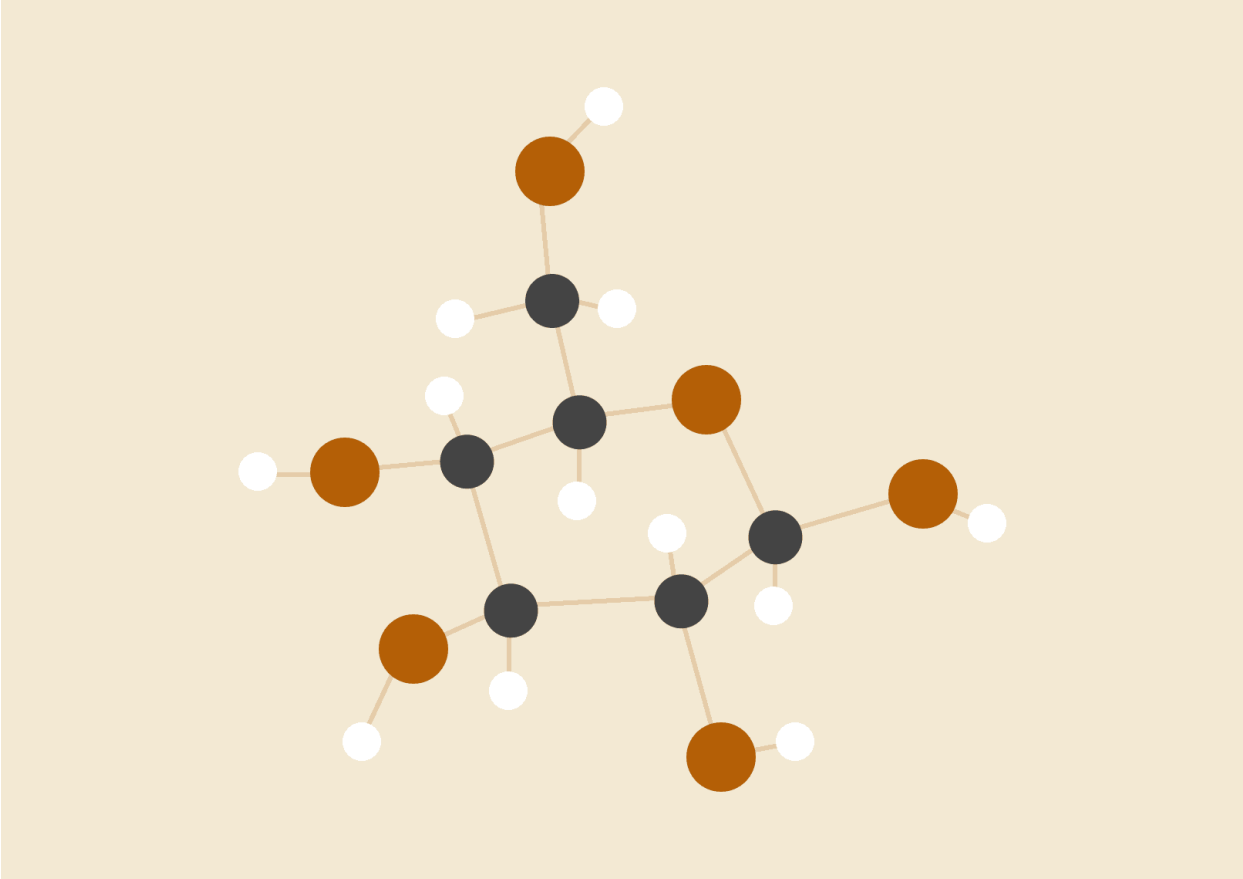
NETWORKING LAB REPORT

CLASS *BCSE III* ***SEM*** *FIFTH* ***YEAR*** *2021*



NAME Neeladri Pal

ROLL 001910501015

GROUP A1

ASSIGNMENT - 2

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.

