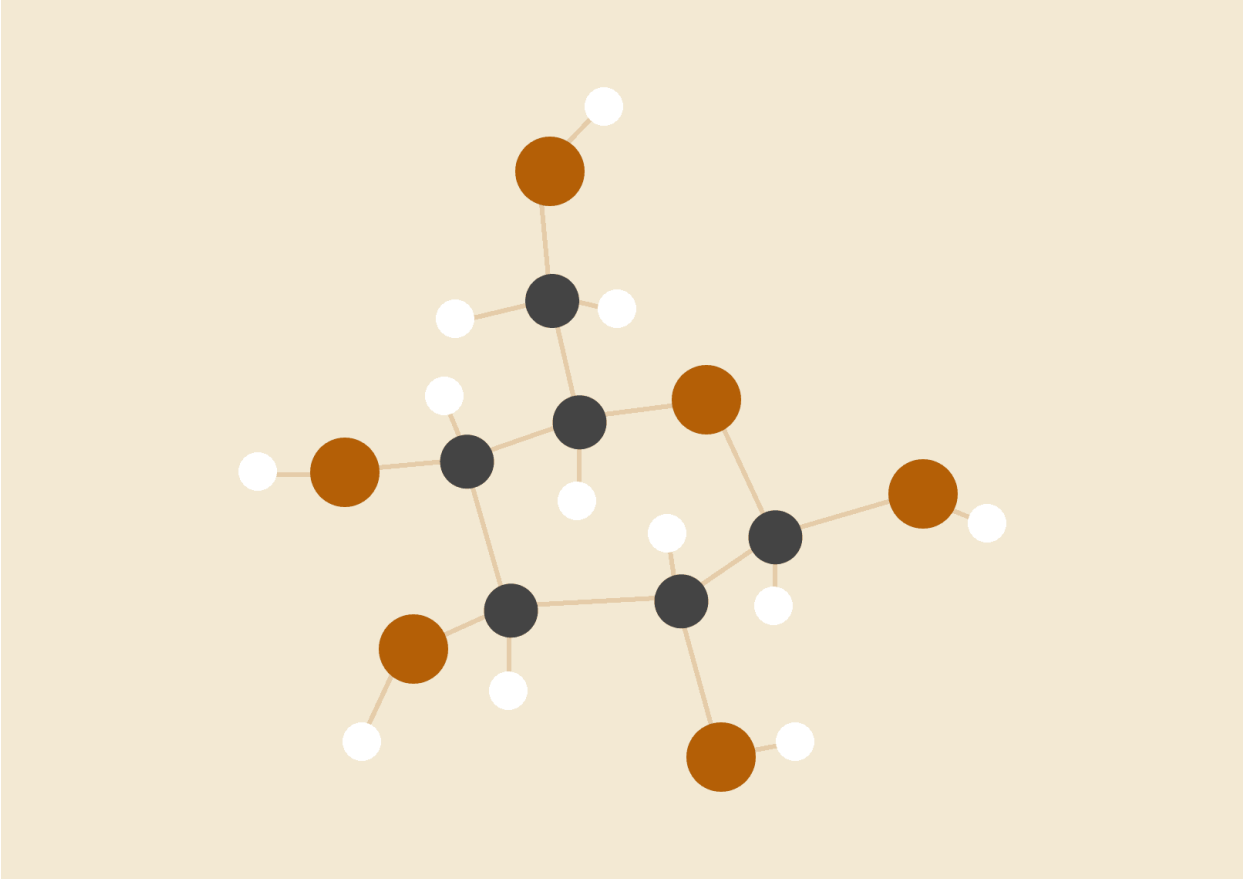
NETWORKING LAB REPORT

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



NAME Neeladri Pal

ROLL 001910501015

GROUP A1

ASSIGNMENT - 1

PROBLEM STATEMENT

Design and implement an error detection module.

Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. Please note that you may need to use these schemes separately for other applications(assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame (decide the size of the frame) from the input. Based on the scheme, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the data word from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases.

(a) Error is detected by all four schemes. Use a suitable CRC polynomial.

(b) Error is detected by checksum but not by CRC.

(c) Error is detected by VRC but not by CRC.

