



Computer Networks Lab Report - Assignment 1

Name: Bikash Sah

Class: BCSE-3

Group: A1

Assignment Number: 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 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 (not limited to).

- (a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given in 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.]

Submission Date: **7 August 2022**

Deadline: **7 August 2022**

Introduction

First question that comes in our mind is that, why even do we need an error detection module? Answer to that is, when we transmit/receive data it is in bits. These bits travel through a medium, and these stream of bits are subjected to electromagnetic (optical) interference due to noise signals (light sources). Thus, the data transmitted may be prone to errors.

So, we need some technique that would help us determine errors in the signal(in the receiver side) and ask the transmitter to retransmit it.

In this project, we will mainly focus on 4 error detection techniques.

1. Vertical Redundancy Check(VRC)

- It is also known as **Parity Check**.
- In this method, a redundant bit also called parity bit is added to each data unit.
- This method includes even parity and odd parity.
- Even parity means the total number of 1s in data(Including redundant bits) is to be even and odd parity means the total number of 1s in data(Including redundant bits) is to be odd.
- We will use **Even Parity** in this project.

First the data bits are split into 4 bit groups.

For example, for the message `1 0 0 1 0 0 0 1 1 1 1 1` with even parity bit

scheme, the message to be transmitted is `1 0 0 1 0` `0 0 0 1 1` `1 1 1 1 0`

The 5th bit at the end of the nibble represents the even parity bit corresponding to that nibble.

Advantages:

- VRC can detect all single bit error.
- It can also detect burst errors but only in those cases where number of bits changed is odd, i.e. 1, 3, 5, 7,etc.

Disadvantages:

- It is not able to detect burst error if the number of bits changed is even, i.e. 2, 4, 6, 8,etc.

2. Longitudinal Redundancy Check(LRC)

- It is also known as 2 Dimensional Parity Check.
- Organize data into a table and create a parity for each column.

Example:

Suppose the message to be transmitted is `1 0 1 1 1 0 0 0 1 0 0 1`

Then, we compute the even parity nibble as follows:

`1 0 1 1`

`1 0 0 0`

`1 0 0 1`

`1 0 1 0`

We note that in this scheme, the number of 1's in each column including the bit in the parity nibble must be even.

Data is sent along with parity bits : `1 0 1 1 1 0 0 0 1 0 0 1 1 0 1 0`

Advantages:

- LRC is used to detect burst errors.

Disadvantage:

- The main problem with LRC is that, it is not able to detect error if two bits in a data unit are damaged and two bits in exactly the same position in other data unit are also damaged.

3. Check Sum

- In check sum method, we divide the stream of bits in 'k' blocks, each block consisting 'n' bits.
- We add all k blocks, if carry is generated it is again added to the sum.
- We do 1's complement of the sum and we get the required checksum value.
- In receiver's side all the values of codewords are added and 1's complement is done of the sum, if sum==0 then No errors else error.

4. Cyclic Redundancy Check

- CRC performs mod 2 arithmetic (exclusive-OR) on the message using a divisor polynomial. Firstly, the message to be transmitted is appended with

CRC bits and the number of such bits is the degree of the divisor polynomial.

- The divisor polynomial 1 1 0 1 corresponds to the $x^3 + x^2 + 1$.

For example:

for the message 1 0 0 1 0 0 with the divisor polynomial 1 1 0 1, the message after appending CRC bits is 1 0 0 1 0 0 0 0. We compute CRC on the modified message M.

