Computer Networks Assignment 2

BCSE III

Name: Mourya Saha

Roll: 002310501036

Group: A2

Problem Statement:

Implementation and analysis of reliable data transfer protocols (Go-Back-N/Stop-and-Wait and Selective Repeat) and error detection using CRC.

Design:

Purpose of the Program

This project simulates two reliable data transfer protocols: a basic sliding window protocol (likely Go-Back-N or Stop-and-Wait) and Selective Repeat. It also includes error detection mechanisms using CRC (Cyclic Redundancy Check) and a simple checksum. The purpose is to send data from a sender to a receiver over an unreliable channel, which is simulated by introducing errors and packet loss. The project allows for a comparative analysis of the two protocols in terms of their efficiency and reliability in handling these errors.

Structure Diagram

- The project is structured into several modules:
    - **Sender (\`sender.py\`, \`sender_sr.py\`):** Reads data from a file, creates frames with headers (source/destination address, sequence number) and a CRC-32 checksum, and sends them to the receiver. It manages a sending window, timers, and retransmissions based on acknowledgements (ACKs) or timeouts.
    - **Receiver (\`reciever.py\`, \`reciever_sr.py\`):** Receives frames from the sender, verifies the CRC-32 checksum, and sends ACKs for correctly received frames. The Selective Repeat receiver can buffer out-of-order frames and send negative acknowledgements (NAKs).
    - **Error Detection (\`crc.py\`):** Provides functions to generate and verify CRC. \`crc.py\` is used in the main sender/receiver loops for robust error detection.
    - **Error Injection (\`injecterror.py\`):** A testing module to simulate an unreliable channel by injecting various types of errors (single bit, burst, odd) into the data frames.
    - **Data Files (\`data.txt\`, \`test_data.txt\`):** Text files containing the binary data to be transmitted.
- **Input and Output Format:**
    - **Input:** The input is a text file (\`test_data.txt\`) containing a binary string. The sender reads this data, and the user can configure parameters like window size and timeout values.
    - **Output:** The sender and receiver print status messages to the console, showing the frames being sent, received, acknowledged, and retransmitted. The receiver, upon successful reception, would have the complete, error-free data.

Implementation

(Stop-and-Wait/ Go-Back-N ARQ)

