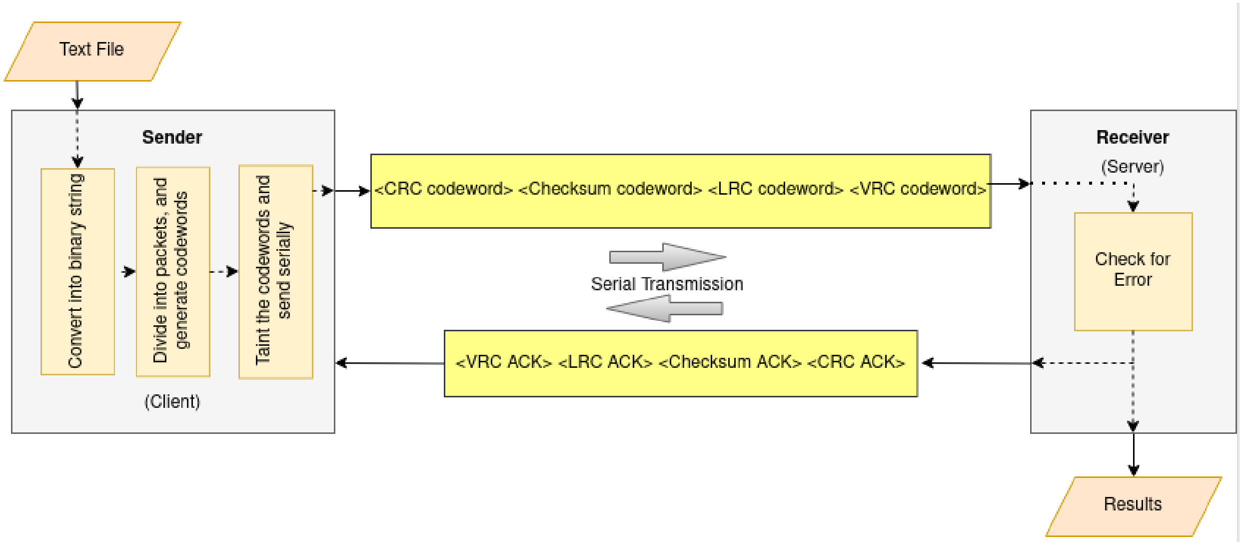
# Networks Lab
# Assignment 1
### Name: Koustav Dhar Class: BCSE UG-III Group: A1 Roll: 001910501022

## Problem Statement:
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 schemes, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword 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 (list is given on next page).
(b) Error is detected by checksum but not by CRC.
(c) Error is detected by VRC but not by CRC.
[Note: Inject error in random positions in the input data frame. Write a separate method for that.]

## Design:
In my error detection model, I have used Socket Programming to transfer the data frames from sender to receiver. The model is a 2-block model, a Client, and a Server, with the client being the sender and the server being the receiver.
Sender Side:
- ➔ Data is extracted from a text file.
- ➔ The data is converted into a binary string.
- ➔ The data is divided into packets(i.e. 32 in my design).
- ➔ For each packet, codewords are generated for all the error schemes.
- ➔ Errors are injected into the codewords.
- ➔ Tainted codewords are sent serially to the server(receiver).

Receiver Side:
- ➔ Codewords are received serially for all error schemes, for each packet.
- ➔ Dataword portion of the codeword is extracted, and new codewords are compared with the received ones. This is the check block of the receiver.
- ➔ If an error is detected, increment a counter corresponding to the respective scheme, else not.
- ➔ Send according to acknowledgment back to the client(sender), whether the error is detected or not.



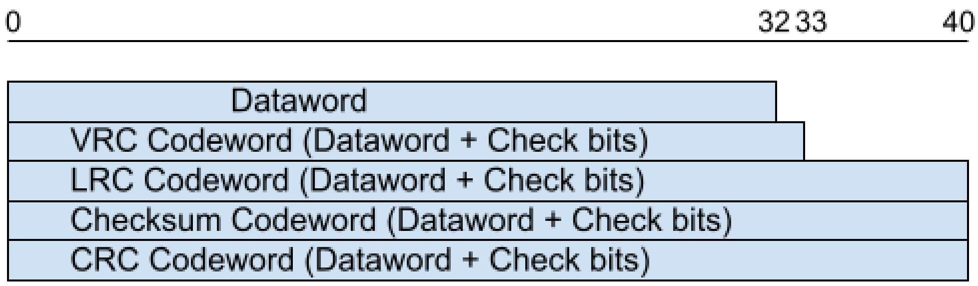

**Client-Server Model**

**Input:**
A 'test.txt' file is created, where a random test case is stored consisting of printable ASCII characters except whitespaces. The size of the string is approximately $4 \times 10^5$ characters. The string is extracted from the text file and then converted to a binary string, which is evidently the input. Then, it performs the packet division and continues the process.