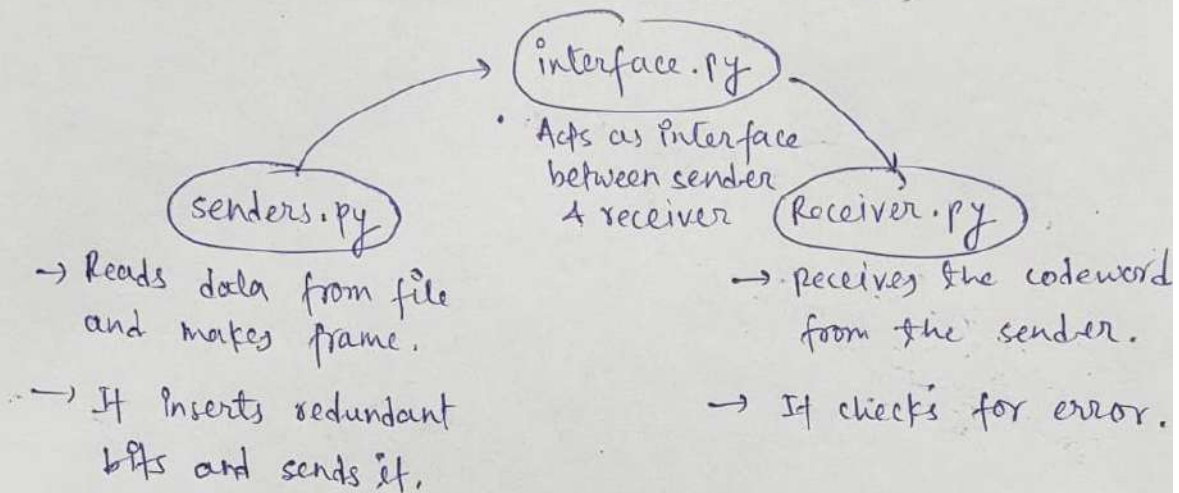
Design

I have solved the problem using 3 different sub problems:

1. Sending data
2. Receiving data
3. Interface between sender and receiver

Code Design

dataword + Redundant bits = codeword
(to be sent)



Attributes

- codeword
- frameSize

Functions

createCodeword()
OneDParityGenerator()
TwoDParityGenerator()
addBinary()
xor()
divideBinary()
Display()

Attributes

- codeword
- FrameSize

Functions

checkError()
Display()

1. senders.py(Task)

- The sequence of 0's and 1's are read from input file.
- This sequence is divided into datawords on the basis of frame size taken as the

input from the user.

- c. According to the four schemes namely VRC, LRC, Checksum and CRC, redundant bits/dataword are added along with the datawords to form codewords.
- d. The datawords and codewords which are to be sent are displayed.
- e. These encoded codewords are then sent to the receiver.

2. receiver.py

- a. The codewords are received from the sender.
- b. The received codewords are then decoded according to the four schemes namely VRC, LRC, Checksum and CRC.
- c. The result is checked and shown if there is any error detected.
- d. The codewords and the datawords extracted from the codewords are also displayed.

3. interface.py

- a. This program acts as an interface to the above programs – sender.py and receiver.py.
- b. In this program, the functions for injecting errors are included.
- c. There are two separate functions for injecting errors on random positions of codeword, and for injecting errors on specific positions taken as input.
- d. A filename is taken as input from command line while executing the program.
- e. This is then used as the input file which has a sequence of 0 and 1 stored in it.
The function invocations of Sender class stored in sender.py and Receiver class stored in receiver.py are done in this file to send and receive the data.
- f. The sent data are injected with errors using the above mentioned functions. For all the following three cases, three different functions have been created to execute a specific case at a moment.
 - i. Error is detected by all four scheme
 - ii. Error is detected by checksum but not CRC.
 - iii. Error is detected by VRC but not CRC.

