Computer Networks Lab Report – Assignment 1

TITLE

Name – Sourav Dutta

Roll – 001610501076

Class – BCSE 3rd year

Group – A3

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.

(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.]

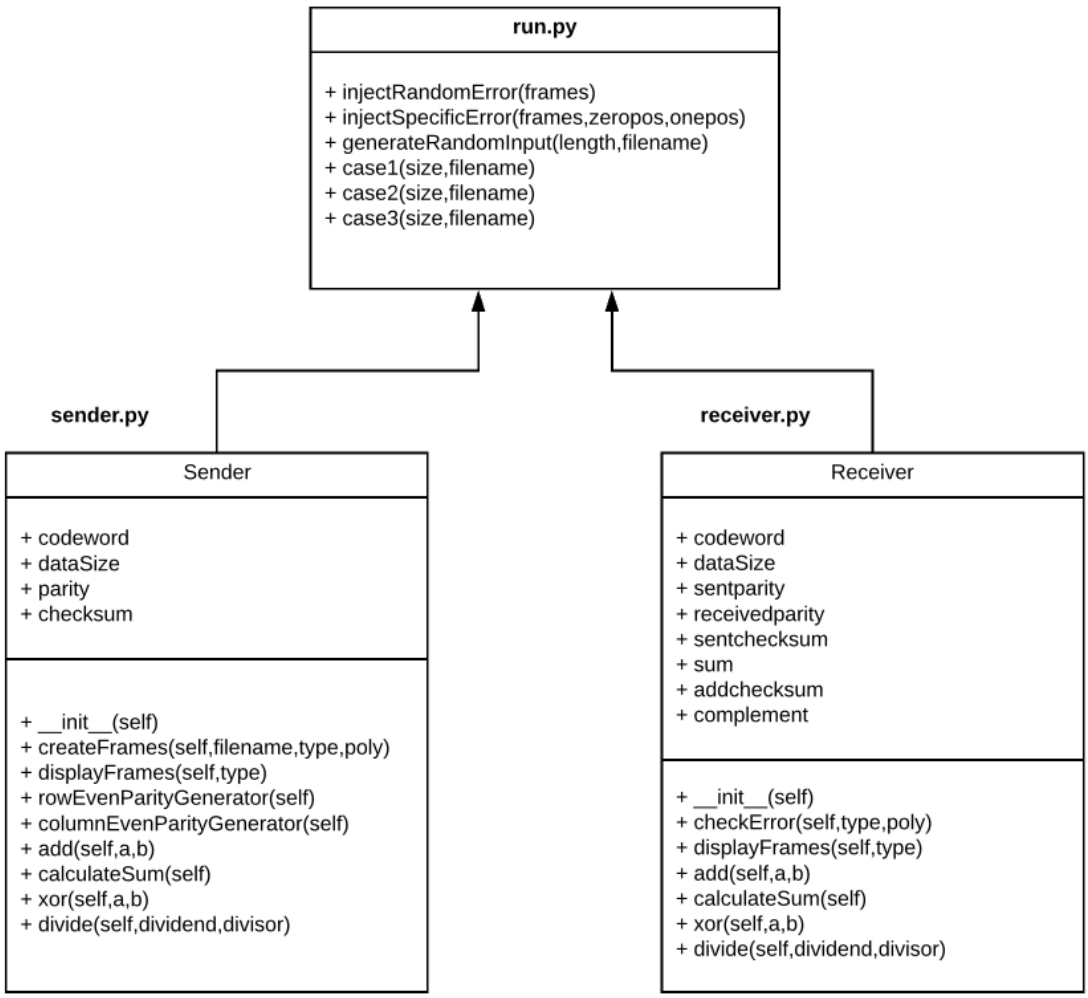
Submission date – 01/02/2019
Deadline – 01/02/2019

DESIGN

A bit, on travel, is subjected to electromagnetic (optical) interference due to noise signals (light sources). Thus, the data transmitted may be prone to errors. In particular, a bit '0' (bit '1') sent by the sender may be delivered as a bit '1' (bit '0') at the receiver. This happens because the voltage present in noise signals either has direct impact on the voltage present in the data signal or creates a distortion leading to mis-interpretation of bits while decoding the signals at the receiver. This calls for a study on error detection and error correction. The receiver must be intelligent enough to detect the error and ask the sender to transmit the data packet or must have the ability to detect and correct the errors.