**Output:**
Both the Sender side and Receiver side prints the sent and received codewords respectively, along with the error scheme. The receiver side displays if the error is detected or not, and the same is sent to the sender as an acknowledgment. At last, the results are displayed, i.e. which error scheme detected errors in how many packets and respective accuracy is also printed.

## Implementation:

The codewords generated for the respective error schemes follow the following structures.




**VRC:** Vertical Redundancy Check is also known as Parity Check. In this method, a redundant bit also called parity bit is added to each data unit. Odd parity check has been used here.

**vrc.py**

```
class VRC:
   redbits = 1
   @classmethod
   def getParity(cls, data):
       parity = 0
       for c in data:
           parity ^= int(c)   # finding the parity by XORing the set bits
       return (data + str(parity))
   @classmethod
   def parityCheck(cls, data):
```

```
        ndata = cls.getParity(data[0:len(data) - 1])
        if ndata == data:  # odd parity VRC
            return True
        return False
```

**LRC:** Longitudinal Redundancy Check is also known as 2-D parity check. In this method, data which the user wants to send is organised into tables of rows and columns. A block of bits is divided into a table or matrix of rows and columns. In order to detect an error, a redundant bit is added to the whole block and this block is transmitted to the receiver. We have splitted the bits in 4 rows.

**lrc.py**

```
# considering packet size of 32bits
# splitted in 4 rows containing 8bits each
class LRC:
   k = 4
   redbits = 8
   @classmethod
   def generateLRC(cls, data):
       table = []
       for i in range(cls.k):
           table.append([])
       n = len(data)
       m = n // cls.k
       for i in range(n):  # creating the 2d table for generating column wise
LRC
           table[i // m].append(int(int(data[i])))
       lrc = ""
       for i in range(m):
           v = 0
           for j in range(cls.k):  # finding XOR of every column
               v ^= table[j][i]
           lrc += str(v)
       data += lrc
       return data # returning the total data with LRC check bits added
   @classmethod
   def checkLRC(cls, data):
       n = len(data)
       m = n // (cls.k + 1)
       lrc = data[m * cls.k : ]    # getting the old LRC bits
       newdata = cls.generateLRC(data[ : cls.k * m])    # generating the new
LRC
       newlrc = newdata[cls.k * m : ]  # extracting the LRC part
       if newlrc != lrc:   # checking if they match
           return False
       return True
```

**Checksum**: In checksum error detection scheme, the data is divided into k segments each of m bits. In the sender's end the segments are added using 1's complement arithmetic to get the sum. The sum is complemented to get the checksum. The checksum segment is sent along with the data segments. At the receiver's end, all received segments are added using 1's complement arithmetic to get the sum. The sum is complemented. If the result is zero, the received data is accepted; otherwise discarded.

**checksum.py**

```
# considering packet size of 32bits
# splitted in 4 rows containing 8bits each
class CheckSum:
    k = 4
    redbits = 8
    @classmethod
    def generateCheckSum(cls, data):
        table = []
        for i in range(cls.k):
            table.append([])
        n = len(data)
        m = n // cls.k
        for i in range(n):  # creating a 2d table for column wise checksum
            table[i // m].append(int(int(data[i])))
        checksum = [0 for i in range(m)]
        carry = 0
        for i in range(cls.k):
            for j in range(m - 1, -1, -1):
                checksum[j] += carry + table[i][j]
                carry = checksum[j] // 2
                checksum[j] %= 2
        while carry != 0:
            for i in range(m - 1, -1, -1):
                checksum[i] += carry
                carry = checksum[i] // 2
                checksum[i] %= 2
        for i in range(m):
            checksum[i] ^= 1
        cs = ""
        for i in range(m):
            cs += str(checksum[i])
        data += cs
        return data # returning the total data with LRC check bits added
    @classmethod
    def checkSum(cls, data):
        table = []
        for i in range(cls.k + 1):
            table.append([])
        n = len(data)
        m = n // (cls.k + 1)
        for i in range(n):  # creating a 2d table for column wise checksum
            table[i // m].append(int(int(data[i])))
        checksum = [0 for i in range(m)]
        carry = 0
        for i in range(cls.k + 1):
```

```
        for j in range(m - 1, -1, -1):
            checksum[j] += carry + table[i][j]
            carry = checksum[j] // 2
            checksum[j] %= 2
    while carry != 0:
        for i in range(m - 1, -1, -1):
            checksum[i] += carry
            carry = checksum[i] // 2
            checksum[i] %= 2
    for i in range(m):
        checksum[i] ^= 1
    for i in range(m):
        if checksum[i] == 1:
            return False
    return True
```

**CRC:** Unlike checksum scheme, which is based on addition, CRC is based on binary division. In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of the data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number. At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted. A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.

