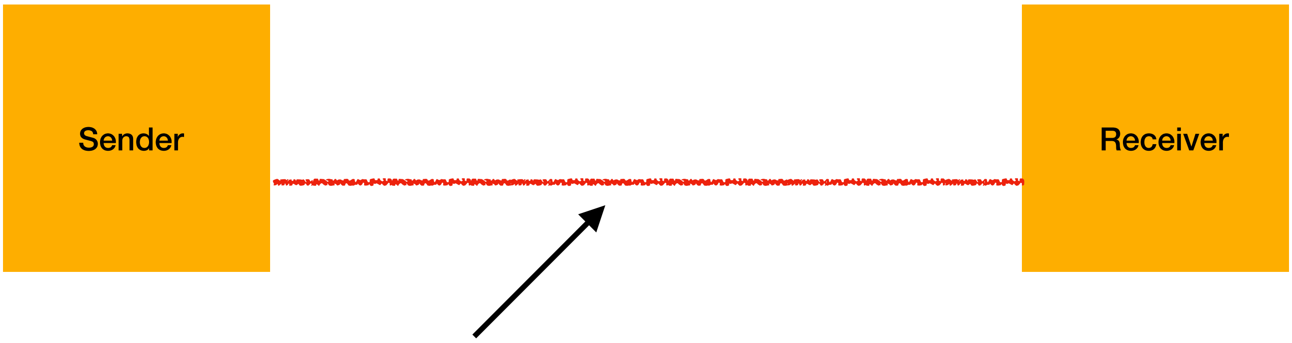
Name: Ritabroto Ganguly BCSE-III, 1st Sem

Roll: 001910501090

Design and implementation of error detection techniques within a simulated network environment



Channel:

As binary data is transmitted through a channel, noise in the channel may inject error into the stream of data by flipping one or more bits

---

Task:

We have to design and implement error detection algorithms

1. LRC
2. VRC
3. CRC
4. CHECKSUM

In a simulated environment for encoding and decoding binary data, so that incorrect binary data can be detected at the receiver side.

Implementation:

- Since we are working in a simulated environment using a high level programming language (python), direct bit manipulation is difficult, so we send strings of 0s and 1s representing the binary data.
- Binary data (as string) in encoded using different encoding techniques

- Error is injected in the encoded data before sending the data, by randomly flipping 0 string value to 1 or 1 to 0. Options for single bit, multi bit errors are given
- Receiver decodes the binary data (as string) and tries to detect error (if any) *and sends back a response to the sender*