ERROR DETECTION MECHANISMS

1. Vertical Redundancy Check (VRC):

   Encoding - An extra '0' bit is appended to the dataword and parity (bitwise-XOR of all bits) is calculated. (dataword + parity bit = codeword)

   Decoding - Received codeword is taken and parity bit (bitwise-XOR of all bits) is calculated. If the parity bit is '0', no error is detected.

   Code snippet-

```
class VRC():
    def generate (self, dataword):
        """Function to find the parity from a dataword"""
        parity = 0

        # Calculate parity of the dataword
        for i in dataword:
            parity = parity ^ int(i, 2)
        return bin(parity)[2:]

    def getCodeword (self, dataword):
```

```
        """Function to get the codeword"""
        extraBits = '0'
        return dataword + self.generate(dataword + extraBits)

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

        # If parity is 0, no error detected
        if int(parity,2) == 0 :
            return True
        else:
            return False
```

2. Longitudinal Redundancy Check (LRC):

   Encoding - The dataword is divided into some words of word_size < dataword_size each. Another word of the same word_size containing only '0's is taken. Then we carry out bitwise-XOR of all words. The resulting word consists of the redundant bits has size=word_size. (dataword + redundant bits = codeword)

   Decoding - Received codeword is divided into words of size = word_size. Again the redundant bit word is calculated in the same manner. If the resulting word contains all '0' bits, no error is detected.

   Code snippet-

```
class LRC():
    def __init__ (self, word_size):
        self.word_size = word_size

    def generate (self, dataword):
        """Function to find the LRC from a dataword"""
        k = self.word_size
        wordCount = len(dataword) // k

        # Calculate XOR of all k-bit words
        tmp = int('0', 2)
        for i in range(wordCount):
            tmp = tmp ^ int(dataword[i*k:(i+1)*k], 2)
```

```
        # Convert the result into binary and append 0s
        # if needed to make it a k-bit word
        lrc = bin(tmp)[2:]
        if(len(lrc) < k):
            lrc = '0'*(k-len(lrc))+lrc

        return lrc

    def getCodeword (self, dataword):
        """Function to get the codeword"""
        extraBits = '0'*self.word_size
        return dataword + self.generate(dataword + extraBits)

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

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

3. CheckSum:

   Encoding - The datword is divided into some words of word_size < dataword_size each. Another word of the same word_size containing only '0's is taken. Then we carry out 1's complement addition of all words. The resulting sum, known as checksum, has size=word_size. (dataword + checksum = codeword)

   Decoding - Received codeword is divided into words of size = word_size. Again checksum is calculated in the same manner. If the resulting checksum contains all '0' bits, no error is detected.

   Code snippet-

```
class CheckSum():
    def __init__ (self, word_size):
        self.word_size = word_size

    def generate (self, dataword):
```

```
        """Function to find the Checksum from a Message"""
        k = self.word_size
        # Dividing sent message in words of k bits
        wordCount = len(dataword) // k

        # Calculating the binary sum of packets
        Sum = int ("0", 2)
        for i in range(wordCount):
            Sum += int(dataword[i*k:(i+1)*k], 2)

        Sum = bin(Sum)[2:]

        # Wrapping and adding the overflow bits
        if(len(Sum) > k):
            x = len(Sum)-k
            Sum = bin(int(Sum[0:x], 2)+int(Sum[x:], 2))[2:]
        if(len(Sum) < k):
            Sum = '0'*(k-len(Sum))+Sum

        # Calculating the complement of sum
        checksum = ''
        for i in Sum:
            if(i == '1'):
                checksum += '0'
            else:
                checksum += '1'
        return checksum

    def getCodeword (self, dataword):
        """Function to get the codeword"""
        extraBits = '0'*self.word_size
        return dataword + self.generate(dataword + extraBits)