We have used CRC-8 i.e. $x^8 + x^7 + x^6 + x^4 + x^2 + 1$as our polynomial.

**crc.py**

```
class CRC:
   @classmethod
   def xor(cls, a, b):
       x = ""
       for i in range(1, len(b)):
           if a[i] == b[i]:
               x += "0"
           else:
               x += "1"
       return x
   @classmethod
   def binaryDivision(cls, divident, divisor):
       m = len(divisor)
       n = len(divident)
       temp = divident[0 : m]

       while m < n:
           if temp[0] == '1':
               temp = cls.xor(divisor, temp) + divident[m]
           else:
               temp = cls.xor("0"*len(divisor), temp) + divident[m]
           m += 1
       if temp[0] == '1':
           temp = cls.xor(divisor, temp)
```

```
        else:
            temp = cls.xor("0"*len(divisor), temp)
        return temp
    @classmethod
    def generateCRC(cls, data, poly):
        k = len(poly)
        ndata = data + "0"*(k - 1)
        rem = cls.binaryDivision(ndata, poly)
        crc = data + rem
        return crc
    @classmethod
    def checkCRC(cls, data, poly):
        rem = cls.binaryDivision(data, poly)
        if rem == "0"*len(rem):
            return True
        return False
```

**Error Injection**: 3 types of error injection methods are considered. (a) Single bit error (b) Burst error and (c) Complete Burst error or Patch error. During error injection, one of the types is selected randomly and error is injected.

**error.py**

```
from random import randint
# packet size of 32
class Error:
    @classmethod
    def singleError(cls, data):
        n = len(data)
        # n = 31
        index = randint(0, n - 1)
        edata = ""
        for i in range(len(data)):
            if i == index:
                edata += str((int(data[i])) ^ 1)
            else:
                edata += data[i]
        return edata
    @classmethod
    def completeBurstError(cls, data):
        n = len(data)
        # n = 31
        indexl = randint(0, n - 1)
        indexr = randint(indexl, n - 1)
        edata = ""
        for i in range(len(data)):
            if i >= indexl and i <= indexr:
                edata += str((int(data[i])) ^ 1)
            else:
                edata += data[i]
```

```
        return edata
    @classmethod
    def burstError(cls, data):
        n = len(data)
        # n = 31
        indices = [i for i in range(n)]
        # shuffle
        for i in range(n):
            swapind = randint(i, n - 1)
            indices[i], indices[swapind] = indices[swapind], indices[i]
        m = randint(2, n)
        s = set()
        for i in range(m):
            s.add(indices[i])
        edata = ""
        for i in range(len(data)):
            if i in s:
                edata += str((int(data[i])) ^ 1)
            else:
                edata += data[i]
        return edata
    @classmethod
    def injectError(cls, data):
        methods = [cls.singleError, cls.completeBurstError, cls.burstError]
        index = randint(0, 2)
        return methods[index](data)
```