In this assignment, we shall discuss the following error detection techniques in detail.

1. Vertical Redundancy check
2. Longitudinal Redundancy check
3. Checksum
4. Cyclic Redundancy check



I have implemented the error detection module in three program files.

- **sender.py** (Sender program)
- **receiver.py** (Receiver program)
- **run.py** (Program to act as an interface to sender and receiver)

The individual files fulfils different assignment purposes, following which have been explained in details :

1. **sender.py** – The following are the tasks performed in this Sender program :
    a. The input file is read, which contains a sequence of 0 and 1.
    b. The message 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** – The following are the tasks performed in this Receiver program :
    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. **run.py** – The following are the tasks performed in this program :
    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.
    f. 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.
    g. The sent data are injected with errors using the above mentioned functions.
    h. 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 schemes.
        ii. Error is detected by checksum but not CRC.
        iii. Error is detected by VRC but not CRC.

IMPLEMENTATION

Code Snippet of sender.py:

```
class Sender:
        #Initialize all the data members of class
        def __init__(self, size):
                self.codeword = []
                self.dataSize = size
                self.parity = ""
                self.checksum = ""

        #Function to generate codewords to be sent
        def createFrames(self, filename, type, poly=""):
```

```
fileinput = open(filename, "r")
packet = fileinput.readline()
fileinput.close()
tempword = ""
if type == 1:
	for i in range(len(packet)):
		if i>0 and i%self.dataSize==0:
			self.codeword.append(tempword)
			tempword = ""
		tempword += packet[i]
	self.codeword.append(tempword)
	self.rowEvenParityGenerator()
elif type == 2:
	for i in range(len(packet)):
		if i>0 and i%self.dataSize==0:
			self.codeword.append(tempword)
			tempword = ""
		tempword += packet[i]
	self.codeword.append(tempword)
	self.columnEvenParityGenerator()
elif type == 3:
	for i in range(len(packet)):
		if i>0 and i%self.dataSize==0:
			self.codeword.append(tempword)
			tempword = ""
		tempword += packet[i]
	self.codeword.append(tempword)

	summ = self.calculateSum()
	for i in range(len(summ)):
		if summ[i]=='0':
			self.checksum += '1'
		else:
			self.checksum += '0'
elif type == 4:
	tempsize = self.dataSize #- (len(poly)-1)
	for i in range(len(packet)):
		if i>0 and i%tempsize==0:
			tempword += '0'*(len(poly)-1)
			remainder = self.divide(tempword, poly)
			remainder = remainder[len(remainder)-(len(poly)-1):]
			tempword = tempword[:tempsize]
			tempword += remainder
			self.codeword.append(tempword)
			tempword = ""
		tempword += packet[i]

	tempword += '0'*(len(poly)-1)
	remainder = self.divide(tempword, poly)
	remainder = remainder[len(remainder)-(len(poly)-1):]
	tempword = tempword[:tempsize]
	tempword += remainder
	self.codeword.append(tempword)
```

```
#Function to display the sent codewords
def displayFrames(self, type):
	datawords = []
	for x in self.codeword:
		datawords.append(x[:self.dataSize])
	print("Datawords to be sent:")
	print(datawords)
	print("Codewords sent by sender:")
	print(self.codeword)
	if type == 2:
		print(self.parity, " - parity")
	elif type == 3:
		print(self.checksum, " - checksum")
	print("\n")

#Helper function for VRC Even Parity generator
def rowEvenParityGenerator(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==1:
			self.codeword[i] += '1'
		else:
			self.codeword[i] += '0'

#Helper function for LRC Even Parity generator
def columnEvenParityGenerator(self):
	i=0
	while i<self.dataSize:
		countOnes = 0
		j=0
		while j<len(self.codeword):
			if self.codeword[j][i] == '1':
				countOnes += 1
			j+=1
		if countOnes%2==1:
			self.parity += '1'
		else:
			self.parity += '0'
		i+=1

#Helper function to add two binary sequence
def add(self, a, b):
	result = ""
	s = 0
	i = len(a)-1
	j = len(b)-1
	while i>=0 or j>=0 or s==1:
		if i>=0:
			s+=int(a[i])
```

```
                if j>=0:
                        s+=int(b[j])
                result = str(s%2) + result
                s //= 2
                i-=1
                j-=1
        return result

#Helper function to calculate sum of all codewords
def calculateSum(self):
        result = self.codeword[0]
        for i in range(1,len(self.codeword)):
                result = self.add(result, self.codeword[i])
                while len(result) > self.dataSize:
                        t1 = result[:len(result)-self.dataSize]
                        t2 = result[len(result)-self.dataSize:]
                        result = self.add(t1, t2)
        return result

#Helper function to XOR two binary sequence
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 divide(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
```