Implementation

Sender:

```
# sender.py - The following are the tasks performed in this Sender program :
# 1. The input file is read, which contains a sequence of 0 and 1(From input.txt)
# 2. The message sequence is divided into datawords on the basis of frame size(k) taken as
#    the input from the user.
# 3. According to the four schemes namely VRC, LRC, Checksum and CRC, redundant
#    bits/dataword are added along with the datawords to form codewords.
#    datawords + redundant bits = codewords
# 4. The datawords and codewords which are to be sent are displayed.
# 5. These encoded codewords are then sent to the receiver.

class sender:
    # Initialize the sender class
    def __init__(self,size):
        self.codeword = []
        self.frame_size = size

    # Generate the codeword from the dataword according to the scheme
    def createCodeword(self,filename,schemeType,poly=""):
        fileinput=open(filename,"r")
        dataword=fileinput.readline()
        fileinput.close()
        tempword=""
        if schemeType ==1:
            for i in range(len(dataword)):
                if(i!=0 and i%self.frame_size==0):
                    self.codeword.append(tempword)
                    tempword=""
                    tempword+=dataword[i]
                self.codeword.append(tempword)
                self.OneDParityGenerator()
        elif schemeType ==2:
            for i in range(len(dataword)):
                if(i!=0 and i%self.frame_size==0):
                    self.codeword.append(tempword)
                    tempword=""
                    tempword+=dataword[i]
                self.codeword.append(tempword)
                self.TwoDParityGenerator()
        elif schemeType ==3:
            # Checksum
            for i in range(len(dataword)):
                if(i!=0 and i%self.frame_size==0):
                    self.codeword.append(tempword)
                    tempword=""
                    tempword+=dataword[i]
                self.codeword.append(tempword)

            checkSum=self.codeword[0]
```

```

        for i in range(1, len(self.codeword)):
            checksum=self.addBinary(checksum, self.codeword[i])
            while( len(checksum)>self.frame_size):
                a=checksum[: len(checksum)-self.frame_size]
                b=checksum[ len(checksum)-self.frame_size:]
                checksum=self.addBinary(a,b)
# Compliment of the checksum
finalChecksum=""
for j in range(len(checksum)):
    if(checksum[j]=='0'):
        finalChecksum+='1'
    else:
        finalChecksum+='0'
self.codeword.append(finalChecksum)
elif schemeType ==4:

    for i in range(len(dataword)):
        if i>0 and i%self.frame_size==0:
            tempword+='0'*(len(poly)-1)
            remainder=self.divideBinary(tempword, poly)
            remainder=remainder[ len(remainder)-( len(poly)-1):]
            tempword=tempword[:self.frame_size]
            tempword+=remainder
            self.codeword.append(tempword)
            tempword=""
            tempword+=dataword[i]
            tempword+='0'*(len(poly)-1)
            remainder=self.divideBinary(tempword, poly)
            remainder=remainder[ len(remainder)-( len(poly)-1):]
            tempword=tempword[:self.frame_size]
            tempword+=remainder
            self.codeword.append(tempword)

def OneDParityGenerator(self):
    for i in range(len(self.codeword)):
        countOnes=0
        for j in range(len(self.codeword[i])):
            if(self.codeword[i][j]=='1'):
                countOnes+=1
        if(countOnes%2==0):
            self.codeword[i]+='0'
        else:
            self.codeword[i]+='1'

def TwoDParityGenerator(self):

    parity=""
    index=0
    while index<self.frame_size:
        countOnes=0
        codeWordIndex=0;
        while codeWordIndex<len(self.codeword):
            if(self.codeword[codeWordIndex][index]=='1'):
                countOnes+=1

```

```

        codeWordIndex+=1
        if(countOnes%2==0):
            parity+='0'
        else:
            parity+='1'
        index+=1
    self.codeword.append(parity)
# Add the two binary numbers of the same length
def addBinary(self,a,b):
    result=""
    a=a[::-1]
    b=b[::-1]
    carry=0

    for i in range(len(a)):
        DigitA=ord(a[i])-ord('0')
        DigitB=ord(b[i])-ord('0')
        total=DigitA+DigitB+carry

        char=str(total%2)
        result=char+result
        carry=total//2

    if carry==1:
        result='1'+result
    return result

def xor(self, a, b):
    result = ""
    for i in range(1, len(b)):
        if a[i]==b[i]:
            result += '0'
        else:
            result += '1'
    return result

#Helper function to divide two binary sequence
def divideBinary(self, dividend, divisor):
    xorlen = len(divisor)
    temp = dividend[:xorlen]
    while len(dividend) > xorlen:
        if temp[0]=='1':
            temp=self.xor(divisor,temp)+dividend[xorlen]
        else:
            temp=self.xor('0'*xorlen,temp)+dividend[xorlen]
        xorlen += 1
    if temp[0]=='1':
        temp=self.xor(divisor,temp)
    else:
        temp=self.xor('0'*xorlen,temp)
    return temp

def Display(self,schemeType):
    dataword=[]
    if schemeType ==1:
        print("VRC Scheme")
        print("Dataword : ",end="")
        for i in self.codeword:
            dataword.append(i[:self.frame_size])

```

```

        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
    elif schemeType ==2:
        parity=self.codeword[len(self.codeword)-1]
        print("LRC Scheme")
        print("Dataword : ",end="")
        for i in range(len(self.codeword)-1):
            dataword.append(self.codeword[i])
        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
        print("Parity : ",end="")
        print(parity)
    elif schemeType ==3:
        checksum=self.codeword[len(self.codeword)-1]
        # push all elements except the checksum
        for i in range(len(self.codeword)-1):
            dataword.append(self.codeword[i])
        print("Checksum Scheme")
        print("Dataword : ",end="")
        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
        print("Checksum : ",end="")
        print(checksum)
    elif schemeType ==4:
        print("CRC Scheme")
        print("Dataword : ",end="")
        for i in range(len(self.codeword)):
            dataword.append(self.codeword[i])
        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)