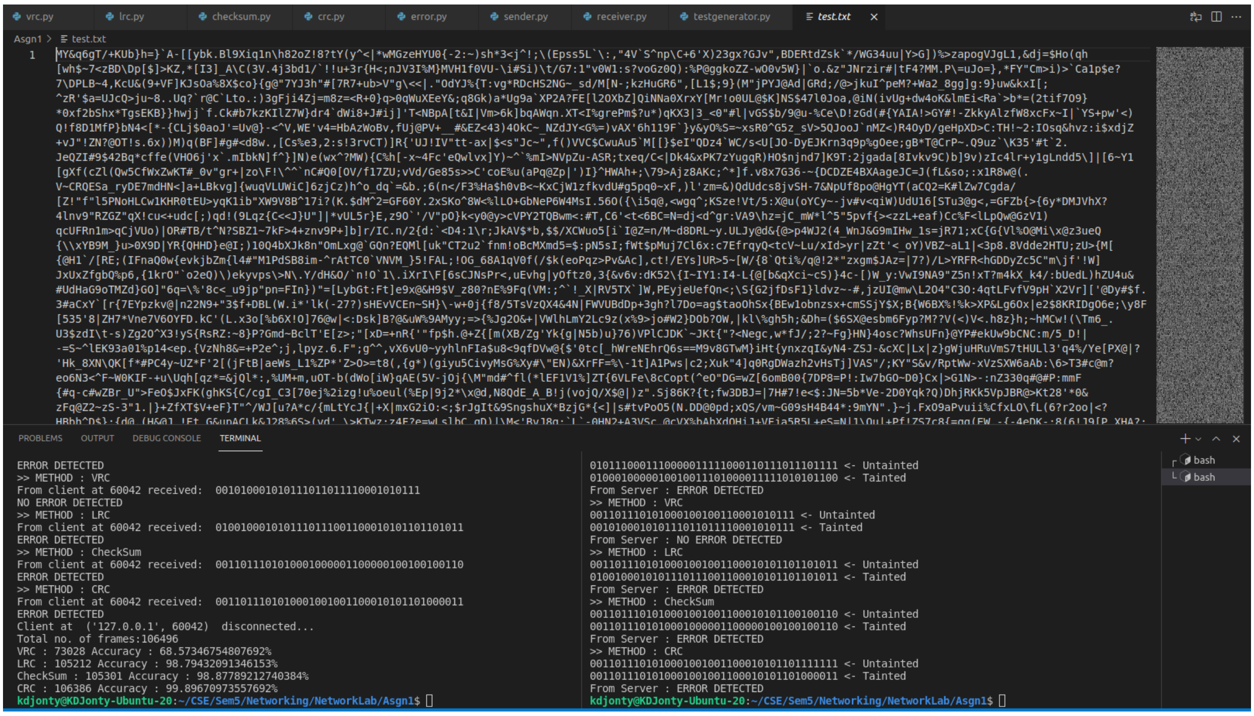
**Test Cases:**

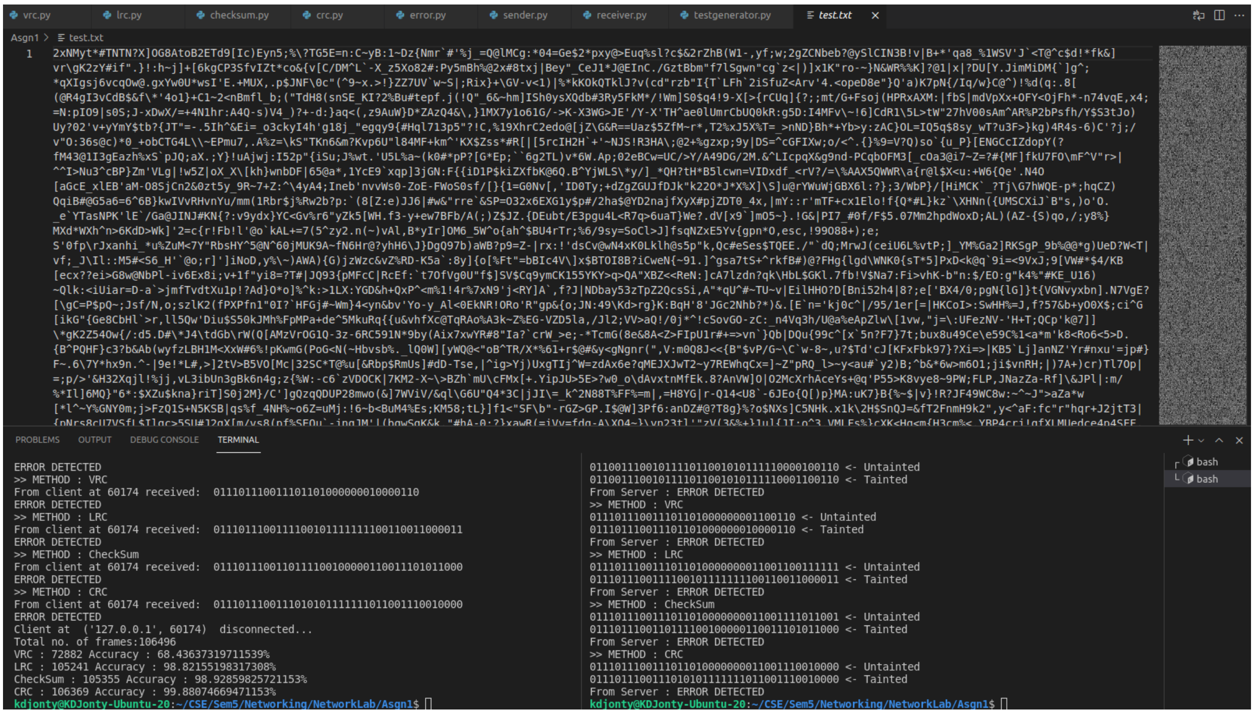
We have generated random ASCII strings without whitespaces, of about $4 \times 10^5$ lengths, and performed all the tests on them.

