISTENING] Receiver is listening....
[RECV] Received message: 100000
[CRC SUCCESS] 1010 and 1010
[ACK SENT] Sent ACK
[TRANSACTION COMPLETED]
-------------------------------------------------
```

Go Back N ARQ:

Sender:

```
[LISTENING] Server is listening on 172.24.128.1
[CONNECTED] Connected to Process Id: ('172.24.128.1', 56632)
[SENDING] Sending frame: 00006101110100101110
[SENDING] Sending frame: 01006001001100010000
[SENDING] Sending frame: 02006111001000111110
[SENDING] Sending frame: 03006011101111110101
[RE SENDING] Resending frame: 00006000001011011110
[RE SENDING] Resending frame: 01006000001101010000
[RE SENDING] Resending frame: 02006000001111101110
[RE SENDING] Resending frame: 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: 04006101100011111111
[SENDING] Sending frame: 05006000001101100100
[SENDING] Sending frame: 06006001001010011100
[SENDING] Sending frame: 07006000001000001010
[RE SENDING] Resending frame: 04006000001100001111
[RE SENDING] Resending frame: 05006000001011010100
[RE SENDING] Resending frame: 06006000001100011100
[RE SENDING] Resending frame: 07006000001000001010
[ACK RECV] ACK 4 successfully received.
[FINISHED] All output read.
[ACK RECV] ACK 5 successfully received.
[ACK RECV] ACK 6 successfully received.
[ACK RECV] ACK 7 successfully received.
[FINISHED] All ACK received.
[CLOSING] Closing sender...
```

Receiver:

```
[RECV] Received message: 0, 010010
[CRC FAILURE] 0011 and 1110
[RECV] Received message: 1, 110001
[CRC FAILURE] 1100 and 0000
[RECV] Received message: 2, 100011
[CRC FAILURE] 1111 and 1110
[RECV] Received message: 3, 111111
[CRC FAILURE] 1101 and 0101
[RECV] Received message: 0, 101101
[CRC SUCCESS] 1110 and 1110
[ACK SENT] Sent ACK
[RECV] Received message: 1, 110101
[CRC SUCCESS] 0000 and 0000
[ACK SENT] Sent ACK
[RECV] Received message: 2, 111110
[CRC SUCCESS] 1110 and 1110
[ACK SENT] Sent ACK
[RECV] Received message: 3, 100101
[CRC SUCCESS] 0101 and 0101
[ACK SENT] Sent ACK
[RECV] Received message: 4, 001111
[CRC FAILURE] 0010 and 1111
[RECV] Received message: 5, 110110
[CRC FAILURE] 0101 and 0100
[RECV] Received message: 6, 101001
[CRC FAILURE] 0010 and 1100
[RECV] Received message: 7, 100000
[CRC FAILURE] 1010 and 1010
[RECV] Received message: 4, 110000
[CRC SUCCESS] 1111 and 1111
[ACK SENT] Sent ACK
[RECV] Received message: 5, 101011
[CRC SUCCESS] 0100 and 0100
[ACK SENT] Sent ACK
[RECV] Received message: 6, 110001
[CRC SUCCESS] 1100 and 1100
[ACK SENT] Sent ACK
[RECV] Received message: 7, 100000
[CRC SUCCESS] 1010 and 1010
[ACK SENT] Sent ACK
[CLOSING] Closing receiver....
```

Selective Repeat ARQ:

Sender:

```
C:\Users\sudip\PycharmProjects\pythonProject\venv\Scripts\python.exe "C:\Users\sudip\PycharmProjects\pythonProject1\My Python Programs\DSA and OOPs and Machine Learning\assig
[LISTENING] Server is listening on 172.24.128.1
[CONNECTED] Connected to Process Id: ('172.24.128.1', 57266)
[SENDING] Sending frame: no. 0, 00006000001011011110
[SENDING] Sending frame: no. 1, 01006101111111010000
[SENDING] Sending frame: no. 2, 02006111101110111110
[ACK RECV] ACK 0 successfully received.
[NAK RECV] NAK 1 successfully received.[SENDING] Sending frame: no. 3, 03006100100110100101

[SENDING] Sending frame: no. 4, 04006111100011111111[RE SENDING] Resending frame: 1, 01006000001101010000

[NAK RECV] NAK 2 successfully received.
[RE SENDING] Resending frame: 2, 02006000001111101110
[NAK RECV] NAK 3 successfully received.
[RE SENDING] Resending frame: 3, 03006000001001010101
[ACK RECV] ACK 1 successfully received.
[NAK RECV] NAK 4 successfully received.
[SENDING] Sending frame: no. 5, 05006101100000000100
[RE SENDING] Resending frame: 4, 04006000001100001111
[ACK RECV] ACK 2 successfully received.
[SENDING] Sending frame: no. 6, 06006010000100011100[ACK RECV] ACK 3 successfully received.