Method descriptions of sender.py:

- createFrames(self, filename, type, poly=””) : This method is used to take input of sequence of 0,1 from a given file, accessing it using the filename passed as argument. Then the packet is divided into datawords of fixed size. According to the type (scheme) of error detection, namely VRC, LRC, Checksum, CRC, the datawords are appended with appropriate redundant bits.

- displayFrames(self,type) : This method has been used for displaying the datawords and codewords which are to be sent.

- rowEvenParityGenerator(self) : This method has been used as a helper function for VRC. In this method, for each dataword, a parity bit is generated such that the parity of the word is even. This parity bit is then appended to dataword to form a codeword.

- columnEvenParityGenerator(self) : This method has been used as a helper function for LRC. In this method, for every column of all dataword, corresponding parity bit is generator such that the parity becomes even in that column. As a result, a parity sequence is appended along with codewords.

- add(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.

- calculateSum(self) : This method has been used as a helper function for implementing Checksum. It calculates the sum of all the datawords (using the above mentioned function) and returns the result.

- 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.

- divide(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 (exclusice-OR) on the message using divisor polynomial, and the remainder is returned.

Code Snippet of receiver.py:

```
class Receiver:
	#Initialize all the data members of class
	def __init__(self, s):
		self.codeword = s.codeword
		self.dataSize = s.dataSize
		self.sentparity = s.parity
		self.receivedparity = ""
		self.sentchecksum = s.checksum
		self.sum = ""
		self.addchecksum = ""
		self.complement = ""

	#Function to decode and check for error in codewords
	def checkError(self, type, poly=""):
		if type == 1:
			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("Parity is even.", end=' ')
                        print("NO ERROR DETECTED")
                else:
                        print("Parity is odd.", end=' ')
                        print("ERROR DETECTED")
        elif type == 2:
                error = False
                i=0
                while i<self.dataSize:
                        countOnes = 0
                        j=0
                        while j<len(self.codeword):
                                if self.codeword[j][i] == '1':
                                        countOnes += 1
                                j+=1
                        if countOnes%2==1:
                                ch = '1'
                        else:
                                ch = '0'
                        self.receivedparity += ch
                        if ch != self.sentparity[i]:
                                error = True
                        i+=1
                if error:
                        print("Parity did not match.",end=' ')
                        print("ERROR DETECTED")
                else:
                        print("Parity matched.",end=' ')
                        print("NO ERROR DETECTED")
        elif type == 3:
                self.sum = self.calculateSum()
                result = self.add(self.sum, self.sentchecksum)
                while len(result) > self.dataSize:
                        t1 = result[:len(result)-self.dataSize]
                        t2 = result[len(result)-self.dataSize:]
                        result = self.add(t1, t2)
                self.addchecksum = result

                #finding complement and checking error
                error = False
                for ch in self.addchecksum:
                        if ch == '0':
                                self.complement += '1'
                                error = True
                        else:
                                self.complement += '0'
                if error:
                        print("Complement is not zero.",end=' ')
                        print("ERROR DETECTED")
                else:
```

```
                        print("Complement is zero.",end=' ')
                        print("NO ERROR DETECTED")
        elif type == 4:
                for i in range(len(self.codeword)):
                        remainder = self.divide(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")
        print()

#Function to display the received codewords
def displayFrames(self, type):
        print("Codewords received by receiver:")
        print(self.codeword)
        if type == 2:
                print(self.receivedparity, " - parity")
        elif type == 3:
                print(self.sum, " - sum")
                print(self.addchecksum, " - sum+checksum")
                print(self.complement, " - complement")
        for i in range(len(self.codeword)):
                self.codeword[i] = self.codeword[i][:self.dataSize]
        print("Extracting datawords from codewords:")
        print(self.codeword)
        print()

#Helper function to add two binary sequence
def add(self, a, b):
        result = ""
        s = 0
        i = len(a)-1
        j = len(b)-1
        while i>=0 or j>=0 or s==1:
                if i>=0:
                        s+=int(a[i])
                if j>=0:
                        s+=int(b[j])
                result = str(s%2) + result
                s //= 2
                i-=1
                j-=1
        return result

#Helper function to calculate sum of all codewords
def calculateSum(self):
        result = self.codeword[0]
        for i in range(1,len(self.codeword)):
```

```
                result = self.add(result, self.codeword[i])
                while len(result) > self.dataSize:
                        t1 = result[:len(result)-self.dataSize]
                        t2 = result[len(result)-self.dataSize:]
                        result = self.add(t1, t2)
        return result

#Helper function to XOR two binary sequence
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 divide(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
```

