Computer Networks Laboratory

Assignment 2

Name: Anirban Das Class: BCSE-III Roll: 001910501077 Group: A3

Problem Statement:

Implement three data link layer protocols, Stop and Wait, Go Back N Sliding Window and Selective Repeat Sliding Window for flow control.

Sender, Receiver and Channel all are independent processes. There may be multiple Transmitter and Receiver processes, but only one Channel process. The channel process introduces random delay and/or bit error while transferring frames. Define your own frame format or you may use IEEE 802.3 Ethernet frame format.

Design:

Multiple threads are used to handle multiple receivers, but at a time only a single sender thread is used. All three protocols are implemented using threads and lock class is used as well to prevent race conditions.

CODE STRUCTURE

- receiver.py – interface for the receiver side, maintains senderList and receiverList. After receiving protocol is chosen, the receiver becomes ready to receive data.

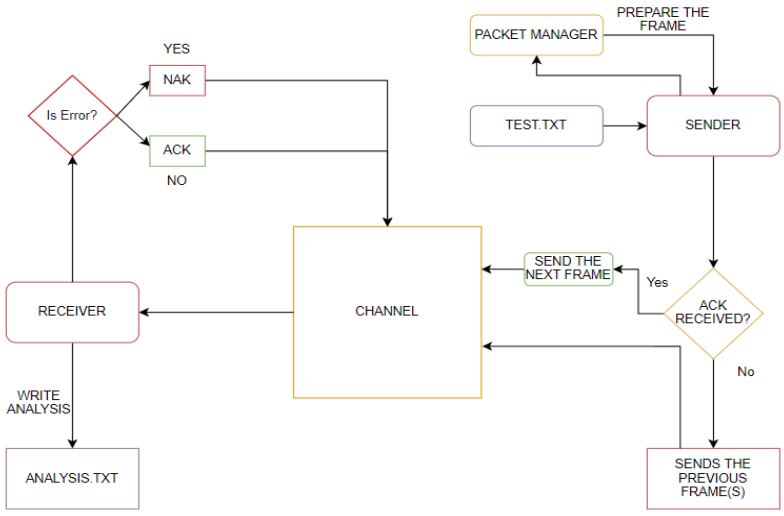
- sender.py – interface for the sender side, maintains receiver list for user to choose from.After protocol is chosen, the protocol's sender side is invoked.

- channel.py – this file handles all the error injection and thread management using lock class. The appropriate thread is used for a particular receiver when sending data to receiver as well as when receiving an ACK/NAK, same goes for the sender.

- <protocol_name>sender.py - sender side of the protocols are implemented in these 3 files.

- <protocol_name>receiver.py – receiver side of the protocols are implemented in these 3 files.

- Analysis.py – Analysis for the protocol after packet transmission is written to respective analysis files using this file



Design and Implementation of DLL protocols

1. Stop and Wait ARQ

Sender: Sender sends one data packet at a time. Sender sends the next packet only when it receives the acknowledgment of the previous packet. Therefore, the idea of stop and wait protocol in the sender's side is very simple, i.e., send one packet at a time, and do not send another packet before receiving the acknowledgment.

Receiver: Receive and then consume the data packet. When the data packet is consumed, receiver sends the acknowledgment to the sender. Therefore, the idea of stop and wait protocol in the receiver's side is also very simple, i.e., consume the packet, and once the packet is consumed, the acknowledgment is sent.

2. Go Back N ARQ

Sender: It is a sliding window protocol responsible for sending packets in windows of some fixed size and waiting for ACK for the entire window. If some frame is lost in the window before reaching the receiver, it will send the entire frame again.

Receiver: The receiver side is responsible for sending an ACK when it receives the entire window of frames and sending a NAK when a packet is lost due to time out or an erroneous packet is sent.