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

        if int(checksum, 2) == 0 :
            return True
        else:
            return False
```

4. Cyclic Redundancy Check (CRC):

   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

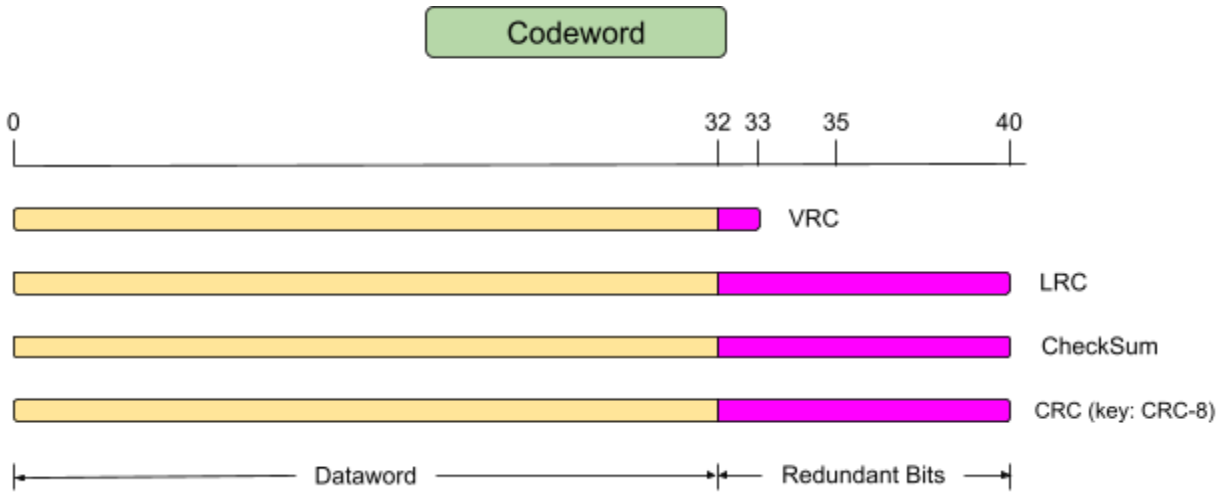
1. Single Bit Error: A random bit from the codeword is chosen and flipped.
2. Burst Error: An error window of size (provided as argument) < codeword is chosen randomly. Within this window, some bits are chosen randomly and flipped.
3. Random Error: Few bits are randomly chosen and flipped.

Code Snippet-

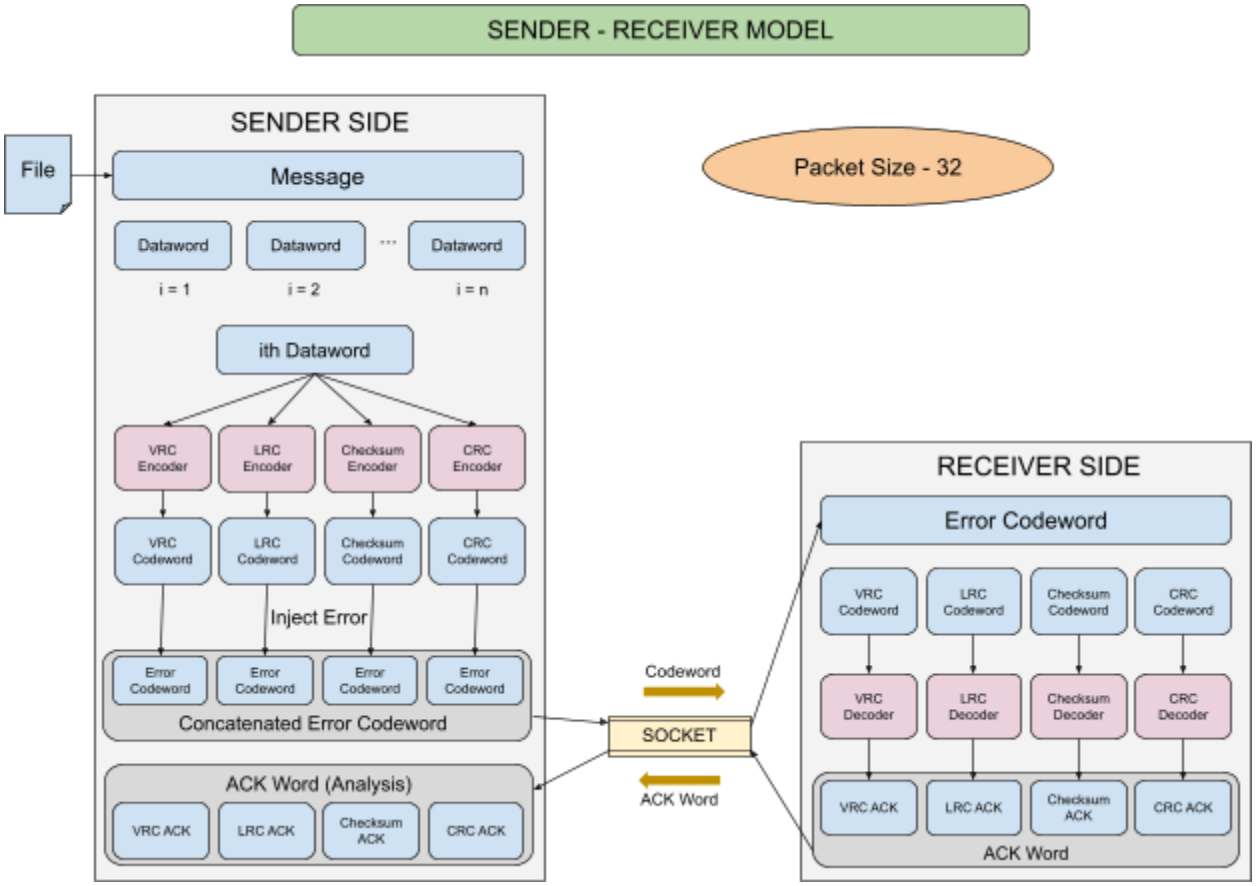
```
import random