Method descriptions of receiver.py:

- checkError(self,type,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.

- displayFrames(self,type) : This method has been used to display the received codewords and datawords from the sender.

- add(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.

- calculateSum(self) : This method has been used as a helper function for implementing Checksum. It calculates the sum of all the datawords (using the above mentioned function) and returns the result.

- 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.

- divide(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 (exclusice-OR) on the message using divisor polynomial, and the remainder is returned.

Code Snippet of run.py:

```
from sender 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

#function to inject errors in specific positions
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

#function to generate random sequence of 0,1 and store it in a file
def generateRandomInput(length, filename):
	fileout = open(filename, "w")
	for i in range(length):
		fileout.write(str(random.randint(0,1)))
	fileout.close()

#function to execute Case 1
def case1(size, filename):
	puterror = True
	print("------------------------- CASE 1 -------------------------")
	print("Error is detected by all four schemes.")
	print("-------------- Vertical Redundancy Check --------------------")
	type = 1
	s = Sender(size)
	s.createFrames(filename,type)
	s.displayFrames(type)
```

```
        if puterror:
                s.codeword = injectRandomError(s.codeword)
        r = Receiver(s)
        r.checkError(type)
        r.displayFrames(type)
        print("---------- Longitudinal Redundancy Check ----------")
        type = 2
        s = Sender(size)
        s.createFrames(filename,type)
        s.displayFrames(type)
        if puterror:
                s.codeword = injectRandomError(s.codeword)
        r = Receiver(s)
        r.checkError(type)
        r.displayFrames(type)
        print("---------- Checksum ----------")
        type = 3
        s = Sender(size)
        s.createFrames(filename,type)
        s.displayFrames(type)
        if puterror:
                s.codeword = injectRandomError(s.codeword)
        r = Receiver(s)
        r.checkError(type)
        r.displayFrames(type)
        print("---------- Cyclic Redundancy Check ----------")
        type = 4
        #poly = "1001"
        #print("Generator polynomial:",poly)
        poly = input("Enter Generator Polynomial: ")
        s = Sender(size)
        s.createFrames(filename,type,poly)
        s.displayFrames(type)
        if puterror:
                s.codeword = injectRandomError(s.codeword)
        r = Receiver(s)
        r.checkError(type,poly)
        r.displayFrames(type)
        print("----------\n")

#function to execute Case 2
def case2(size, filename):
        print("---------- CASE 2 ----------")
        print("Error is detected by checksum but not by CRC.")
        print("---------- Checksum ----------")
        type = 3
        s = Sender(size)
        s.createFrames(filename,type)
        s.displayFrames(type)
        zeropos = []
        onepos = [[5]]
        s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
        r = Receiver(s)
```

```
        r.checkError(type)
        r.displayFrames(type)
        print("---------- Cyclic Redundancy Check ----------")
        type = 4
        poly = "1000"
        print("Generator Polynomial:",poly)
        s = Sender(size)
        s.createFrames(filename,type,poly)
        s.displayFrames(type)
        s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
        r = Receiver(s)
        r.checkError(type,poly)
        r.displayFrames(type)
        print("------------------------------------------------------------\n")

#function to execute Case 3
def case3(size, filename):
        print("-------------------------- CASE 3 -------------------------")
        print("Error is detected by VRC but not by CRC.")
        print("---------- Vertical Redundancy Check ----------")
        type = 1
        s = Sender(size)
        s.createFrames(filename,type)
        s.displayFrames(type)
        zeropos = []
        onepos = [[5]]
        s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
        r = Receiver(s)
        r.checkError(type)
        r.displayFrames(type)
        print("---------- Cyclic Redundancy Check ----------")
        type = 4
        poly = "100"
        print("Generator Polynomial:",poly)
        s = Sender(size)
        s.createFrames(filename,type,poly)
        s.displayFrames(type)
        s.codeword = injectSpecificError(s.codeword,zeropos,onepos)
        r = Receiver(s)
        r.checkError(type,poly)
        r.displayFrames(type)
        print("------------------------------------------------------------\n")

#driver function to run the error detection module
if __name__ == "__main__":
        print("------------------------------------------------------------")
        print("1. Error is detected by all four schemes.")
        print("2. Error is detected by checksum but not by CRC.")
        print("3. Error is detected by VRC but not by CRC.")
        case = int(input("Enter Case Number : "))
        if case == 1:
                size = int(input("Enter length of the dataword: "))
                generateRandomInput(size*4)
```

```
        case1(size, sys.argv[1])
elif case == 2:
        size = 8
        case2(size, sys.argv[1])
elif case == 3:
        size = 6
        case3(size, sys.argv[1])
else:
        print("You entered invalid choice.")
```