CRC ERROR DETECTION MECHANISM

Encoding - A key is decided. Length(key) - 1 '0' bits are appended to the dataword and fed into the CRC check bit generator. Here modulo 2 binary division is performed on the augmented dataword with the key as divisor. The remainder consists of the check bits which are appended to the message to get the codeword.

Decoding - Received codeword is fed into the CRC check bit generator. If the remainder contains all '0' bits, no error is detected.

Code snippet-

```
class CRC():
   def __init__ (self, key):
       self.key = key

   def xor(self, a, b):
       """Returns XOR of 'a' and 'b' (both of same length)"""
       # initialize result
       result = []

       # Traverse all bits, if bits are
       # same, then XOR is 0, else 1
       for i in range(1, len(b)):
           if a[i] == b[i]:
               result.append('0')
           else:
               result.append('1')

       return ''.join(result)

   def generate (self, dividend):
```

```
        """Function to generate the CRC checkword by modulo-2
division"""
        divisor = self.key

        # Number of bits to be XORed at a time.
        k = len(divisor)

        # Slicing the dividend to appropriate
        # length for particular step
        tmp = dividend[0 : k]
        pick = k

        while pick < len(dividend):

            if tmp[0] == '1':

                # replace the dividend by the result
                # of XOR and pull 1 bit down
                tmp = self.xor(divisor, tmp) + dividend[pick]

            else: # If leftmost bit is '0'
                # If the leftmost bit of the dividend (or the
                # part used in each step) is 0, the step cannot
                # use the regular divisor; we need to use an
                # all-0s divisor.
                tmp = self.xor('0'*k, tmp) + dividend[pick]

            # increment pick to move further
            pick += 1

        # For the last n bits, we have to carry it out
        # normally as increased value of pick will cause
        # Index Out of Bounds.
        if tmp[0] == '1':
            tmp = self.xor(divisor, tmp)
        else:
            tmp = self.xor('0'*pick, tmp)

        checkword = tmp
        return checkword

    def getCodeword (self, dataword):
        """Function to get the codeword"""
        extraBits = '0' * (len(self.key) - 1)
```

```
        return dataword + self.generate(dataword + extraBits)

    def checkCodeword (self, codeword):
        """Function to verify the CRC"""
        rem = self.generate(codeword)

        # If the generated lrc is zero, no error detected
        if int(rem, 2) == 0 :
            return True
        else:
            return False
```

ERROR INJECTION STRATEGIES

1. Single Bit Error: A random bit from the codeword is chosen and flipped.
2. Burst Error: An error window of size (provided as argument) < codeword is chosen randomly. Within this window, some bits are chosen randomly and flipped.
3. Random Error: Few bits are randomly chosen and flipped.

Code Snippet-

```
import random

# inject error in binary string
# if burst length provided, flip bits randomly within a
# window of that length, else choose the length randomly
def injectError (data, errLength = -1, burst = False):
    # convert to list
    s = list(data)
    l = len(s)

    # if error length not provided, generate randomly
    if errLength < 0 or errLength > l:
        if burst == True:
            errLength = random.randint(2, l)
        else:
            errLength = random.randint(1, l)

    # decide the left and right indices for the error window
    left = random.randint(0, l - errLength)
    right = left + errLength - 1
```

```
    s[left] = '0' if s[left] == '1' else '1'
    s[right] = '0' if s[right] == '1' else '1'
    left += 1

    # flip random bits within the window
    while left < right:
        if random.randint(0,1) == 1 :
            s[left] = '0' if s[left] == '1' else '1'
        left += 1

    # convert to string
    errorData = ''.join(s)
    return errorData

# Function to inject single bit error in binary string
def injectSingleError (data):
    return injectError(data, 1)

# Function to inject burst error in binary string
def injectBurstError (data, errLength = -1):
    return injectError(data, errLength, True)
```