**testgenerator.py**

```
import string
import random
import re

n = 13 * (32 ** 3)

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

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

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




## Results:

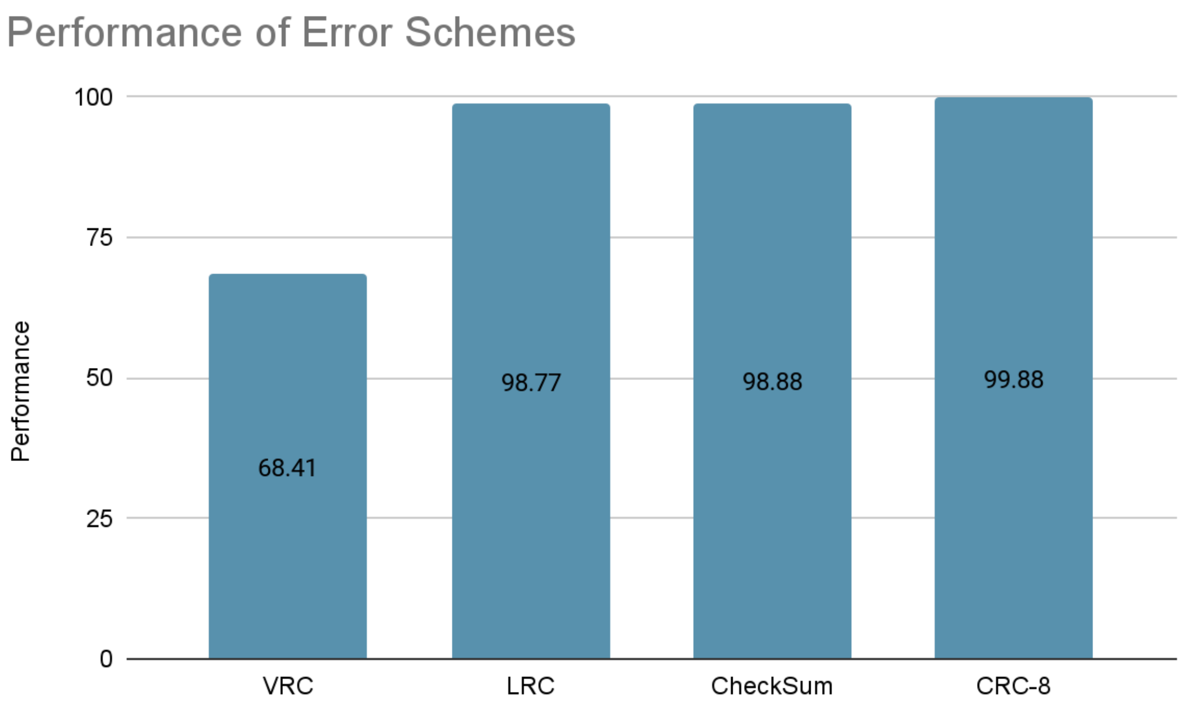
Accuracy has been used as the metric for performance evaluation here. We define accuracy as,

$$Accuracy\ of\ an\ Error\ Scheme\ =\ \frac{No.\ of\ packets\ where\ error\ is\ detected\ by\ the\ scheme}{Total\ no.\ of\ packets}$$

$Total\ no.\ of\ packets\ used\ for\ testing\ =\ 106496.$

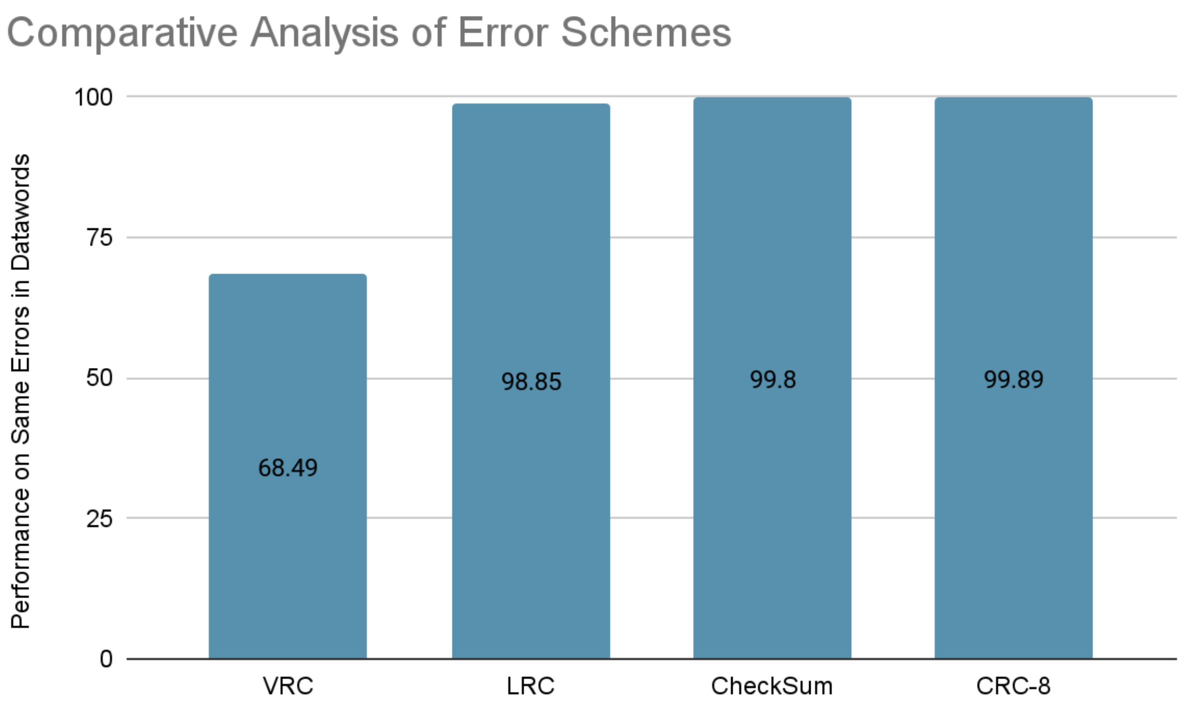
Errors in Complete Codeword (Performance Check)

| Runs / Scheme | VRC | LRC | Checksum | CRC |
|---|---|---|---|---|
| Run 1 | 73065 (68.61%) | 105173(98.76%) | 105357 (98.93%) | 106386 (99.89%) |
| Run 2 | 72746 (68.30%) | 105165(98.75%) | 105274 (98.85%) | 106370 (99.88%) |
| Run 3 | 72766 (68.32%) | 105176(98.76%) | 105270 (98.84%) | 106356 (99.86%) |
| Run 4 | 72887 (68.44%) | 105258(98.83%) | 105331 (98.90%) | 106373 (99.88%) |
| Run 5 | 72850 (68.40%) | 105191(98.77%) | 105344 (98.91%) | 106368 (99.87%) |
| *Mean* % | 68.41% | 98.77% | 98.88% | 99.88% |




Errors in only Dataword (Comparative Analysis on Same Dataword Errors)

| Runs / Scheme | VRC | LRC | Checksum | CRC-8 |
|---|---|---|---|---|
| Run 1 | 72842 (68.39%) | 105242(98.82%) | 106295 (99.81%) | 106381 (99.89%) |
| Run 2 | 72828 (68.38%) | 105316(98.89%) | 106274 (99.79%) | 106377 (99.88%) |
| Run 3 | 72839 (68.39%) | 105235(98.81%) | 106291 (99.80%) | 106376 (99.88%) |