# 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



ACK Word = {ith bit = **0** if ERROR NOT DETECTED, **1** if ERROR DETECTED, i = 0, 1, 2, 3}



Sender breaks the datawords into packets of size 32 bits. For every dataword, codewords are generated using the error checking schemes. Error is induced in each codeword according to different error injection strategies. The 4 error codewords are concatenated and sent to the receiver via socket.
On the receiver side, the concatenated codeword is broken and each of the codewords are checked. An ACK word of 4 bits is generated, where ith bit is 1 if error is detected and 0 if not detected in the corresponding codeword. The ACK word is sent back to the sender for analysis.
In the entire process, the order of codewords and bits in ACK word →
VRC, LRC, CHECKSUM, CRC(using key CRC-8)

TEST CASES

A binary string consisting of N bits is given.

The following script generates the random string --

```
import random
```

```
# generate a random binary string
def generateMessage (size):
   message = ''
   for count in range(size):
       message += str(random.randint(0,1))

   return message

n = 3200000        # size of data

file = open("testdata.txt", "w")
file.write(generateMessage(n))
file.close()
```

PACKET_SIZE = 32.

The string is divided into N / PACKET_SIZE (=PACKET_COUNT) datawords.

For each dataword, codewords are obtained using 4 schemes: VRC, LRC, Checksum, CRC-8

```
Dataword:            10001010000100101111111010101100
VRC Codeword:        10001010000100101111111010101100 0
LRC Codeword:        10001010000100101111111010101100 11001010
CHECKSUM Codeword:   10001010000100101111111010101100 10110111
CRC Codeword:        10001010000100101111111010101100 10100011
```

Error can be of any of the 4 kinds mentioned:

1. Single Bit Error

```
Dataword:                10001010000100101111111010101100

VRC Codeword:            10001010000100101111111010101100 0
VRC ErrorCodeword:       10001010000100101111111010100100 0

LRC Codeword:            10001010000100101111111010101100 11001010
LRC ErrorCodeword:       10001010000100101111111010101100 11011010

CHECKSUM Codeword:       10001010000100101111111010101100 10110111
CHECKSUM ErrorCodeword:  10001010001100101111111010101100 10110111

CRC Codeword:            10001010000100101111111010101100 10100011
CRC ErrorCodeword:       10001010000000101111111010101100 10100011
```

2. Burst Error of length 9

```
Dataword:                 10001010000100101111111010101100

VRC Codeword:             10001010000100101111111010101100 0
VRC ErrorCodeword:        10001010000100101111111001011111

LRC Codeword:             10001010000100101111111010101100 11001010
LRC ErrorCodeword:        10000111111110101111111010101100 11001010

CHECKSUM Codeword:        10001010000100101111111010101100 10110111
CHECKSUM ErrorCodeword:   10001010000100101111000111111100 10110111

CRC Codeword:             10001010000100101111111010101100 10100011
CRC ErrorCodeword:        10001010000100101111111011001100 10100011
```