Method descriptions of run.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 containes the list of positions in which the value of those positions will be made 1.

- case1(size,filename) : 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(size,filename) : 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(size,filename) : 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.

TEST CASES

```
C:\Users\SOURAV\Desktop\comp-networks-lab\ass1>python prog.py random-input.txt
------------------------------------------------------------
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
Enter length of the dataword: 8
-------------------------- CASE 1 ------------------------
Error is detected by all four schemes.
------------- Vertical Redundancy Check ------------------
Datawords to be sent:
['00010000', '11101110', '10010100', '10100001']
Codewords sent by sender:
['000100001', '111011100', '100101001', '101000011']


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:
['010100001', '111011100', '100101001', '101000011']
Extracting datawords from codewords:
['01010000', '11101110', '10010100', '10100001']

------------- Longitudinal Redundancy Check --------------
Datawords to be sent:
['00010000', '11101110', '10010100', '10100001']
Codewords sent by sender:
['00010000', '11101110', '10010100', '10100001']
11001011  - parity


Parity did not match. ERROR DETECTED

Codewords received by receiver:
['00010000', '11101110', '10010100', '10101001']
11000011  - parity
Extracting datawords from codewords:
['00010000', '11101110', '10010100', '10101001']
------------- 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:
['00011000', '11101110', '10010110', '10100001']
00111111  - sum
00001010  - sum+checksum
11110101  - complement
Extracting datawords from codewords:
['00011000', '11101110', '10010110', '10100001']

------------- Cyclic Redundancy Check --------------------
Enter Generator Polynomial: 111010101
Datawords to be sent:
['00010000', '11101110', '10010100', '10100001']
Codewords sent by sender:
['0001000001010010', '1110111001111110', '1001010001000011', '1010000110011110']


Remainder: 00000100 ERROR DETECTED
Remainder: 00000000 NO ERROR DETECTED
Remainder: 00000000 NO ERROR DETECTED
Remainder: 00000000 NO ERROR DETECTED

Codewords received by receiver:
['0001000001010110', '1110111001111110', '1001010001000011', '1010000110011110']
Extracting datawords from codewords:
['00010000', '11101110', '10010100', '10100001']
```

```
C:\Users\SOURAV\Desktop\comp-networks-lab\ass1>python run.py case2-sender-input.txt
------------------------------------------------------------
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
-------------------------- CASE 2 ------------------------
Error is detected by checksum but not by CRC.
-------------- Checksum ------------------------------------
Datawords to be sent:
['10100001']
Codewords sent by sender:
['10100001']
01011110  - checksum


Complement is not zero. ERROR DETECTED

Codewords received by receiver:
['10100101']
10100101  - sum
00000100  - sum+checksum
11111011  - complement
Extracting datawords from codewords:
['10100101']

-------------- Cyclic Redundancy Check ---------------------
Generator Polynomial: 1000
Datawords to be sent:
['10100001']
Codewords sent by sender:
['10100001000']


Remainder: 000 NO ERROR DETECTED

Codewords received by receiver:
['10100101000']
Extracting datawords from codewords:
['10100101']

------------------------------------------------------------
```

```
C:\Users\SOURAV\Desktop\comp-networks-lab\ass1>python run.py case3-sender-input.txt
------------------------------------------------------------
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
-------------------------- CASE 3 ------------------------
Error is detected by VRC but not by CRC.
-------------- Vertical Redundancy Check -------------------
Datawords to be sent:
['110010']
Codewords sent by sender:
['1100101']


Parity is odd. ERROR DETECTED

Codewords received by receiver:
['1100111']
Extracting datawords from codewords:
['110011']

-------------- Cyclic Redundancy Check ---------------------
Generator Polynomial: 100
Datawords to be sent:
['110010']
Codewords sent by sender:
['11001000']


Remainder: 00 NO ERROR DETECTED

Codewords received by receiver:
['11001100']
Extracting datawords from codewords:
['110011']

------------------------------------------------------------
```