3. Selective Repeat ARQ

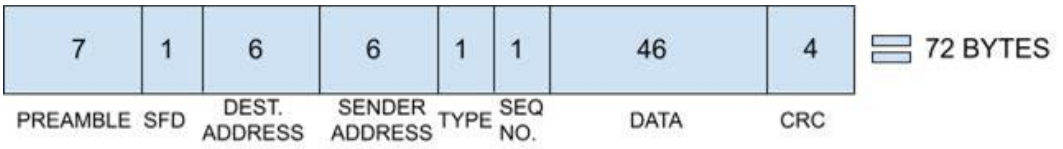
Sender: Sender can transmit new packets as long as their number is with W of all unACKed packets. It retransmits un-ACKed packets after a timeout – Or upon a NAK if NAK is employed.

Receiver: Receiver ACKs all correct packets. Receiver stores correct packets until they can be delivered in order to the higher layer.

(Window size should be less than or equal to the half of the sequence number in SR protocol. This is to avoid packets being recognized incorrectly. If the windows size is greater than half of the sequence number space, then the ACK is lost, the sender may send new packets that the receiver believes are retransmissions.)

Implementation:

We have used the IEEE 802.3 Ethernet Frame Format for our packets here. Size of a frame is 72 bytes.



IEEE 802.3 ETHERNET FRAME FORMAT

● PREAMBLE – Ethernet frame starts with 7-Bytes Preamble. This is a pattern of alternative 0's and 1's which indicates starting of the frame and allows sender and receiver to establish bit synchronization. Initially, PRE (Preamble) was introduced to allow for the loss of a few bits due to signal delays. But today's high-speed Ethernet doesn't need Preamble to protect the frame bits. PRE (Preamble) indicates to the receiver that the frame is coming and allows the receiver to lock onto the data stream before the actual frame begins.

● Start of frame delimiter (SFD) – This is a 1-Byte field that is always set to 10101011. SFD indicates that upcoming bits are starting of the frame, which is the destination address. Sometimes SFD is considered the part of PRE, this is the reason Preamble is described as 8 Bytes in many places. The SFD warns station or stations that this is the last chance for synchronization.

● Destination Address – This is a 6-Byte field that contains the port address of the destination socket.

● Source Address – This is a 6-Byte field that contains the port address of the sender socket.

● Type - This is a 1-Byte field that contains the type of the packet, whether it is a data packet or an acknowledgment.

● Sequence No. - This is a 1-Byte field that contains the sequence no. of the current frame.

● Data – This is the place where actual data is inserted, also known as Payload. Both IP header and data will be inserted here if Internet Protocol is used over Ethernet. The maximum data present may be as long as 1500 Bytes. In case data length is less than minimum length i.e. 46 bytes, then padding 0's is added to meet the minimum possible length.

● Cyclic Redundancy Check (CRC) – CRC is a 4 Byte field. This field contains a 32-bits hash code of data, which is generated over the Destination Address, Source Address, Length, and Data field. If the checksum computed by destination is not the same as the sent checksum value, the data received is corrupted

Introduction of Noise in the Data Packets:

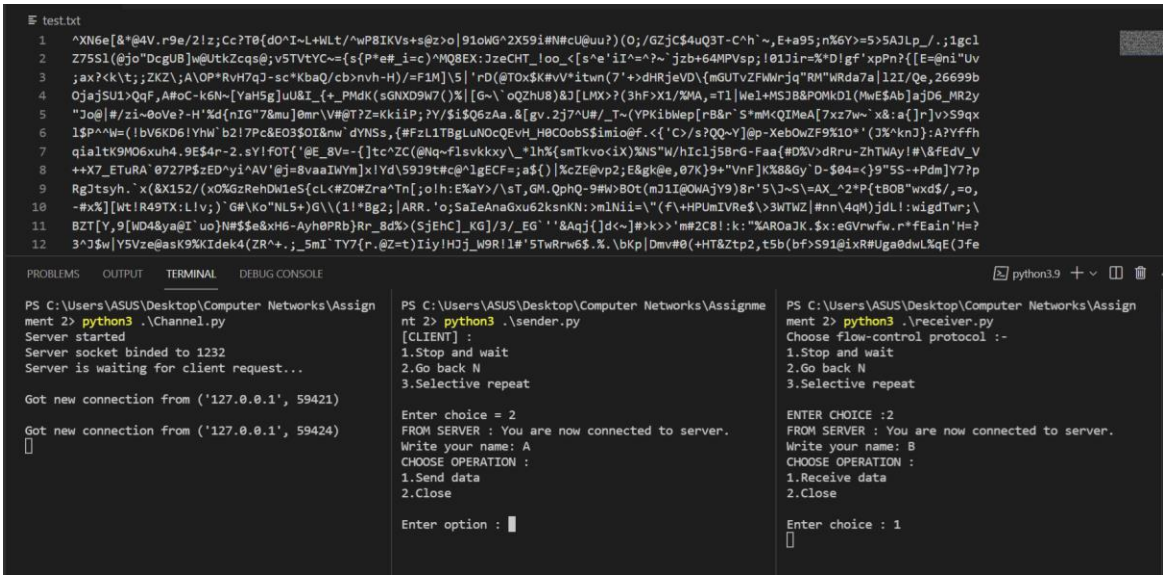
```
# Function to select if original packet or tainted packet or no packet will be sent
def process_packet(packet:str):
    # Randomly generate what would be done
    flag=random.randint(0,100)

    # If flag is 0-70, original packet will be sent
    # If flag is 71-80, taint the packet and send
    # If flag is 81-99, drop the packet
    if(flag >= 0 and flag < 65):
        return packet
    elif(flag >= 65 and flag < 80):
        return ErrorInsertion.Error.injectError(packet)
    elif(flag >= 80 and flag < 90):
        time.sleep(0.5)
        return packet
    else:
        return ''
```

Test Cases:

This script is used to generate random strings which is fed as input to the sender before generating packets.

```
import string
import random
import re

n = 46 * 50

print(n)

print(type(string.printable))
res = ''.join(random.choices(re.sub(r'\s+', '',string.printable), k = n))
# print(res)

file = open("test.txt", "w")
file.write(res)
file.close()
```

Some of the test cases along with terminal output screens are shown below:



Results:

The following analysis assumes the bandwidth to be 4000 bps for all number of packets transmitted between the server and clients.

Following metrics are used to compare the protocols:

Performance = (effectivePacketsTransmitted / totalPackets)

Throughput = (effectivePacketsTransmitted*72*8) / totalDelay

Efficiency = (Throughput / Bandwidth)

Utilization Percentage = (Efficiency * 100) %

“effectivePacketsTransmitted” and “totalDelay” denotes the amount of packets successfully acknowledged and total time taken to transmit all the data packets and receive ACK/NAK.

TABLE 1

	Performance (in %)			Utilization (in %)			Throughput (in bps)			Total Delay (in s)		
PROTOCOL NAME	**SW**	**GBN**	**SR**	**SW**	**GBN**	**SR**	**SW**	**GBN**	**SR**	**SW**	**GBN**	**SR**
No. Of Packets												
10	66.67	45.45	47.62	10.63	29.04	24.11	425	1161	964	5.94	4.8	13.5
20	48.78	51.28	47.61	5.98	35.84	32.42	239	1433	1296	48.16	8.03	8.88
30	63.82	29.12	56.60	10.29	17.49	45.77	411	699	1830	41.99	24.69	9.43
50	51.02	26.45	48.37	6.51	16.20	25.78	260	647	1031	110.54	44.45	27.92

TABLE 2

PROTOCOL	RECEIVER THROUGHPUT (in bps)	Total Delay (in s)	UTILIZATION PERCENTAGE (in %)
STOP & WAIT	333.75	51.66	8.35
GO BACK N	985	20.49	24.64
SELECTIVE REPEAT	**1280.25**	**14.93**	**32.02**