3. Burst Error of length 8

```
Dataword:                 10001010000100101111111010101100

VRC Codeword:             10001010000100101111111010101100 0
VRC ErrorCodeword:        10001010000100101111111011000101 000

LRC Codeword:             10001010000100101111111010101100 11001010
LRC ErrorCodeword:        10001010000100101111111011010111 11001010

CHECKSUM Codeword:        10001010000100101111111010101100 10110111
CHECKSUM ErrorCodeword:   10001010000100101000010010101100 10110111

CRC Codeword:             10001010000100101111111010101100 10100011
CRC ErrorCodeword:        00100111000100101111111010101100 10100011
```

4. Random Error

```
Dataword:                 10001010000100101111111010101100

VRC Codeword:             10001010000100101111111010101100 0
VRC ErrorCodeword:        10001010000100101111111010101000 0

LRC Codeword:             10001010000100101111111010101100 11001010
LRC ErrorCodeword:        01000100000110110010010110100000 10010000

CHECKSUM Codeword:        10001010000100101111111010101100 10110111
CHECKSUM ErrorCodeword:   00100000101001100110000000010111 00100001

CRC Codeword:             10001010000100101111111010101100 10100011
CRC ErrorCodeword:        00001111100100011101100100100011 11100011
```

In any case, the 4 error injected codewords are appended and sent to the checker.

The receiver checks the codewords in order and returns a 4-bit ACK word, where ith bit indicates whether error is detected (=1) or not (=0).

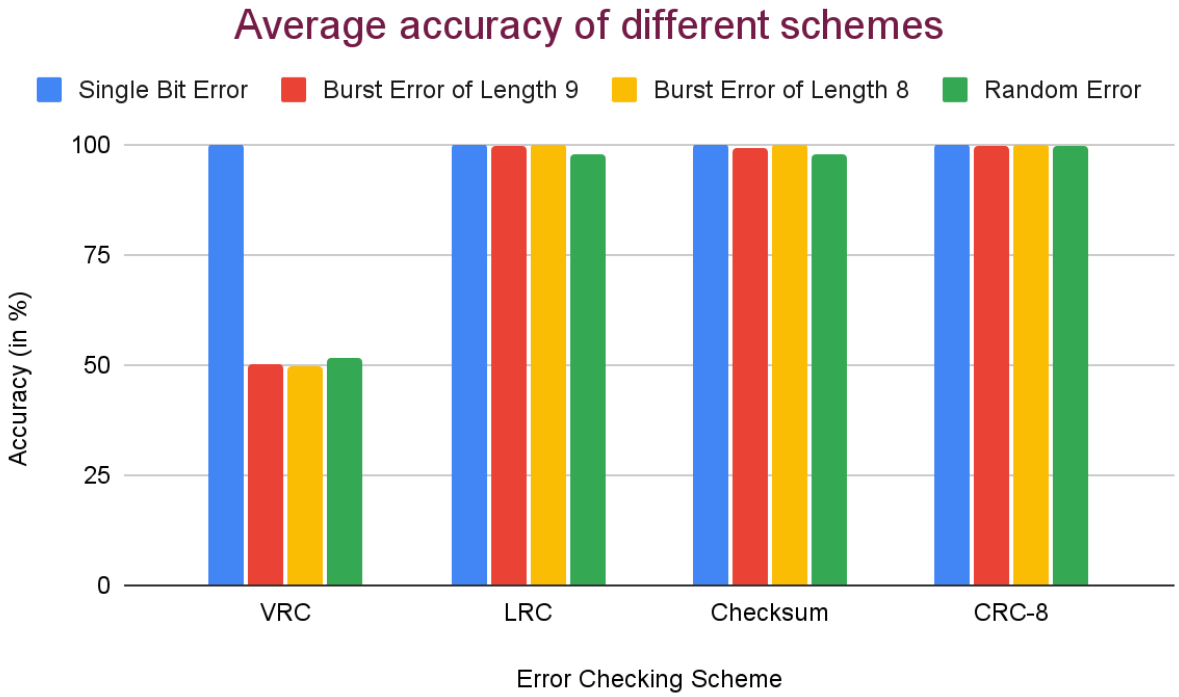
This process is repeated for all the N / PACKET_SIZE datawords.