RESULTS

Vertical Redundancy Check (VRC):

This error detection scheme is quite popular in telecommunications and computer networking. This scheme works as follows: the message to be transmitted is split into nibbles (4 bits) and with each nibble a parity bit is associated before data transmission. Note that with even parity bit scheme, the number of 1's in the nibble including the parity bit must be even. 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.

Longitudinal Redundancy Check (LRC):

In contrast to VRC, LRC assigns a parity byte (nibble) along with the message to be transmitted. 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.

Checksum:

This scheme is a peculiar one wherein we perform 1's complement arithmetic on the data to be transmitted. For the message 1 0 1 1 1 0 0 0 1 0 0 1 , we compute checksum in two steps. The first step performs 1's complement addition on the data and the second step performs the complement on the result of Step 1. The result of Step 2 is the checksum which would be augmented along with the data to be transmitted.

Step 1: Perform 1's complement addition

1 0 1 1
1 0 0 0
1 0 0 1
1 1 0 0
1 (carry from the addition)
1 1 0 1

Step 2: Compute Checksum which is the 1's complement of the result of addition.

For the above example the checksum is 0 0 1 0. Along with the message, we append the checksum of the message, i.e., the message to be transmitted is

1 0 1 1 1 0 0 0 1 0 0 1 0 0 1 0

At the receiver, we perform 1's complement addition on the received data (message plus checksum), if the result is 1 1 1 1, then the receiver declares that there is no error in the transmission. Otherwise, it declares that there is an error in the transmission.

Cyclic Redundancy Check (CRC):

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 0. We compute CRC on the modified message M.

ANALYSIS

Vertical Redundancy Check (VRC):

1. This scheme detects all single bit errors. Further, it detects all multiple errors as long as the number of bits corrupted is odd (referred to as odd bit errors). Suppose the message to be transmitted is

   1 0 1 1 1 1 0 0 0 1 1 0 0 1 0

   and the message received at the receiver is

   1 0 0* 1 1 1 0 0 0 1 1 0 0 1 0
2. 0* represents that this bit is in error. For the above example, when the receiver performs the parity check, it detects that there was an error in the first nibble during transmission as there is a mismatch between the parity bit and the data in the nibble. However, the receiver does not know which bit in that nibble is in error. Similarly, the following error is also detected by the receiver.

   The message transmitted is

   1 0 1 1 1 1 0 0 0 1 1 0 0 1 0

   and the message received at the receiver is

   0*1*1 1 1 1 0 0 1*1 1 0 0 1 0

   Three bits are corrupted by the transmission medium and this is detected by the receiver as there is a mismatch between the 2nd nibble and the 2nd parity bit. It is important to note that the error in the first nibble is unnoticed as there is no mismatch between the data and the parity bit.
3. The above example also suggests that not all even bit erros (multiple bit error with the number of bits corrupted even) are detected by this scheme. If even bit error is such that the even number of bits are corrupted in each nibble, then such error is unnoticed at the receiver. However, if even bit is such that at least one of the nibble has odd number of bits in error, then such error is detected by VRC.