DESIGN

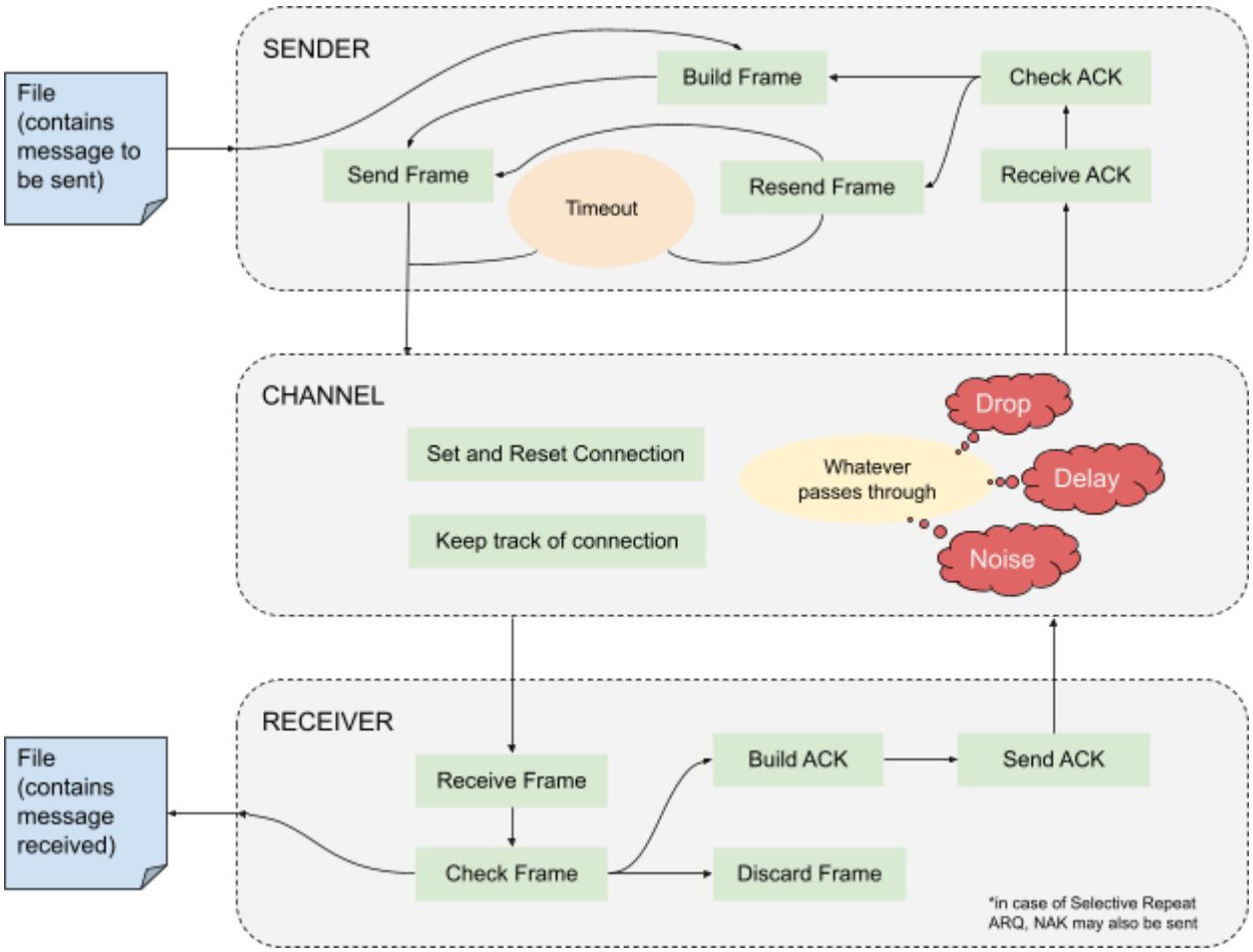
IEEE 802.3 Ethernet Frame Format								
Preamble	Frame Delimiter	Destination Address	Sender Address	Type	Sequence Number	Data	CRC Check Bits	Total
7	1	6	6	1	1	46	4	72