```

Receiver:

```

class receiver:
    def __init__ (self,s):
        self.codeword=s.codeword
        self.frame_size=s.frame_size

    def checkError(self,schemeType,poly=""):
        if schemeType ==1:
            self.OneDParityCheck()
        elif schemeType ==2:
            self.TwoDParityCheck()
        elif schemeType ==3:
            self.checksumCheck()
        elif schemeType ==4:
            self.checkCRCError(poly)
        else:

```

```

        print("Invalid Scheme")

def checkCRCError(self,poly):
    for i in range(len(self.codeword)):
        remainder = self.divideBinary(self.codeword[i], poly)
        error = False
        for j in range(len(remainder)):
            if remainder[j] == '1':
                error = True
        print("Remainder:",remainder,end=' ')
        if error:
            print("ERROR DETECTED")
        else:
            print("NO ERROR DETECTED")
        # exit the loop if error is detected

def OneDParityCheck(self):

    flag=True
    for i in range(len(self.codeword)):
        countOnes=0
        for j in range(len(self.codeword[i])):
            if(self.codeword[i][j]=='1'):
                countOnes+=1
        if(countOnes%2!=0):
            print("ERROR is detected by VRC")
            flag=False
            break
    if flag==True:
        print("No Error is detected by VRC")

def TwoDParityCheck(self):
    parity=""
    index=0
    while index<self.frame_size:
        countOnes=0
        codeWordIndex=0;
        while codeWordIndex<len(self.codeword)-1:
            if(self.codeword[codeWordIndex][index]=='1'):
                countOnes+=1
            codeWordIndex+=1
        if(countOnes%2==0):
            parity+='0'
        else:
            parity+='1'
        index+=1
    if(parity==self.codeword[len(self.codeword)-1]):
        print("No Error is detected by LRC")
    else:
        print("ERROR is detected by LRC")
    return parity

def checksumCheck(self):
    checkSum=self.codeword[0]
    for i in range(1,len(self.codeword)):

```

```

        checksum=self.addBinary(checksum,self.codeword[i])
    while(len(checksum)>self.frame_size):
        a=checksum[:len(checksum)-self.frame_size]
        b=checksum[len(checksum)-self.frame_size:]
        checksum=self.addBinary(a,b)
    # Compliment of the checksum
    finalChecksum=""
    for j in range(len(checksum)):
        if(checksum[j]=='0'):
            finalChecksum+='1'
        else:
            finalChecksum+='0'
    #convert the finalChecksum to int
    val=int(finalChecksum,2)
    if(val==0):
        print("No Error is detected by Checksum")
    else:
        print("ERROR is detected by Checksum")
def addBinary(self,a,b):
    result=""
    a=a[::-1]
    b=b[::-1]
    carry=0

    for i in range(max(len(a),len(b))):
        DigitA=ord(a[i])-ord('0') if i<len(a) else 0
        DigitB=ord(b[i])-ord('0') if i<len(b) else 0
        total=DigitA+DigitB+carry

        char=str(total%2)
        result=char+result
        carry=total//2

    if carry==1:
        result='1'+result
    return result
def Display(self,schemeType):
    # display the codeword
    dataword=[]
    if schemeType ==1:
        print("VRC Scheme")
        print("Dataword : ",end="")
        for i in self.codeword:
            dataword.append(i[:self.frame_size])

        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
    elif schemeType ==2:
        parity=self.codeword[len(self.codeword)-1]
        print("LRC Scheme")
        print("Dataword : ",end="")
        for i in range(len(self.codeword)-1):
            dataword.append(self.codeword[i])
        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
        print("Parity : ",end="")