Longitudinal Redundancy Check (LRC):

1. Similar to VRC, LRC detects all single bit and odd bit errors. Some even bit errors are detected and the rest is unnoticed by the receiver.
2. The following error is detected by LRC but not by VRC. For the message 1011 1000 1001, supoose the received message is

   1 1 0 1
   1 0 0 0
   1 0 0 1
   1 0 1 0
3. In the above example, there is mismatch between the number of 1's and the parity bit in Columns 2 and 3.
4. The following error is detected by VRC but not by LRC. For the message 1011 1000 1001, suppose the received message is

   1 1 0 0*

1 0 0 1*
1 0 0 1
1 0 1 0

The above error is unnoticed by the receiver if we follow VRC scheme whereas it is detected by LRC as illustrated below.

1 0 1 0*1 1 0 0 1*1 1 0 0 1 0

5. Here again, not all even bit errors are detected by this scheme. If the error is such that each column has even number of bits in error, then such error is undetected. However, if the distribution is such that at least one column contains an odd number of bits in error, then such errors are always detected at the receiver.

Checksum:

1. If multiple bit error is such that in each column, a bit '0' is flipped to bit '1', then such an error is undetected by this scheme. Essentially, the message received at the receiver has lost the value 1 1 1 1 1 with respect to the sum. Although, it loses this value, this error is unnoticed at the receiver.
2. Also, multiple bit error is such that the difference between the sum of the sender's data and the sum of receiver's data is 1 1 1 1, then this error is unnoticed by the receiver.
3. The above two errors are undetected by the receiver due to the following interesting observations.
4. For any binary data a such that a != 0 0 0 0, the value of a + (1 1 1 1) in 1's complement arithmetic is a. On the similar line, if a1,...,an are nibbles, then a1+...+ak+(1 1 1 1) = a1+...+ak This is true because, a + 1 1 1 1 gives a – 1 with a carry '1' and inturn this '1' is added with a – 1 as part of 1's complement addition, yielding a.
5. Consider the scenario in which the sender transmits a1,...,ak along with the checksum and the transmission line corrupts multiple bits due to which we lose 1 1 1 1 on the sum. Although, the receiver received a1+...+ak-(1 1 1 1), it follows from the above observation that a1+...+ak-(1 1 1 1) is still a1+...+ak, thus error is undetected.
6. Similar to other error detection schemes, checksum detects all odd bit errors and most of even bit errors.

Cyclic Redundancy Check (CRC):

1. The answer to whether CRC detects all errors depends on the divisor polynomial used as a part of CRC computation. Consider the divisor polynomial x^2 which corresponds to 1 0 0 and the message to be transmitted is 1 0 1 0 1 0. After appending CRC bits (in this case 0 0) we get 1 0 1 0 1 0 0 0. Assuming during the data transmission, the 3rd bit is in error and the message received at the receiver is 1 0 1 0 1 1*0 0. On performing CRC check on 1 0 1 0 1 1 0 0, we see that the remainder is zero and the receiver wrongly concludes that there is no error in transmission. The reason this error is undetected by the receiver is that the message received is perfectly divisible by the divisor 1 0 0. More appropriately, if we visualize the received message as the xor of the original message and the error polynomial, then we see that the error polynomial is divisible by 1 0 0.
2. Message received = 1 0 1 0 1 1 0 0
   Message received = message transmitted + error polynomial, i.e.,
   1 0 1 0 1 1 0 0 = 1 0 1 0 1 0 0 0 + 0 0 0 0 0 1 0 0
   Clearly both are divisible by the divisor 1 0 0. In general, if the error polynomial is divisible by the divisor, then such errors are undetected by the receiver. The receiver wrongly concludes that there is no error in transmission. For example, if the error polynomial is 0 0 0 0 1 1 0 0 (3rd and 4th bits in the message in error), then the receiver is unaware of this error.

However, if the divisor is 1 0 1, then both errors are detected by the receiver. Thus, choosing an appropriate divisor is crucial in detecting errors at the receiver if any during the transmission.

COMMENTS

This assignment has helped me in understanding the different error detection schemes immensely, by researching and implementing them. It has also helped in understanding the demerits of a detection scheme, and how such demerits are overcome by other detection scheme.