TABLE 3

PROTOCOL	PERFORMANCE RANGE (MAX-MIN) (in %)
STOP & WAIT	15.65
GO BACK N	24.83
SELECTIVE REPEAT	8.23

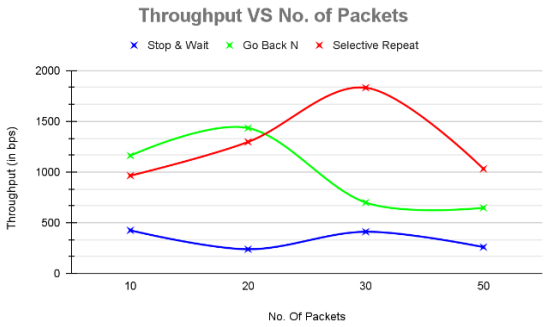


Chart A

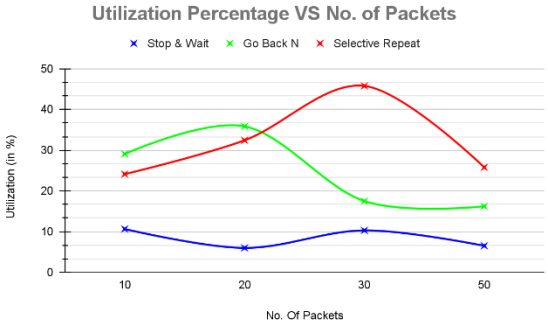


Chart B

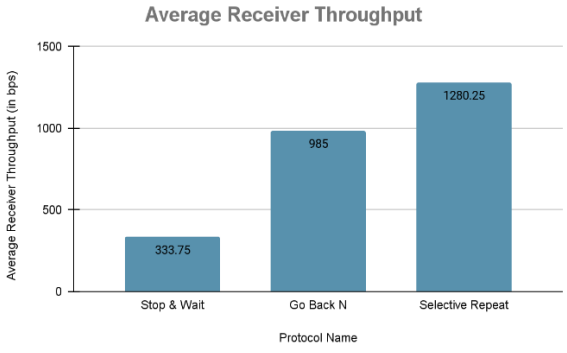


Chart C

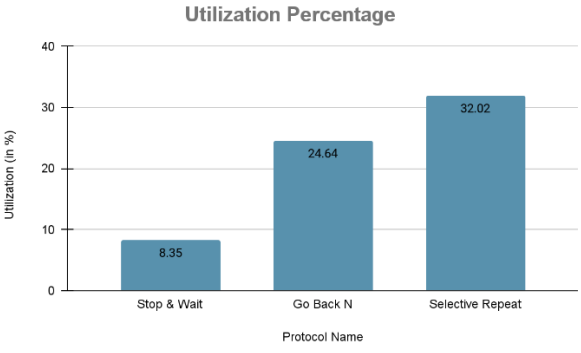


Chart D

Analysis:

The above observations give us a clear idea about how data is transmitted across channel using different protocols.

TABLE 1 shows a collection of different metric values for Stop & Wait (SW), Go Back N (GBN) and Selective Repeat (SR) protocols.

TABLE 2 gives a more detailed analysis with the average metric values for the above protocols.

Selective Repeat Protocol performs better than its peers which is visible from the highest throughput and least total delay values.

High Average Receiver Throughput value shows it can transmit most data packets successfully in a given amount of time. The Utilization Percentage value shows the high efficiency of the protocol and is just an extension of the high throughput value (as bandwidth is fixed for all the protocols)

There were cases as depicted in TABLE 1, where Stop & Wait performed better than Go Back N and Selective Repeat even after having the lowest throughput values than its peers. This might be because of the error injected in the packets and must not be used as a parameter to compare them as on average, Stop & Wait was clearly behind.

This brings us to the TABLE 3 of the observations where we have done a simple analysis comparing the “Performance Range” values of all the protocols. This shows us the reliability of the protocols as to how much can one trust them to get efficient results. Selective Repeat with the minimum range value shows that it can be trusted to give accurate results without any significant deviation from values.

N.B.: We have not considered Bandwidth-Delay Product as a metric to compare the protocols as “bandwidth” itself is a property of the channel and hence is same for all the protocols. Although, we have made a comparison on Total Delay the protocols produced during the packet transmission.

Improvements:

The implementation would have been more efficient if C/C++ was used as the programming language because of its simplicity and build in system functions.