```

```

        parity=""
        index=0
        while index<self.frame_size:
            countOnes=0
            codewordIndex=0;
            while codewordIndex<len(self.codeword)-1:
                if(self.codeword[codewordIndex][index]=='1'):
                    countOnes+=1
                    codewordIndex+=1
            if(countOnes%2==0):
                parity+='0'
            else:
                parity+='1'
            index+=1
        print(parity)

    elif schemeType ==3:
        checksum=self.codeword[len(self.codeword)-1]
        # push all elements except the checksum
        for i in range(len(self.codeword)-1):
            dataword.append(self.codeword[i])
        print("Checksum Scheme")
        print("Dataword : ",end="")
        print(dataword)
        print("Codeword : ",end="")
        print(self.codeword)
        print("Checksum : ",end="")
        print(checksum)

#Helper function to divide two binary sequence
def divideBinary(self, dividend, divisor):
    xorlen = len(divisor)
    temp = dividend[:xorlen]
    while len(dividend) > xorlen:
        if temp[0]=='1':
            temp=self.xor(divisor,temp)+dividend[xorlen]
        else:
            temp=self.xor('0'*xorlen,temp)+dividend[xorlen]
        xorlen += 1
    if temp[0]=='1':
        temp=self.xor(divisor,temp)
    else:
        temp=self.xor('0'*xorlen,temp)
    return temp
def xor(self, a, b):
    result = ""
    for i in range(1, len(b)):
        if a[i]==b[i]:
            result += '0'
        else:
            result += '1'
    return result

```

Interface.py

```

from senders import *
from receiver import *
import random
import sys

#function to inject errors in random positions
def injectRandomError(frames):
    for i in range(len(frames)):
        pos = random.randint(0, len(frames[i])-1)
        frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames

def injectSpecificError(frames, zeropos, onepos):
    for i in range(len(zeropos)):
        for j in range(len(zeropos[i])):
            pos = zeropos[i][j]
            frames[i] = frames[i][:pos]+'0'+frames[i][pos+1:]
    for i in range(len(onepos)):
        for j in range(len(onepos[i])):
            pos = onepos[i][j]
            frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames

def case1(filename,size):
    # changing 1 bit in the codeword
    print("CASE 1:")
    print(" VRC ")
    type=1
    s=sender(size)
    s.createCodeword(filename,type)
    s.Display(type)
    s.codeword=injectRandomError(s.codeword)
    r=receiver(s)
    r.checkError(type)
    r.Display(type)

    print(" LRC ")
    type=2
    s=sender(size)
    s.createCodeword(filename,type)
    s.Display(type)
    s.codeword=injectRandomError(s.codeword)
    r=receiver(s)
    r.checkError(type)
    r.Display(type)

    print("Checksum")
    type=3
    s=sender(size)
    s.createCodeword(filename,type)
    s.Display(type)
    s.codeword=injectRandomError(s.codeword)
    r=receiver(s)
    r.checkError(type)
    r.Display(type)

```

```

    print("CRC")
    poly="1001"
    type=4
    s=sender(size)
    s.createCodeword(filename, type, poly)
    s.Display(type)
    s.codeword=injectRandomError(s.codeword)
    r=receiver(s)
    r.checkError(type, poly)
    r.Display(type)

def case2(filename, size):
    print("Case 2")
    print("Checksum")
    type=3

    zeropostion=[]
    oneposition=[[5]]
    s=sender(size)
    s.createCodeword(filename, type)
    s.Display(type)
    s.codeword=injectSpecificError(s.codeword, zeropostion, oneposition)
    r=receiver(s)
    r.checkError(type)
    r.Display(type)

    print("CRC")
    poly="100"
    type=4
    s=sender(size)
    s.createCodeword(filename, type, poly)
    s.Display(type)
    s.codeword=injectSpecificError(s.codeword, zeropostion, oneposition)
    r=receiver(s)
    r.checkError(type, poly)
    r.Display(type)

def case3(filename, size):
    print("Case 3")
    print("VRC")
    type=1

    zeropostion=[]
    oneposition=[[5]]
    s=sender(size)
    s.createCodeword(filename, type)
    s.Display(type)
    s.codeword=injectSpecificError(s.codeword, zeropostion, oneposition)
    r=receiver(s)
    r.checkError(type)
    r.Display(type)

    print("CRC")
    poly="100"
    type=4
    s=sender(size)
    s.createCodeword(filename, type, poly)
    s.Display(type)
    s.codeword=injectSpecificError(s.codeword, zeropostion, oneposition)

```

```

r=receiver(s)
r.checkError(type,poly)
r.Display(type)

if __name__=='__main__':
    print("1. Error is Detected by All Schemes")
    print("2. Error is Detected by Checksum but not CRC ")
    print("3. Error is Detected by VRC but not by CRC")
    case=int(input())
    print("Enter file name ")
    filename=input()
    print("Enter size of frame")
    frameSize=int(input())

    if case==1:
        case1(filename,frameSize)
    elif case==2:
        case2(filename,frameSize)
    elif case==3:
        case3(filename,frameSize)

```

Test Cases

We had to check 3, cases as per as the question.