sender.py

- The sender sends a window of frames and waits for an ACK for the base of the window.
- If an ACK is received, the window slides forward.
- If a timeout occurs, the sender retransmits all frames from the last acknowledged frame onwards.
- Window size can be mentioned when calling send(n)

```
import socket
import threading
import time
import random
import injecterror
import crc

HOST = '127.0.0.1'
PORT = 3000

buffer = {}
next_frame = 0
recieved_ack = -1
lock = threading.Lock()
file_complete = False

timer_running = False
timer = None
ack_buffer = ""   # <-- buffer for partial/multiple ACKs

SRC_ADDR = "000000010000000100000001000000010000000100000001"
DEST_ADDR = "000000010000000100000001000000010000000100000010"
CODEWORD_SIZE = 64*8
HEADER_SIZE = 12*8 + 16
TAILER_SIZE = 4*8
PAYLOAD_SIZE = CODEWORD_SIZE - (HEADER_SIZE + TAILER_SIZE)

def start_timer():
    global timer, timer_running
    if not timer_running:
        timer_running = True
        timer = threading.Timer(5.0, timeout_handler)
        timer.start()

def stop_timer():
    global timer, timer_running
    if timer is not None:
        timer.cancel()
    timer_running = False
```

```
def timeout_handler():
    global next_frame, recieved_ack
    print("Timeout! Resending frames...")
    with lock:
        for f in range(recieved_ack+1, next_frame):
            frame = buffer[f]
            sock.send(f"{frame}\n".encode("utf-8"))
            print(f"Resent: {f}")
    start_timer()

def sender(n):
    global next_frame, buffer, file_complete, timer_running
    while True:
        with lock:
            if next_frame - recieved_ack < n:

                # prepping data
                data = f.read(PAYLOAD_SIZE)
                if not data:
                    print("End of file reached")
                    file_complete = True
                    return
                data = data + ('0'*(PAYLOAD_SIZE - len(data)))

                frame_no = bin(next_frame)[2:]
                payload = SRC_ADDR + DEST_ADDR + ('0'*(16-len(frame_no)) + frame_no) +
data
                tailer = crc.generate_crc(payload, 'CRC-32')

                frame = payload + ('0'*(TAILER_SIZE - len(tailer)) + tailer)

                buffer[next_frame] = frame
                if random.random() < 0.1:
                    frame = injecterror.injectodderror(frame)

                sock.send(f"{frame}\n".encode("utf-8"))
                print(f"Frame Sent: {next_frame}")
                if not timer_running:
                    start_timer()
                next_frame += 1
        # time.sleep(1)

def acknowledge():
    global recieved_ack, ack_buffer, file_complete
    while True:
        data = sock.recv(1024)
        if not data:
            break

        ack_buffer += data.decode("utf-8")

        # process all complete ACKs separated by newline
        while "\n" in ack_buffer:
            msg, ack_buffer = ack_buffer.split("\n", 1)
```

```python
            if not msg:
                continue
            try:
                ack_type, ack_no = msg.split(":", 1)
                ack_no = int(ack_no)
                with lock:
                    if ack_no > recieved_ack:
                        for i in range(recieved_ack+1, ack_no+1):
                            if i in buffer:
                                del buffer[i]
                        recieved_ack = ack_no
                        print(f"Acknowledged: frame {ack_no}")

                        # stop timer if all outstanding frames acked
                        if recieved_ack == next_frame-1:
                            if file_complete:
                                return
                            stop_timer()
                        else:
                            # restart timer for next unacked frame
                            stop_timer()
                            start_timer()
            except ValueError:
                print(f"Corrupt ACK ignored: {msg}")

def send(n):
    threading.Thread(target=sender, args=(n,), daemon=True).start()
    threading.Thread(target=acknowledge, daemon=True).start()

with open('test_data.txt', 'r') as f, socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    while True:
        try:
            print(f"Attempting to connect to {HOST}:{PORT}...")
            sock.connect((HOST, PORT))
            break
        except ConnectionRefusedError:
            print("Connection refused, retrying in 2 seconds...")
            time.sleep(2)
    print(f"Connected to {HOST}:{PORT}")
    send(4)

    while True:
        if file_complete and recieved_ack == next_frame-1:
            sock.close()
            print("Transfer complete. Closing program.")
            break
        time.sleep(1)
```

reciever.py

The receiver only accepts frames in the expected order and discards any out-of-order frames.

```python
import socket
import random
import time
import crc

HOST = '127.0.0.1'
PORT = 3000

CODEWORD_SIZE = 64*8
HEADER_SIZE = 12*8 + 16
TAILER_SIZE = 4*8
PAYLOAD_SIZE = CODEWORD_SIZE - (HEADER_SIZE + TAILER_SIZE)

def receiver():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"Server listening on {HOST}:{PORT}")

        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            next_frame = 0
            buffer = ""

            while True:
                data = conn.recv(1024)
                if not data:
                    break

                buffer += data.decode("utf-8")

                # process all complete frames separated by newline
                while "\n" in buffer:
                    msg, buffer = buffer.split("\n", 1)
                    if not msg:
                        continue

                    try:
                        # frame_no_str, frame_payload = msg.split(":", 1)
                        # frame_no = int(frame_no_str)

                        header = msg[:HEADER_SIZE]
                        src = header[:6*8]
                        dest = header[6*8:12*8]
                        frame_no = int(msg[12*8: HEADER_SIZE], 2)
                        payload = msg[HEADER_SIZE: HEADER_SIZE+PAYLOAD_SIZE]
                        tail = msg[CODEWORD_SIZE - TAILER_SIZE:]
                        check_tail = crc.generate_crc(header + payload, 'CRC-32')
                        if tail != check_tail:
                            print(f"Corrupted frame recieved: frame {frame_no} ignored")
                            continue


                        # time.sleep(random.uniform(1, 6))  # artificial delay

                        if frame_no == next_frame:
                            print(f"Frame received: {frame_no}")
                            if random.random() < 0.1:
                                print("Simulated ACK loss")
                            else:
                                conn.send(f"ack:{next_frame}\n".encode("utf-8"))
                            next_frame += 1
                        else:
                            # duplicate ACK for last in-order frame
                            if random.random() < 0.1:
                                print("Simulated ACK loss")
                            else:
                                conn.send(f"ack:{next_frame-1}\n".encode("utf-8"))

                    except ValueError:
                        print(f"Corrupted/partial frame ignored: {msg}")
            print("Transfer complete. Connection closed.")

receiver()
```