Network.py (driver program):

```
import socket
import sys
import time
from math import sqrt
import random
import vrc
import lrc
import crc
import checksum
from threading import Thread,Condition,Lock


class Network:
    PORT = 12345
    def __init__(self,opt,timeout=1,recvBytes=1024,bits=256):
        self.recvB = recvBytes
        self.timeout = timeout
        self.bits = bits
        self.opt = opt
        #random.seed(Network.PORT)
        Network.PORT = random.randint(1025,12345)#using the same port for different runs can give error, so the port number is changed for each run


        #---fill up file data.txt with 'bits' number of random bits---#
        with open("data.txt","w") as file:
            l = ['0']*bits
            for i in range(0,bits):
                bit = random.randint(0,1)
                l[i] = str(bit)
            l = "".join(l)
            file.write(l)

    def injectError(self,word,burst=4):
        errors = random.randint(1,len(word)//burst)
        pos = []
        while errors>0:
            ind = random.randint(0,len(word)-1)
            if ind in pos:
                continue

            pos.append(ind)
            error = '0'
            if(word[ind]=='0'):
                error = '1'
            word = word[0:ind] + error + word[(ind+1):len(word)]
            if(burst<=1):
                return word
            errors -= 1
        return word

    def sender_vrc_or_crc(self,data,mysock,vrc_or_crc):
        frame = 8 #frame size for vrc and crc is 8, assuming total data bits is divisible
by 8
        num = 1
        j = 0
        for i in range(frame,len(data)+1,frame):
            dataword =  data[j:i]
            j = i
            print(f"dataword{num} =",dataword)
            codeword = vrc_or_crc.sender_check(dataword)
            print(f"sender codeword{num} =",codeword)
            codeword = self.injectError(codeword)
            mysock.sendall(codeword.encode())
            response = mysock.recv(self.recvB)
            print(f"response{num} =",response.decode())
            num += 1

    def recvr_vrc_or_crc(self,sender,vrc_or_crc):
        j = 1
        while True:
            codeword = sender.recv(self.recvB) #recv method waits until sender socket
closes.
            if(len(codeword)<=0):
                break

            codeword = codeword.decode()
            print(f'recver codeword{j} =',codeword)
            error = vrc_or_crc.recvr_check(codeword)
            sender.sendall(error.encode())
            j += 1

    def sender_lrc_or_checksum(self,data,mysock,lrc_or_checksum):
        frame = self.bits//4 #frame size is (total no. of bits)/4 for lrc, assuming that the
total number of bits is divisible by 4
        if(lrc_or_checksum is lrc):
            frame = int(sqrt(self.bits)) #frame size is square root of the total number of
bits for checksum, assuming that the total number of bits is a perfect square.
        j = 0
        words = []
        for i in range(frame,len(data)+1,frame):
            words.append(data[j:i])
            j = i

        print('grouped words:',words)
        codeword = lrc_or_checksum.sender_check(words,frame)
        print('sender codeword:',codeword)
        codeword = self.injectError(codeword) + 'x' #x appended to indicated end of
frame
        mysock.sendall(codeword.encode())
        response = mysock.recv(self.recvB)
        print(f"response =",response.decode())

    def recvr_lrc_or_checksum(self,sender,lrc_or_checksum):
        frame = self.bits//4
        if(lrc_or_checksum is lrc):
            frame = int(sqrt(self.bits))

        codeword = ""
        try:
            while True:
                recv = sender.recv(self.recvB)
                if(len(recv)<=0 or recv[-1]==120):#120 is x, the terminating character
                    codeword += recv[:len(recv)-1].decode()
                    break
                codeword += recv.decode()
        except Exception as e:
            if(isinstance(e,socket.timeout)==False):
                print('receiver exception:',e)

        print('recver codeword:',codeword)
        error = lrc_or_checksum.recvr_check(codeword,frame)
        #print(error)
        sender.sendall(error.encode())

    def sender(self):
        time.sleep(0.3) #so that the receiver thread starts before the sender thread
        mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        mysock.connect((socket.gethostname(),Network.PORT))

        file = open("data.txt","r")
        try:
            data = file.read() #reads entire data from the file, and then makes frames of
required size later
            file.close()
            if(self.opt=='1'):
                self.sender_vrc_or_crc(data,mysock,vrc)
            elif(self.opt=='2'):
                self.sender_vrc_or_crc(data,mysock,crc)
            elif(self.opt=='3'):
                self.sender_lrc_or_checksum(data,mysock,lrc)
            elif(self.opt=='4'):
                self.sender_lrc_or_checksum(data,mysock,checksum)

        finally:
            mysock.close()

    def receiver(self):
        with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as s:
            s.bind((socket.gethostname(),Network.PORT))
            s.listen()
            sender,addr = s.accept()

        try:
            if(self.opt=='1'):
                self.recvr_vrc_or_crc(sender,vrc)
            elif(self.opt=='2'):
                self.recvr_vrc_or_crc(sender,crc)
            elif(self.opt=='3'):
                self.recvr_lrc_or_checksum(sender,lrc)
            elif(self.opt=='4'):
                self.recvr_lrc_or_checksum(sender,checksum)

        except Exception as e:
            print('receiver exception:',e)
        finally:
            sender.close()

while(True):
    opt = input('''
1. VRC
2. CRC
3. LRC
4. CHECKSUM
0. Exit
''').strip()
    if(opt!='1' and opt!='2' and opt!='3' and opt!='4' and opt!='0'):
        print('Invalid Input')
        continue

    if(opt=='0'):
        sys.exit(0)

    net = Network(opt)
    st = Thread(None,net.receiver)
    ct = Thread(None,net.sender)

    st.start()
    ct.start()
    st.join()
    ct.join()
```