- Error is Detected by All Schemes
 - Here I have injected a random error at a bit position, which is detected by all the schemes.
- Error is Detected by Checksum but not CRC
 - As we know that if the degree of the dividing polynomial is less than the degree of the codeword and the codeword is also divisible by the polynomial then the CRC cannot detect error.
 - So, in the implementation I have designed a test case that checksum and easily detect but due to the choice of polynomial, CRC Couldn't detect it.
- Error is Detected by VRC but not by CRC

- It is similar to the above test case used in point 2.

Results

Here, all the test cases have been checked.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER zsh
thebikashsah@BIKASHs-MacBook-Air Computer Network % python3 interface.py
-----
1. Error is detected by all four schemes.
2. Error is detected by checksum but not by CRC.
3. Error is detected by VRC but not by CRC.
Enter Case Number : 1
Input file name:
input.txt
Enter length of the dataword: 4
CASE 1
VRC
Datawords to be sent:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
Codewords sent by sender:
['00011', '00000', '11101', '11101', '10010', '01001', '10100', '00011']

Parity is odd. ERROR DETECTED
Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED

Codewords received by receiver:
['01011', '00010', '11101', '11101', '11010', '01001', '10100', '00011']
Extracting datawords from codewords:
['0101', '0001', '1110', '1110', '1101', '0100', '1010', '0001']

LRC
Datawords to be sent:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
Codewords sent by sender:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
0111 - parity
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER zsh

```

Parity did not match. ERROR DETECTED

Codewords received by receiver:
['0001', '0100', '1111', '1110', '1001', '1100', '1010', '0101']
1110 - parity
Extracting datawords from codewords:
['0001', '0100', '1111', '1110', '1001', '1100', '1010', '0101']

Checksum
Datawords to be sent:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
Codewords sent by sender:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
0111 - checksum

Complement is not zero. ERROR DETECTED

Codewords received by receiver:
['1001', '0100', '1110', '1110', '1001', '0101', '1010', '0001']
0110 - sum
1101 - sum+checksum
0010 - complement
Extracting datawords from codewords:
['1001', '0100', '1110', '1110', '1001', '0101', '1010', '0001']

CRC
Enter Generator Polynomial: 1010
Datawords to be sent:
['0001', '0000', '1110', '1110', '1001', '0100', '1010', '0001']
Codewords sent by sender:
['0001010', '0000000', '1110010', '1110010', '1001110', '0100010', '1010000', '0001010']

Remainder: 000 NO ERROR DETECTED
Remainder: 100 ERROR DETECTED
Remainder: 001 ERROR DETECTED

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER zsh

Remainder: 000 NO ERROR DETECTED
Remainder: 000 NO ERROR DETECTED
Remainder: 000 NO ERROR DETECTED
Remainder: 010 ERROR DETECTED
Remainder: 100 ERROR DETECTED

Codewords received by receiver:
['0001010', '1000000', '1110011', '1110010', '1001110', '0100010', '1011000', '1001010']
Extracting datawords from codewords:
['0001', '1000', '1110', '1110', '1001', '0100', '1011', '1001']

● thebikashsah@BIKASHs-MacBook-Air Computer Network % python3 interface.py
-----
1. Error is detected by all four schemes.
2. Error is detected by checksum but not by CRC.
3. Error is detected by VRC but not by CRC.
Enter Case Number : 2
Input file name:
input.txt
CASE 2
Checksum
Datawords to be sent:
['00010000', '11101110', '10010100', '10100001']
Codewords sent by sender:
['00010000', '11101110', '10010100', '10100001']
11001010 - checksum

Complement is not zero. ERROR DETECTED

Codewords received by receiver:
['00010100', '11101110', '10010100', '10100001']
00111001 - sum
00000100 - sum+checksum
11111011 - complement
Extracting datawords from codewords:
['00010100', '11101110', '10010100', '10100001']
```

CRC

Datawords to be sent:

['00010000', '11101110', '10010100', '10100001']

Codewords sent by sender:

['00010000000', '11101110000', '10010100000', '10100001000']

Remainder: 000 NO ERROR DETECTED

Remainder: 000 NO ERROR DETECTED

Remainder: 000 NO ERROR DETECTED

Remainder: 000 NO ERROR DETECTED

Codewords received by receiver:

['00010100000', '11101110000', '10010100000', '10100001000']

Extracting datawords from codewords:

['00010100', '11101110', '10010100', '10100001']