[NAK RECV] NAK 5 successfully received.
[SENDING] Sending frame: no. 7, 07006101001010111010
[RE SENDING] Resending frame: 5, 05006000001010110100
[ACK RECV] ACK 4 successfully received.
[FINISHED] All output read.
[NAK RECV] NAK 6 successfully received.
[RE SENDING] Resending frame: 6, 06006000001100011100
[NAK RECV] NAK 7 successfully received.
[RE SENDING] Resending frame: 7, 07006000001000001010
[ACK RECV] ACK 5 successfully received.
[ACK RECV] ACK 6 successfully received.
[ACK RECV] ACK 7 successfully received.
[CLOSING] Closing sender...
```

Receiver:

```
[RECV] Received message: 0, 101101
[CRC SUCESS] 1110 and 1110
[ACK SENT] Sent ACK 0
------------------------------------------------
[RECV] Received message: 1, 111101
[CRC FAILURE] 1011 and 0000
[NAK SENT] Sent NAK 1
------------------------------------------------
[RECV] Received message: 2, 111011
[CRC FAILURE] 0001 and 1110
[NAK SENT] Sent NAK 2
------------------------------------------------
[RECV] Received message: 3, 011010
[CRC FAILURE] 1000 and 0101
[NAK SENT] Sent NAK 3
------------------------------------------------
[RECV] Received message: 1, 110101
[CRC SUCESS] 0000 and 0000
[ACK SENT] Sent ACK 1
------------------------------------------------
[RECV] Received message: 4, 001111
[CRC FAILURE] 0010 and 1111
[NAK SENT] Sent NAK 4
------------------------------------------------
[RECV] Received message: 2, 111110
[CRC SUCESS] 1110 and 1110
[ACK SENT] Sent ACK 2
[RECV] Received message: 3, 100101
[CRC SUCESS] 0101 and 0101
[ACK SENT] Sent ACK 3
------------------------------------------------
[RECV] Received message: 5, 000000
[CRC FAILURE] 0000 and 0100
[NAK SENT] Sent NAK 5
------------------------------------------------
[RECV] Received message: 4, 110000
[CRC SUCESS] 1111 and 1111
[ACK SENT] Sent ACK 4
------------------------------------------------
[RECV] Received message: 6, 001000
[CRC FAILURE] 1011 and 1100
[NAK SENT] Sent NAK 6
------------------------------------------------
[RECV] Received message: 7, 101011
[CRC FAILURE] 0100 and 1010
[NAK SENT] Sent NAK 7
------------------------------------------------
[RECV] Received message: 5, 101011
[CRC SUCESS] 0100 and 0100
[ACK SENT] Sent ACK 5
------------------------------------------------
[RECV] Received message: 6, 110001
[CRC SUCESS] 1100 and 1100
[ACK SENT] Sent ACK 6
------------------------------------------------
------------------------------------------------
[RECV] Received message: 7, 100000
[CRC SUCESS] 1010 and 1010
[ACK SENT] Sent ACK 7
------------------------------------------------
[CLOSING] Closing receiver....
```

RESULTS (TEST CASES)

Table 1: showing the propagation time and ack reception time:

Protocol	**Propagation Time(ms)**	**ACK reception Time(ms)**	**Total round Trip time(ms)**
Stop and Wait ARQ			
Go Back N ARQ			
Selective Repeat ARQ			

Table 2: showing Efficiency without error or lost frame:

Protocol	Efficiency(%)
Stop and Wait ARQ	
Go Back N ARQ	
Selective Repeat ARQ	

Table 3: showing Efficiency with 0.1-0.5 probability of error or lost frame:

Protocol	Efficiency(%)
Stop and Wait ARQ	
Go Back N ARQ	
Selective Repeat ARQ	

ANALYSIS

In our comparative analysis of flow control protocols, the differences in performance are clear when measured in terms of throughput efficiency and average transmission time. **Selective Repeat** stands out as the champion, outperforming both **Go Back N** and the **primitive Stop and Wait** protocol. The use of **sliding window** techniques showcases their superiority, significantly enhancing efficiency over the more basic Stop and Wait approach.

While examining **Round Trip Time (RTT)**, which measures the total time taken for a packet to traverse to the receiver and back, it's important to note that RTT is fundamentally a property of the network, not the protocol itself. Though evaluating RTT on a noiseless channel can provide insights, it shouldn't be our primary metric for protocol performance. Additionally, we've opted to overlook processing and transmission times in our RTT calculations, even though they can play a substantial role in overall performance. The **Bandwidth-Delay Product** also falls into this category, being a channel characteristic rather than a protocol attribute.

Instead of employing multiprocessing, we've leveraged **multithreading** to streamline communication through a single port—62000 in this case. This decision was made to address the challenges of establishing seamless communication across multiple ports, ensuring that all threads can effectively interact without unnecessary complications.

COMMENTS

This assignment has greatly enhanced my understanding of different automatics repeat request(ARQ) techniques through both theoretical study and hands-on application. By delving into these methods, I have gained a deeper appreciation for their benefits and drawbacks. Furthermore, I have learned how the limitations of one approach can be mitigated by leveraging alternative strategies.

I would like to express my heartfelt thanks to my teachers Dr Sarbani Roy and Dr Nandini Mukherjee for their guidance

and support throughout this journey. Their encouragement has played a key role in helping me better comprehend these concepts.