- The driver code is implemented using multithreading and TCP sockets.
- The class Network represents the entire connection
- The sender() method of Network class, drives the sender thread
- The receiver() method of Network class, drives the receiver thread
- Frame size is dependant upon the encoding being used. Frame sizes have been decided on the assumptions, that the total number of bits to be transferred is known before hand
- injectError() method is used to corrupt the bits
- lrc, vrc, crc and checksum are the 4 modules containing the algorithms for encoding and decoding data

lrc.py

```
def sender_check(words,frame):
    vert_bits = ['0']*frame
    for i in range(0,frame):
        vert_count = 0
        for word in words:
            if(word[i]=='1'):
                vert_count += 1

        if(vert_count%2!=0):
            vert_bits[i] = '1'

    hor_bits = ['0']*frame
    for i in range(0,len(words)):
        word = words[i]
        hor_count = 0
        for c in word:
            if(c=='1'):
                hor_count += 1

        if(hor_count%2!=0):
            hor_bits[i] = '1'

    words = "".join(words)
    vert_red = "".join(vert_bits)
    hor_red = "".join(hor_bits)
    #print('vert:',vert_red,'hor:',hor_red)
    return words+vert_red+hor_red #vertical redundant bits in the second last and horizontal redundant bits in the last


def recvr_check(word,frame):
    ERROR = "Error Detected"
    NOERR = "No Error"
    words = []
    j = 0
    for i in range(frame,len(word)+1,frame):
        words.append(word[j:i])
        j = i
    #print('recvr_ch:',words)
    for i in range(0,frame):
        vert_count = 0
        for j in range(0,len(words)-2):
            w = words[j]
            if(w[i]=='1'):
                vert_count += 1

        vert_count += int(words[len(words)-2][i])
        #print(f'after vert{i}',vert_count)
        if(vert_count%2!=0):
            return ERROR

    for i in range(0,len(words)-2):
        hor_count = 0
        for c in words[i]:
            if(c=='1'):
                hor_count += 1

        hor_count += int(words[len(words)-1][i])
        if(hor_count%2!=0):
            return ERROR

    return NOERR
```

- *Even parity is maintained/detected across rows and columns (formed by groups of frames) of bits by checking the number of 1s and encoding/decoding the necessary parity bits*

<u>vrc.py</u>

```
def sender_check(dw):
    count = 0
    for c in dw:
        if(c=='1'):
            count += 1
    dw += str(count%2)
    return dw

def recvr_check(cw):
    count = 0
    for c in cw:
        if(c=='1'):
            count += 1

    #print('recvr_check:',cw,count)
    if(count%2!=0):
        return "Error detected"

    return "No error”
```

- *Even parity is maintained/detected in frames by checking the number of 1s and encoding/decoding the necessary parity bit*

crc.py

```
generator = '100101'
ZERO = '00000'
def crc_div(word,recv):
    z = word
    if(recv==False):
        for i in range(0,len(generator)-1):
            z += '0'

    cur = z[0:len(generator)]
    for j in range(len(cur),len(z)+1):
        if(j>=len(z)):
            y = ''
        else:
            y = z[j]
        if(cur[0]=='0'):
            cur = cur[1:]+y
            continue

        x = ''
        for i in range(1,len(cur)):
            if(cur[i]==generator[i]):
                x += '0'
            else:
                x += '1'
        cur = x+y

    return cur

def sender_check(word):
    return word+crc_div(word,False)

def recvr_check(word):
    rem = crc_div(word,True)
    if(rem==ZERO):
        return "No Error"

    return 'Error Detected’
```

- *Parity bits are appended to the data word and error is detected on the codeword by performing manual mod-2 division on binary strings*
- *The generator word is fixed to 100101 (CRC-5-ITU)*

