|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| **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 = "000000100000001000000010000000100000001000000010"

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)

            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.

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**

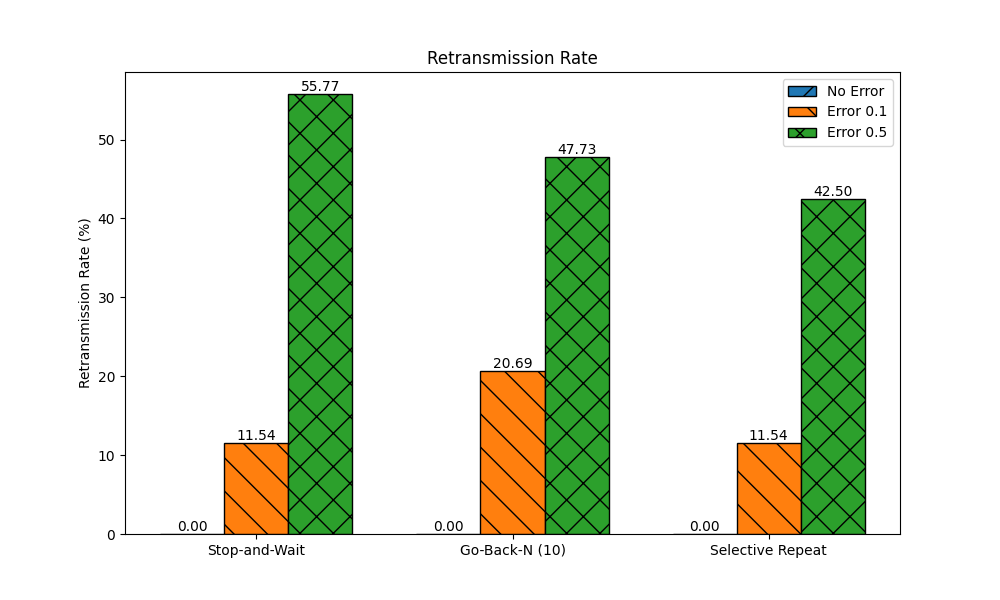
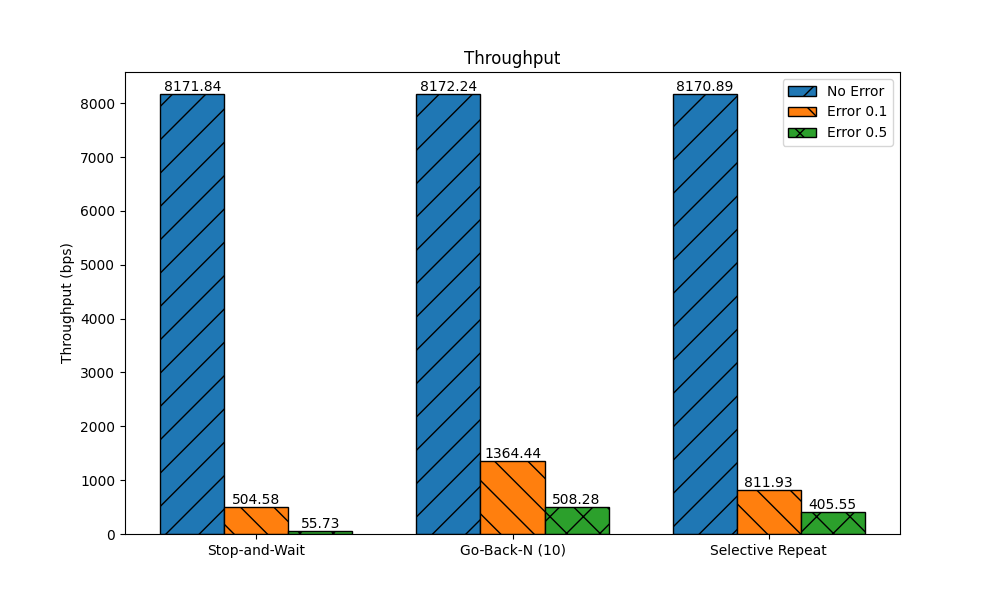
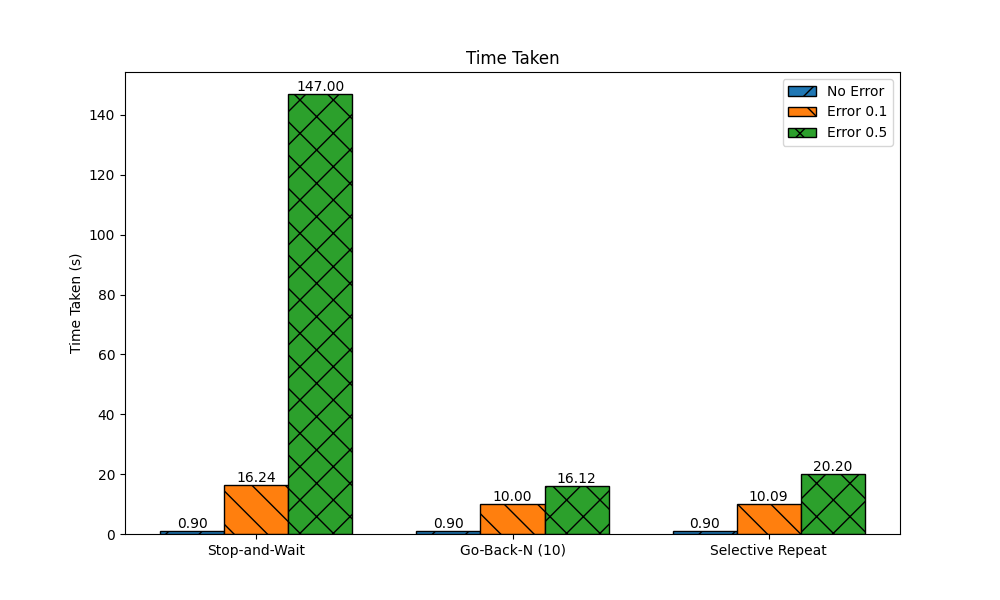
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Transfer protocol** | **Time taken** | **Throughput** | **Total frames sent** | **Total retransmissions** | **Retransmission Rate** |
| **Stop-and-Wait** | 16.24 seconds | 504.58 bps | 26 | 3 | 11.54% |
| **Go-Back-N (10)** | 10 second | 1364.44 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% |



**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.