*all sizes are in bytes

CRC key used - `100000100110000010001110110110111` (CRC - 32)

Preamble and Frame Delimiter - used for synchronization purposes
Type - 0 (Data), 1 (ACK), 2 (NAK)
Data - contains the contents of the message

FLOW CONTROL MECHANISM



Sender reads in the **message** to be sent from the **input file**, makes a **packet** out of it, adds **redundant bits**, converts it into **frame** and sends it to the channel. The sender keeps a **copy** of the sent frame according to various protocols, lest needed later. The **noisy channel** introduces error, delay or it may even drop the frame. If anyhow the frame lands up in the receiver's arena, the receiver checks it for any sort of **error**. If everything is fine, it sends an **ACK**, else it may either silently discard or send an **NAK** (in case of Selective Repeat ARQ). The ACKs/NAKs pass through the channel and reach the sender. The sender then checks it and either **sends** the next frames or **resends** the previous (lost in transit) frames. Also, according to the specific policy, the frames may be resent by the sender based on some **timeout** policy. The receiver, when it receives the correct frames, writes them to an output file. Special attention is given to ensure the messages are written **in order**, the implementation varying through protocols. At the end, this file contains the total message received.

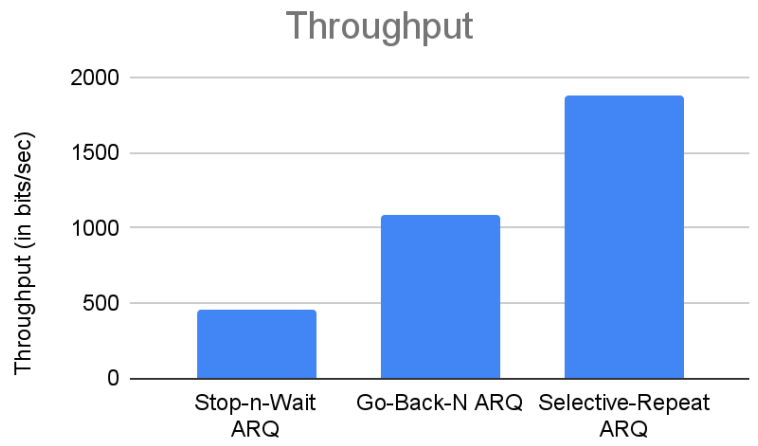
RESULTS

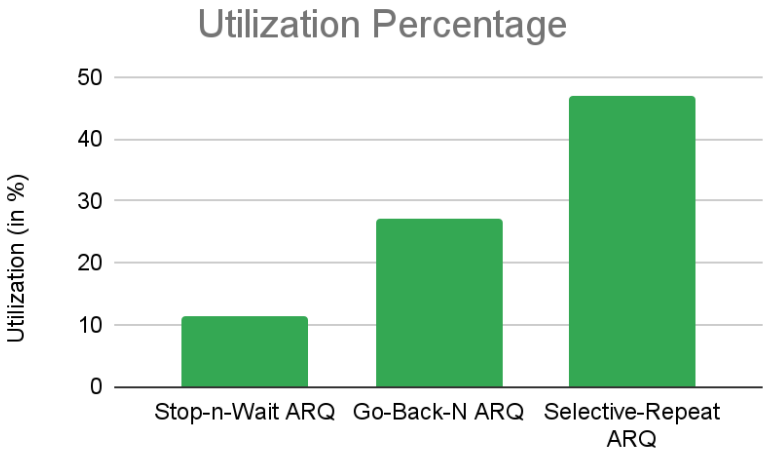
Stop-n-Wait ARQ						
Run	**Effective Frames**	**Frames sent**	**Total time taken (in sec)**	**Throughput**	**Utilization**	**Transmission time (seconds per frame)**
Run I	46	67	52	501	12.53%	1.14963
Run II	46	77	73	361	9.05%	1.59175
Run III	46	76	67	394	9.87%	1.45888
Run IV	46	65	48	546	13.66%	1.05433
Run V	46	70	58	450	11.26%	1.27854
Average	46	71	59.6	450.4	11.27%	1.306626