(Selective Repeat)

sender_sr.py

- The sender sends a window of frames, each with its own timer.
- The sender only retransmits the specific frame that was lost or for which a NAK was received.
- Window size in set in a global variable

```
import socket, threading, time, random, crc, injecterror

HOST = "127.0.0.1"
PORT = 3000
WINDOW_SIZE = 4
TIMEOUT = 5
LOSS_PROB = 0.2   # 20% chance to "lose" a frame

SRC_ADDR = "000000010000000100000001000000010000000100000001"
DEST_ADDR = "000000100000001000000010000000100000001000000010"
CODEWORD_SIZE = 64*8
HEADER_SIZE = 12*8 + 16
TAILER_SIZE = 4*8
PAYLOAD_SIZE = CODEWORD_SIZE - (HEADER_SIZE + TAILER_SIZE)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
while True:
    try:
        print(f"Attempting to connect to {HOST}:{PORT}...")
        sock.connect((HOST, PORT))
        break
    except ConnectionRefusedError:
        print("Connection refused, retrying in 2 seconds...")
        time.sleep(2)
print(f"Connected to {HOST}:{PORT}")

buffer = {}       # frame_no -> data
timers = {}       # frame_no -> Timer
next_frame = 0
base = 0
lock = threading.Lock()
ack_buffer = ""
file_complete = False

def timeout_handler(frame_no):
    with lock:
        if frame_no in buffer:
            frame = buffer[frame_no]

            sock.send(f"{frame}\n".encode())
            print(f"Timeout! Resent frame {frame_no}")
            # restart timer
            t = threading.Timer(TIMEOUT, timeout_handler, args=(frame_no,))
            timers[frame_no] = t
            t.start()

def sender():
    global next_frame, file_complete, buffer
    with open('test_data.txt', 'r') as f:
        while True:
            with lock:
                if next_frame < base + WINDOW_SIZE:   # within window
                    # prepping data
                    data = f.read(PAYLOAD_SIZE)
                    if not data:
                        print("End of file reached")
                        file_complete = True
                        return
                    data = data + ('0'*(PAYLOAD_SIZE - len(data)))
                    frame_no = bin(next_frame)[2:]
                    payload = SRC_ADDR + DEST_ADDR + ('0'*(16-len(frame_no)) + frame_no) +
data

                    tail = crc.generate_crc(payload, 'CRC-32')

                    frame = payload + ('0'*(TAILER_SIZE - len(tail)) + tail)
                    buffer[next_frame] = frame
                    if random.random() < 0.4:
                        frame = injecterror.injectodderror(frame)

                    if random.random() < LOSS_PROB:
                        print(f"Simulated loss: frame {next_frame} not sent")
                    else:
                        sock.send(f"{frame}\n".encode())
                        print(f"Sent: {next_frame}")
                    # start per-frame timer
                    t = threading.Timer(TIMEOUT, timeout_handler, args=(next_frame,))
                    timers[next_frame] = t
                    t.start()
                    next_frame += 1
            time.sleep(1)

def acknowledge():
    global base, ack_buffer
    while True:
        data = sock.recv(1024)
        if not data: break
        ack_buffer += data.decode()
        while "\n" in ack_buffer:
            msg, ack_buffer = ack_buffer.split("\n", 1)
            if not msg: continue
            try:
                type, ack_no = msg.split(":", 1)
                ack_no = int(ack_no)

                # Simulate lost ACK reception
                if random.random() < LOSS_PROB:
                    print(f"Simulated ACK loss: ack {ack_no} ignored")
                    continue
                with lock:
                    if type == "ack":
                        if ack_no in buffer:
                            buffer.pop(ack_no)
                            if ack_no in timers:
                                timers[ack_no].cancel()
                                timers.pop(ack_no)
                            print(f"ACK {ack_no} received -> frame removed")
                            if file_complete and not buffer:
                                return

                            # slide base forward if possible
                            while base not in buffer and base < next_frame:
                                base += 1
                    elif type == "nak":
                        if ack_no in timers and ack_no in buffer:
                            timers[ack_no].cancel()
                            timers.pop(ack_no)
                            frame = buffer[ack_no]
                            sock.send(f"{frame}\n".encode())
                            print(f"NAK! Resent frame {ack_no}")

            except ValueError:
                print(f"Bad ACK ignored: {msg}")

def run():
    threading.Thread(target=sender, daemon=True).start()
    threading.Thread(target=acknowledge, daemon=True).start()

run()
while True:
    if file_complete and not buffer:
        print("Transfer complete. Closing program.")
        sock.close()
        break
    time.sleep(1)
```