● thebikashsah@BIKASHs-MacBook-Air Computer Network % python3 interface.py

1. Error is detected by all four schemes.

2. Error is detected by checksum but not by CRC.

3. Error is detected by VRC but not by CRC.

Enter Case Number : 3

Input file name:

input.txt

CASE 3

VRC

Datawords to be sent:

['000100', '001110', '111010', '010100', '101000', '011']

Codewords sent by sender:

['0001001', '0011101', '1110100', '0101000', '1010000', '011']

Parity is odd. ERROR DETECTED

Parity is even. NO ERROR DETECTED

Parity is even. NO ERROR DETECTED

Parity is even. NO ERROR DETECTED

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER zsh

Codewords sent by sender:
['0001001', '0011101', '1110100', '0101000', '1010000', '011']

Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED

Codewords received by receiver:
['0001011', '0011101', '1110100', '0101000', '1010000', '011']
Extracting datawords from codewords:
['000101', '001110', '111010', '010100', '101000', '011']

CRC
Datawords to be sent:
['000100', '001110', '111010', '010100', '101000', '010000']
Codewords sent by sender:
['00010000', '00111000', '11101000', '01010000', '10100000', '010000']

Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED

Codewords received by receiver:
['00010100', '00111000', '11101000', '01010000', '10100000', '010000']
Extracting datawords from codewords:
['000101', '001110', '111010', '010100', '101000', '010000']

-----
○ thebikashsah@BIKASHs-MacBook-Air Computer Network %
```

Analysis

Analysis of senders.py

- `createCodeword(self, filename, schemeType, poly="")`

We take input as the filename type(type of technique) and the polynomial if it is CRC.

Here we read the entire sequence of 0's and 1's from the file and then divide the

sequence into data frames of the required length that is also given by the user.

After dividing into frames we append each message with the redundant bits receptive to

the type of error technique;

- `Display(self, schemeType)`
This method simply displays the frames with the corresponding redundant bits on console.
- `OneDParityGenerator(self)`
This method is used to generate parity in case of VRC.
- `TwoDParityGenerator(self)`
This method is used to generate parity in case of LRC.
- `addBinary(self, a, b)`
This method has been used as a helper function for implementing Checksum. It takes two binary sequence as parameters, and returns the result of wrapped addition of them.
- `xor(self, a, b)`
This method has been used as a helper function for implementing CRC. In this method, two binary sequence is taken as input from its arguments, and return the xor of them.
- `divideBinary(self, dividend, divisor)`
This method has been used as a helper function for implementing CRC. In this method, dividend and divisor is taken as input from its arguments. The division is performed using mod 2 arithmetic (exclusive-OR) on the message using divisor polynomial, and the remainder is returned.

Analysis of receivers.py

- `checkError(self, schemeType, poly="")`
This method takes type (scheme) of error detection method as one of its argument. On the basis of the scheme the codewords received is checked for error. If there is an error, appropriate message is displayed on the screen.

Rest functions are similar to senders.py

Analysis of interface.py

- `injectRandomError(frames)`

This method is used to inject errors in random positions of the codewords which are sent by sender program. The `randint()` function of `random` python module has been used for generating random position within the size of the codewords.

- `injectSpecificError(frames, zeropos, onepos)`

This method is used to inject errors in specific positions of the codewords. The positions in which the error is to be inserted is passed as arguments, as `zeropos` and `onepos`. `Zeropos` contains the list of positions in which the value of those positions will be made 0, whereas `Onepos` contains the list of positions in which the value of those positions will be made 1.

- `case1(filename, size)`

This method is implemented to make function invocations and displaying the results of all schemes, considering the case 1 of the problem statement (Error is detected by all four schemes). The dataword size and input file name is taken as function arguments.

- `case2(filename, size)`

This method is implemented to make function invocations and displaying the results of Checksum and CRC, such that the case 2 of the problem statement satisfies (Error is detected by Checksum but not CRC). The dataword size and input file name is taken as function arguments.

- `case3(filename, size)`

This method is implemented to make function invocations and displaying the results of VRC and CRC, such that the case 3 of the problem statement satisfies (Error is detected by VRC but not CRC). The dataword size and input file name is taken as function arguments.

Comments

This assignment was like a real life project, it had a problem statement and I had to come up with a solution, I have learnt a lot,

I learnt a lot of python in this assignment and also learnt how to implement big problems by dividing it into subproblems.