| Run 4 | 73203 (68.73%) | 105350(98.92%) | 106293 (99.80%) | 106389 (99.89%) |
| Run 5 | 73032 (68.57%) | 105268(98.84%) | 106299 (99.81%) | 106383 (99.89%) |
| *Mean* % | 68.49% | 98.85% | 99.80% | 99.89% |




Special Classes of Errors

A) Single Bit Errors

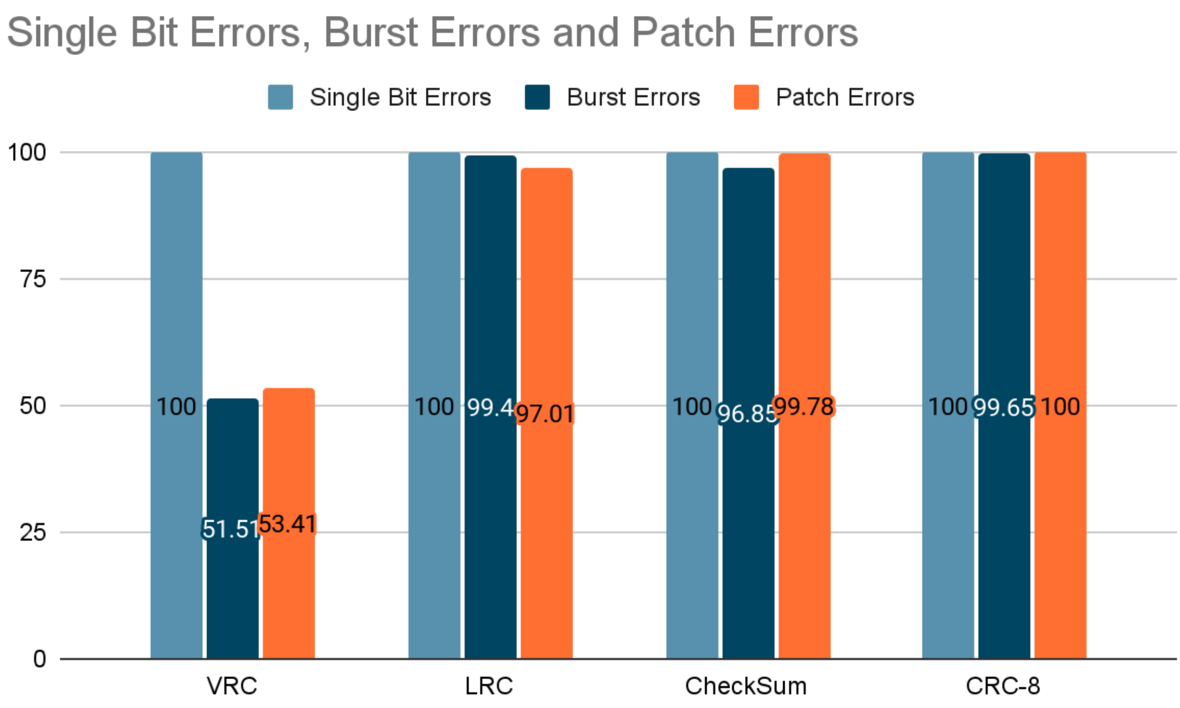
| Runs / Scheme | VRC | LRC | Checksum | CRC-8 |
|---|---|---|---|---|
| Run 1 | 106496 (100%) | 106496 (100%) | 106496 (100%) | 106496 (100%) |
| *Mean* % | 100% | 100% | 100% | 100% |

B) Burst Errors

| Runs / Scheme | VRC | LRC | Checksum | CRC-8 |
|---|---|---|---|---|
| Run 1 | 54890 (51.54%) | 105850(99.39%) | 103223 (96.92%) | 106087 (99.61%) |
| Run 2 | 54975 (51.62%) | 105859(99.40%) | 103120 (96.82%) | 106143 (99.66%) |
| Run 3 | 54728 (51.38%) | 105866(99.40%) | 103113 (96.82%) | 106161 (99.68%) |
| *Mean* % | 51.51% | 99.40% | 96.85% | 99.65% |

C) Patch Errors (Complete Burst Errors)

| Runs / Scheme | VRC | LRC | Checksum | CRC-8 |
|---|---|---|---|---|
| Run 1 | 56803 (53.33%) | 103355(97.05%) | 106269 (99.78%) | 106496 (100%) |
| Run 2 | 56884 (53.41%) | 103294(96.99%) | 106267 (99.78%) | 106496 (100%) |
| Run 3 | 56963 (53.48%) | 103301(96.99%) | 106252 (99.77%) | 106496 (100%) |
| *Mean* % | 53.41% | 97.01% | 99.78% | 100% |




**Analysis:**

From the tables and charts presented in the Results section, we can observe how each of the error schemes is performing. In terms of general performance, it is evident that CRC performs the best in terms of error detection among all the schemes. VRC performs the worst among the four error schemes, which can be explained since only a single check bit accounts for all the bits in the dataword. We discuss some other observations as follows:

- For analysis purposes, if the error injection is limited to the dataword only, the CheckSum method shows a significant ≈1% increase in terms of accuracy, while the other error detection schemes show insignificant improvements.
- For analysis purposes, if the error is limited to single-bit errors only, all the methods tend to detect errors each time.
- For analysis purposes, if the error is limited to burst errors only, the performance of VRC drops significantly by ≈17%, showing that VRC isn't good enough for detecting burst errors. The performance of Checksum also drops by a significant ≈2%. On contrary, LRC tends to perform quite well for burst errors, its performance increasing by ≈0.7%. The performance of CRC for burst errors drops by an insignificant amount.
- For analysis purposes, if the error is limited to patch errors only, patch errors being a kind of complete burst errors, such that all bits in a range of the codeword are tainted. The performance of VRC drops significantly by ≈15%, showing that VRC isn't good enough for detecting patch errors. The performance of LRC also drops by a significant ≈1.7%. On contrary, Checksum tends to perform quite well for burst errors, its performance increasing by ≈1%. CRC tends to detect each of the patch errors.

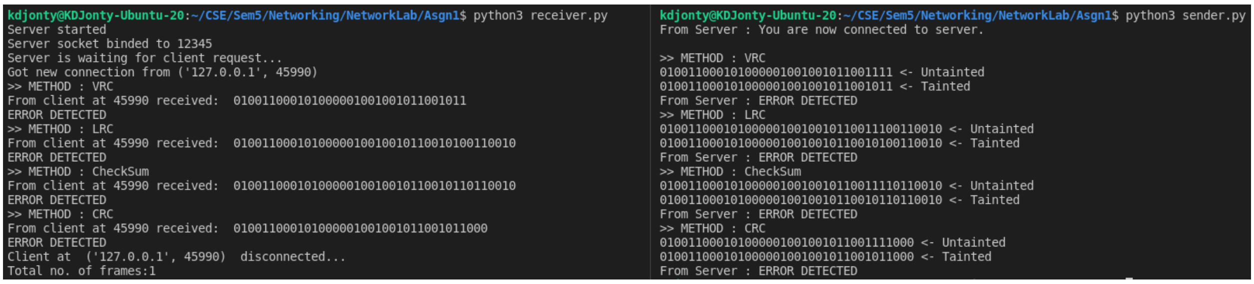
Now, we take a look at some test cases, where:

**(a) Error is detected by all four schemes.**

For the following test case (a single bit error), all four schemes detect the error.