receiver_sr.py

- The receiver ACKs each correctly received frame, even if it's out of order, and buffers it.
- If a frame is missing, the receiver sends a NAK.

```
import socket, time, random, crc

HOST = "127.0.0.1"
PORT = 3000
LOSS_PROB = 0.2   # 20% chance to drop a received frame or ack

CODEWORD_SIZE = 64*8
HEADER_SIZE = 12*8 + 16
TAILER_SIZE = 4*8
PAYLOAD_SIZE = CODEWORD_SIZE - (HEADER_SIZE + TAILER_SIZE)

def receiver():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"Server listening on {HOST}:{PORT}")

        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            expected = 0
            buffer = {}
            data_buf = ""
            last_recieved = -1

            while True:
                data = conn.recv(1024)
                if not data: break
                data_buf += data.decode()

                while "\n" in data_buf:
                    msg, data_buf = data_buf.split("\n", 1)
                    if not msg: continue
                    try:
                        header = msg[:HEADER_SIZE]
                        src = header[:6*8]
                        dest = header[6*8:12*8]
                        fno = int(msg[12*8: HEADER_SIZE], 2)
                        payload = msg[HEADER_SIZE: HEADER_SIZE+PAYLOAD_SIZE]
                        tail = msg[CODEWORD_SIZE - TAILER_SIZE:]
                        time.sleep(1)

                        check_tail = crc.generate_crc(header + payload, 'CRC-32')
                        if tail != check_tail:
                            print(f"Corrupted frame recieved: frame {fno} ignored")
                            continue

                        for no in range(last_recieved+1, fno):
                            if no not in buffer:
                                conn.send(f"nak:{no}\n".encode())
                                print(f"Sent NAK for missing frame {no}")

                        # Simulate frame loss
                        if random.random() < LOSS_PROB:
                            print(f"Simulated frame loss: frame {fno} dropped")
                            continue

                        if fno not in buffer:
                            buffer[fno] = payload
                            print(f"Frame {fno} buffered")
                            last_recieved = fno

                            # Simulate ACK loss
                            if random.random() < LOSS_PROB:
                                print(f"Simulated ACK loss: ack {fno} not sent")
                            else:
                                conn.send(f"ack:{fno}\n".encode())

                            # deliver in-order frames
                            while expected in buffer:
                                print(f"Delivered in order: {expected}")
                                buffer.pop(expected)
                                expected += 1
                    except ValueError:
                        print(f"Corrupted frame ignored: {msg}")

receiver()
```

Test Cases

- **Error-free transmission:** Run the sender and receiver without any error injection to verify that the data is transmitted correctly.
- **Single bit errors:** Use `` `injecterror.injecterror()` `` to introduce single bit flips in the frames. The receiver should detect these errors using the CRC check and discard the frames. The sender should then retransmit the frames.
- **Frame loss:** The sender and receiver scripts include a `LOSS_PROB` variable to simulate frame and ACK loss. This tests the timeout and retransmission mechanisms of the protocols.
- **ACK loss:** Similar to frame loss, the loss of ACK packets is simulated to test the sender's timeout and retransmission logic.
- **Out-of-order delivery:** By introducing delays, the out-of-order arrival of frames can be simulated to test the receiver's buffering mechanism (especially in Selective Repeat).

Results

- **Performance Metrics:**
    - **Throughput:** The rate of successful data transfer (bits per second). This can be calculated by measuring the total data sent and the time taken.
    - **Error Detection Rate:** The percentage of injected errors that are successfully detected by the receiver.
    - **Retransmission Rate:** The number of retransmitted frames as a percentage of the total frames sent.
- **Expected Results:**
    - Selective Repeat is expected to have a higher throughput than Go-Back-N, especially in a high-error-rate environment, because it retransmits only the lost frames, not the entire window.
    - CRC-32 should detect all single-bit errors, all burst errors up to 32 bits, and a very high percentage of other errors.
    - The retransmission rate will increase with the error and loss probability.