checksum.py

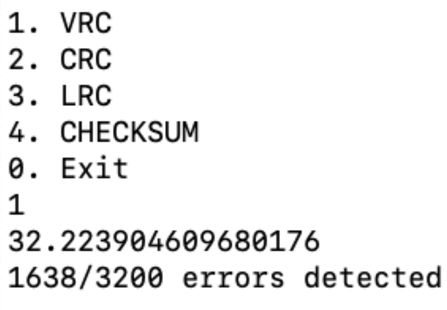
```
def add_frames(words,frame):
    s = ''
    c = '0'
    for i in range(frame-1,-1,-1):
        if(c=='1'):
            count1 = 1
        else:
            count1 = 0

        for word in words:
            if(word[i]=='1'):
                count1 += 1

        if(count1==0):
            s = '0' + s
            continue

        if(count1%2==0):
            s = '0' + s
            c = '1'
        else:
            s = '1' + s
            if(count1==1):
                c = '0'
            else:
                c = '1'
    if(c=='1'):
        return c+s

    return s

def get_sum(words,frame):
    t = ['0' for _ in range(frame)]
    s = ''.join(t)
    for i in range(0,len(words)):
        extra_bits = len(s)-len(words[i])
        if(extra_bits>0):
            for _ in range(extra_bits):
                words[i] = '0'+words[i]
        s = add_frames([s,words[i]],(frame+extra_bits))

    extra_bits = len(s) - frame
    if(extra_bits>0):
        f = frame - extra_bits
        t = ''
        for i in range(f):
            t += '0'

        t += s[0:extra_bits]
        s = add_frames([t,s[extra_bits:]],frame)

    t = ''
    for c in s:
        if(c=='0'):
            t += '1'
        else:
            t += '0'

    return t

def sender_check(words,frame):
    codeword = ''
    for w in words:
        codeword += w

    t = get_sum(words,frame)
    codeword += t
    return codeword

def recvr_check(word,frame):
    words = []
    j = 0
    for i in range(frame,len(word)+1,frame):
        words.append(word[j:i])
        j = i

    s = get_sum(words,frame)
    for c in s:
        if(c!='0'):
            return 'Error Detected'

    return 'No Error'
```

- *All data frames are added up and the complemented sum is appended to the concatenated data frames to form the codeword*
- *Receiver adds up the frames extracted from the codeword and detects error if the sum is not equal to 0*
- *Overflow of bits in the sum value is handled appropriately*

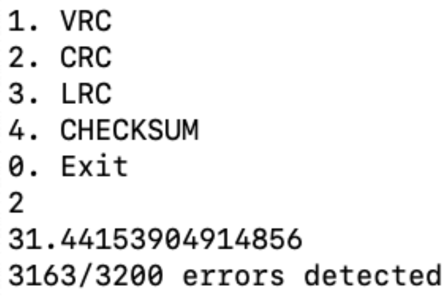
Analysis:

- We judge the correctness and accuracy of the each of the 4 frame coding algorithms by checking if they give “No Error” when no error is injected and by how many times it gives “Error Detected” when error is injected
- We also measure the performance of the each frame coding implementation on the basis of the time it consumed to complete the entire algorithm (encoding, sending, decoding, receiving response)

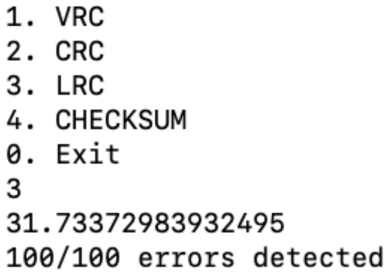
- *We make the total number number bits 256 and we repeat the process a 100 times for each encoding method*



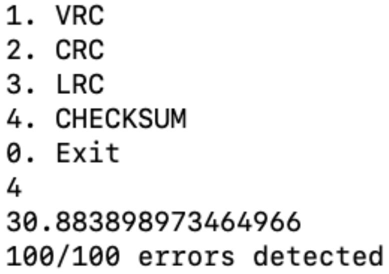
```
1. VRC
2. CRC
3. LRC
4. CHECKSUM
0. Exit
1
32.223904609680176
1638/3200 errors detected
```



```
1. VRC
2. CRC
3. LRC
4. CHECKSUM
0. Exit
2
31.44153904914856
3163/3200 errors detected
```



```
1. VRC
2. CRC
3. LRC
4. CHECKSUM
0. Exit
3
31.73372983932495
100/100 errors detected
```



```
1. VRC
2. CRC
3. LRC
4. CHECKSUM
0. Exit
4
30.883898973464966
100/100 errors detected
```

- VRC and CRC break the total bits in frames of 8 and hence total 3200 frames are checked
- For LRC and CHECKSUM all the total bits are sent at once, so total 100 frames are checked

Result:

As observed from the outputs above,

- My CHECKSUM implementation is the most efficient
- My CRC and CHECKSUM implementations are much more accurate than my LRC and VRC
- My VRC is the least accurate and also the least efficient

Comments:

There are 2 drawbacks of using multithreading with sockets in this simulated environment

- The port number needs to be changed for each thread.start(), because the same port may not be free for use on consecutive runs and hence might throw *ConnectionRefusedError*
- We have to delay the run of the sender thread by a few milliseconds on each thread.start(), so that the receiver thread can setup the socket and start listening before the sender thread sends a connection request to that socket, otherwise error may be thrown