Go-Back-N ARQ						
Run	**Effective Frames**	**Frames sent**	**Total time taken (in sec)**	**Throughput**	**Utilization**	**Transmission time (seconds per frame)**
Run I	46	112	24	1083	27.08%	0.53168
Run II	46	105	22	1185	29.63%	0.48598
Run III	46	116	23	1112	27.82%	0.51758
Run IV	46	127	27	973	24.33%	0.59197
Run V	46	123	24	1094	27.36%	0.52637
Average	46	116.6	24	1089.4	27.24%	0.530716

Selective-Repeat ARQ						
Run	**Effective Frames**	**Frames sent**	**Total time taken (in sec)**	**Throughput**	**Utilization**	**Transmission time (seconds per frame)**
Run I	46	72	13	2013	50.33%	0.28614
Run II	46	83	13	2007	50.20%	0.28685
Run III	46	86	16	1632	40.82%	0.35279
Run IV	46	80	14	1839	45.99%	0.31311
Run V	46	73	13	1899	47.49%	0.30321
Average	46	78.8	13.8	1878	46.97%	0.30842

ANALYSIS


Throughput
Throughput (in bits/sec)
2000
1500
1000
500
0
Stop-n-Wait ARQ
Go-Back-N ARQ
Selective-Repeat ARQ


Utilization Percentage
Utilization (in %)
50
40
30
20
10
0
Stop-n-Wait ARQ
Go-Back-N ARQ
Selective-Repeat ARQ


Transmission time
Transmission time (in secs/frame)
1.5
1.0
0.5
0.0
Stop-n-Wait ARQ
Go-Back-N ARQ
Selective-Repeat ARQ

Metrics taken into consideration for comparison of Protocols - Throughput, Utilization Percentage, Transmission Time for an average frame

Bandwidth assumed - 4000 bps

- Stop n Wait is memory efficient as the sequence numbers are only 0 and 1 and thus, keeps a copy of just 1 sent frame. If the channel is thick and long, the potential of the channel is wasted because we are just waiting for an ACK from the receiver, whereas we could have sent a few packets lined up next too at the same time. This would boost the throughput to a great extent.
- The need to utilize more of the channel brings us to Go Back N ARQ, where we send many frames before waiting for ACK. This ensures that many frames are in transit at the same time, which is desired when the bandwidth-delay product is high. But here the receiver needs to accept the frames in order. So a timer is maintained on the sender side to resend the frames, in case the frame or ACK was lost during transit and thus the frame was either not acknowledged or the sender didn't receive the ACK. Whenever such happens, all the frames from the last acknowledged frames are resent by the sender.
- In Selective Repeat ARQ, multiple frames are in transit and the channel is also utilized well. The improvement here is that the receiver can accept the frames in any order. It just needs to make sure that the data is delivered to the file accurately. As a result, the frames within a window can be acknowledged in any order. 1 NAK can inform regarding the last missing packet and 1 ACK can serve as ACK for the previously received ACKs as well because an ACK is transferred only when the frames are converted in order to message and delivered to the file. This releases contention on the channel. But the out-of-order hack necessitates individual timers, so more memory overhead is present on the sender side and special care must be given to synchronization issues.

Note: Round trip Delay and Bandwidth-Delay product are properties of channel, so they are not considered for comparison.

COMMENTS

The code is very hard to implement and even harder to debug. Waiting hours to debug a synchronization issue, changing the design all over again to make it scalable is normal. At last when everything works, it works like magic. The performance metrics are essential to compare the protocols and realise which to use when. There are a ton of parameters and a number of unique cases to consider. This makes it difficult to optimise the code without running into trouble.