We can keep a count of the number of errors detected for each checking mechanism to obtain the accuracy for each mechanism.

RESULTS

Accuracy Statistics (in %)					
Packet Count	100000		Packet Size	32	
Scheme	Run	VRC	LRC	Checksum	CRC-8
Single Bit Error	Run I	100	100	100	100
	Run II	100	100	100	100
	Average	100	100	100	100
Burst Error of Length 9	Run I	50.36	99.55	99.63	99.64
	Run II	50.12	99.56	99.21	99.89
	Run III	49.98	99.52	99.36	99.91
	Run IV	50.26	99.57	99.44	99.84
	Average	50.18	99.55	99.41	99.82
Burst Error of Length 8	Run I	49.32	100	99.98	100
	Run II	50.12	100	99.89	100
	Run III	50.16	100	99.91	100
	Run IV	49.72	100	99.92	100
	Average	49.83	100	99.93	100
Random Error	Run I	51.50	97.74	97.66	99.50
	Run II	51.80	97.70	97.65	99.44
	Run III	51.68	97.70	97.69	99.63
	Run IV	51.55	97.67	97.68	99.41
	Average	51.63	97.70	97.67	99.50

% accuracy = #errors detected / #packets * 100, CRC-8 key - 111010101



ANALYSIS

The following conclusions can be drawn:

1. VRC can detect all odd numbered errors, and cannot detect any even numbered errors as it simply takes a xor of all the bits in the code word. Thus, VRC can detect all single bit errors as shown in the first block of rows.

2. LRC can detect the errors which get distributed in such a way that all the columns have even numbered bit flips. In such a situation the xor will remain unchanged and error will go undetected. This is because the LRC divides the code word in packets of equal bits and then calculates bitwise XOR of the packets, so an odd number of bit flips in a particular position will generate '1' for that position and the resultant check bits will be non-zero. So single bit errors and burst errors of length 8 can be detected as in these cases, only 1 bit per column is changed. However in case of burst error of length 9, the first and last error bit will overlap, and if the bits in between do not flip, no error is detected. In general, if word_size is odd, LRC can detect all even-numbered errors and if word_size is even, LRC can detect all odd-numbered errors.

3. Checksum works by dividing the codeword into words and then calculating 1's complement binary addition of these words. It is unable to detect any error if the net change in sum is 0 or a multiple of 2^(size of each word).

4. In CRC-8, the coefficient of x^0 is 1 and 6 high bits are present, so it can detect all single bit errors. If the difference between 2 error bits is less than the number of check bits, the error is always detected. This CRC module hence can detect all burst errors of burst length less/equal to the degree of polynomial. Since the degree of the divisor polynomial is 8, the remainder in case of burst errors of length <= 8 will always be non-zero and hence these errors will be detected with 100% accuracy.

This is clearly visible that CRC with key CRC-8 outperforms all other mechanisms with a slight margin in detecting any kind of errors. VRC is the least powerful but is easy to implement. LRC and Checksum are pretty much the same in their ability to detect errors.

To summarize: ability to detect errors - **VRC** << **Checksum** ≈ **LRC** < **CRC**

Some special cases (for comparison, error is induced in only dataword):

1. Error is detected by Checksum but not by CRC (key CRC-8 "111010101")

```
Error detected by checksum but not by CRC
Dataword:            00011110100111001110110111101110
Checksum codeword:   00011110001011001010110111101111001101000
CRC Codeword:        00011110001011001010110111101111000001010
```

2. Error is detected by VRC but not by CRC - For this, CRC-4 ("10101") is taken as the key because CRC-8 will detect all odd-numbered errors as it has a factor x + 1

```
Error detected by VRC but not by CRC
Dataword:            01101110010101110001111101011010
VRC codeword:        01101110010101110111001011001101 01
CRC Codeword:        01101110010101110111001011001101 00110
```

COMMENTS

The assignment helps to gain in-depth understanding of different error checking schemes, how they work, their advantages and disadvantages. This will enable me to deal with possible bugs. For example, a good choice of CRC polynomial can eliminate errors to a much larger extent. The project is simulated using a client-server socket model which gives a sense of real-word scenario, and can be improved through a module simulating the channel, instead of inducing errors on the sender side itself.