```
01001100010100000100100101100111 <- Original Dataword
01001100010100000100100101100101 <- Tainted Dataword
```

Output:




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

For the following test case, Checksum detects the error, but CRC doesn't.

```
01101111011011100110001100100110 <- Original Dataword
00011000111000010111111011011000 <- Tainted Dataword
```

Output:


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

A CRC polynomial with $x + 1$as a factor can detect all errors affecting an odd number of bits, and VRC can't detect errors affecting even numbers of bits, since it uses a parity check. Thus, CRC-8 won't satisfy this special case for any testcase.

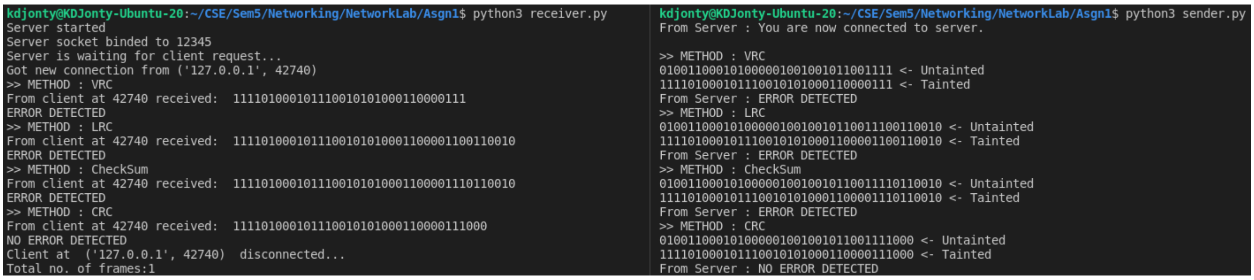
Hence, we have taken CRC polynomial as $x^4 + x^2 + 1$.

```
crcpoly = "10101"
```

For the following test case, VRC detects the error, but CRC doesn't.

```
01001100010100000100100101100111 <- Original Dataword
11110100010111001010100011000011 <- Tainted Dataword
```

Output:




**Improvements:**

- This could also have been done using a 3-block Sender, Channel, Receiver model, where the error is injected in the Channel, but for simplicity, it is done in 2 blocks, where the error is injected in the sender side only.
- This would have been more efficient if it was implemented in a language closer to the system such as C/C++.