Without error or lost frame:

Transfer protocol	Time taken	Throughput	Total frames sent	Total retransmissions	Retransmission Rate
Stop-and-Wait	<1 second	8171.84 bps	23	0	0.00%
Go-Back-N (10)	<1 second	8172.24 bps	23	0	0.00%
Selective Repeat	<1 seconds	8170.89 bps	23	0	0.00%

With error or lost frame:

Error probability: 0.1

Lost frame probability: 0.1

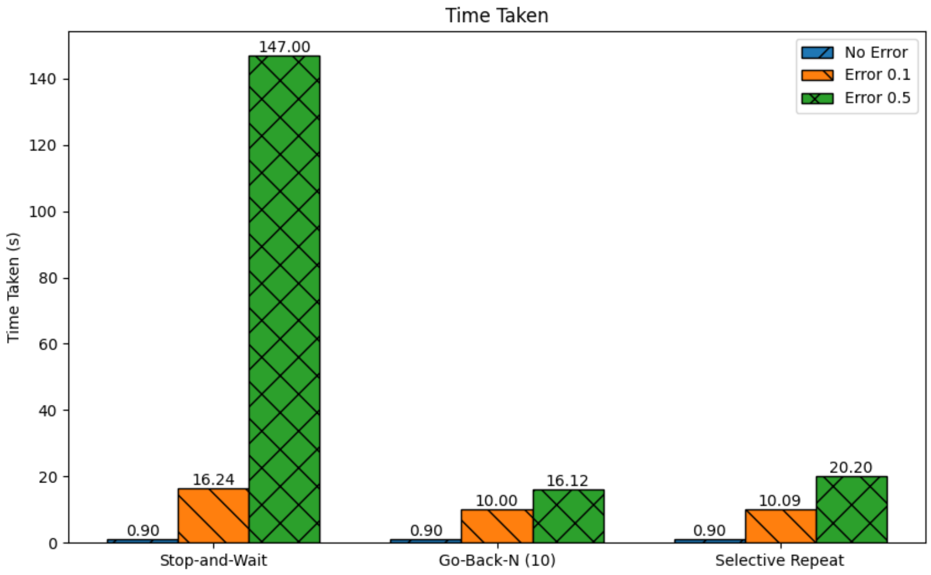
Transfer protocol	Time taken	Throughput	Total frames sent	Total retransmissions	Retransmission Rate
Stop-and-Wait	16.24 seconds	504.58 bps	26	3	11.54%
Go-Back-N (10)	10 second	1364.44 bps	29	6	20.69%
Selective Repeat	10.09 seconds	811.93 bps	26	3	11.54%

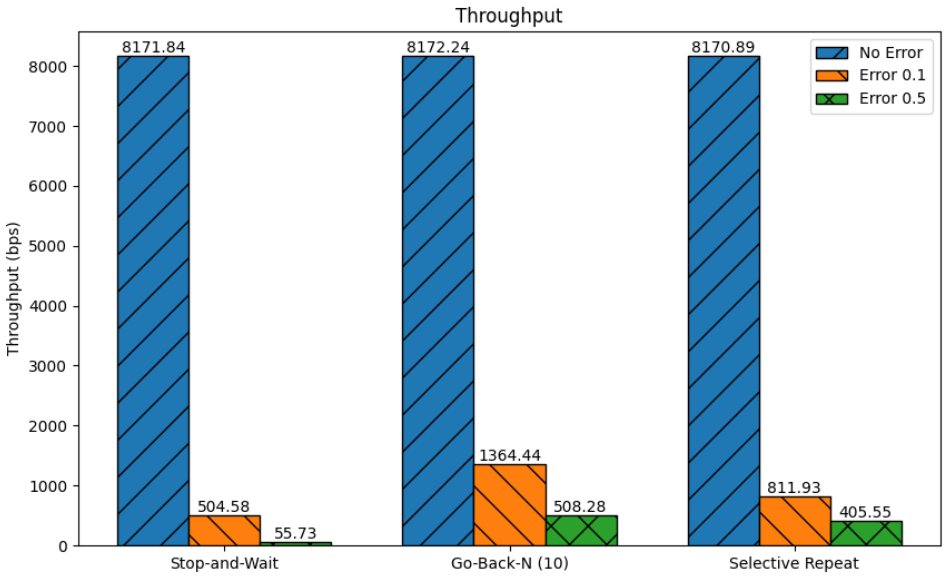
With error or lost frame:

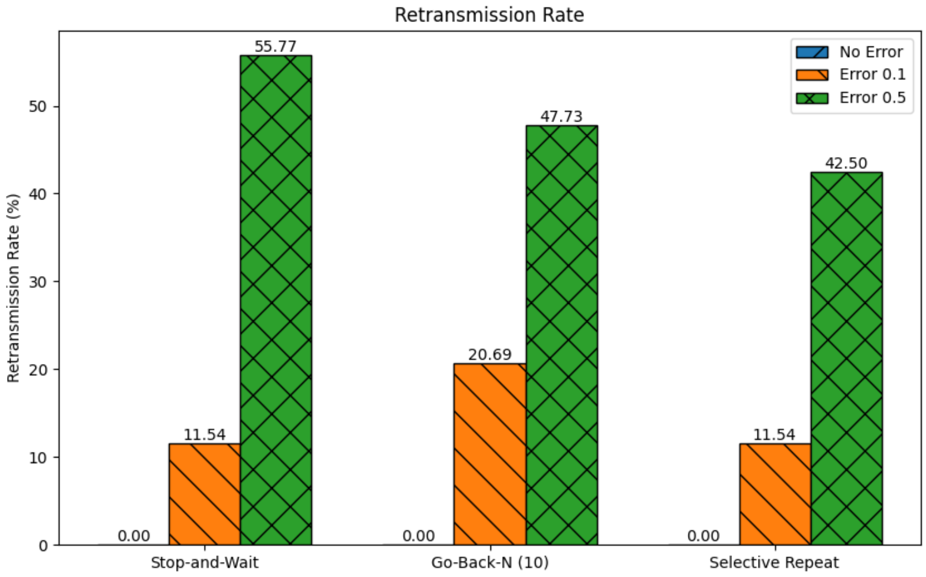
Error probability: 0.5

Lost frame probability: 0.5

Transfer protocol	Time taken	Throughput	Total frames sent	Total retransmissions	Retransmission Rate
Stop-and-Wait	147.00 seconds	55.73 bps	52	29	55.77%
Go-Back-N (10)	16.12 seconds	508.28 bps	44	21	47.73%
Selective Repeat	20.20 seconds	405.55 bps	40	17	42.50%


Time Taken
Time Taken (s)
0
20
40
60
80
100
120
140
0.90
16.24
147.00
0.90
10.00
16.12
0.90
10.09
20.20
Stop-and-Wait
Go-Back-N (10)
Selective Repeat
No Error
Error 0.1
Error 0.5


Throughput
Throughput (bps)
0
1000
2000
3000
4000
5000
6000
7000
8000
8171.84
504.58
55.73
8172.24
1364.44
508.28
8170.89
811.93
405.55
Stop-and-Wait
Go-Back-N (10)
Selective Repeat
No Error
Error 0.1
Error 0.5


Retransmission Rate
Retransmission Rate (%)
0
10
20
30
40
50
0.00
11.54
55.77
0.00
20.69
47.73
0.00
11.54
42.50
Stop-and-Wait
Go-Back-N (10)
Selective Repeat
No Error
Error 0.1
Error 0.5

Analysis

- **Protocol Comparison:**
    - **Go-Back-N/Stop-and-Wait:** Simpler to implement, but inefficient if the bandwidth-delay product is large or the error rate is high. A single lost frame causes the retransmission of many subsequent frames.
    - **Selective Repeat:** More complex to implement due to the need for individual timers and buffering at the receiver. However, it is much more efficient on unreliable links as it minimizes retransmissions.
- **Error Detection:**
    - CRC is a powerful error detection mechanism and is much more robust than a simple checksum. The choice of the polynomial is crucial for its effectiveness. CRC-32 is a standard for Ethernet and other protocols due to its excellent error detection capabilities.

Possible Improvements

- The current implementation uses a fixed timeout value. A dynamic timeout, based on the round-trip time (RTT), would make the protocols more adaptive to network conditions.
- The file reading and sending loop could be made more efficient.

Comments

- **Evaluation of the Lab:**
  This assignment provides a practical understanding of the challenges of reliable data transfer over unreliable networks. It demonstrates the core principles of sliding window protocols and the importance of error detection.
- **What was learned:**
    - The implementation details of Go-Back-N and Selective Repeat protocols.
    - The working of CRC and its superiority over simple checksums.
    - The complexities of handling timeouts, retransmissions, and duplicate packets.
- **Suggestions for Improvements:**
    - Visualize the sending and receiving windows and the movement of frames.
    - Implement a graphical user interface (GUI) to control the simulation parameters and visualize the results.
    - Compare the performance of different CRC polynomials.