NAME: Debadrita Roy

CLASS: BCSE-III

GROUP: A1

ASSIGNMENT NUMBER: 1

PROBLEM STATEMENT:

Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. 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. (b) Error is detected by checksum but not by CRC. (c) Error is detected by VRC but not by CRC.

DEADLINE: 10th August, 2021

DATE OF SUBMISSION: 10th August, 2021

DESIGN

The purpose of the program is to simulate the real-world network environment and design and implement error detection techniques to detect whether the data packets have been corrupted on the way from the sender to the receiver. The required programs are written in Python 3.

An error detection module (errordetect.py) is designed which contains the four error detection schemes- VRC, LRC, Checksum and CRC. There are functions in this module which add redundant bits to the dataword according to the respective schemes and return the codeword. Also, there are functions which take a codeword as a parameter and detect whether there is an error or not (the functions return TRUE if error is detected and FALSE if no error is detected). This module is available to both the sender and the receiver programs.

errordetect.py

def vrc(data, parity=’odd’): This makes a codeword from the data using VRC scheme and returns the codeword. The default parity is odd, to use even parity we need to pass ‘even’ to the function.

def detect_error_vrc(codeword, parity=’odd’): This detects error in the codeword made using VRC scheme and returns True if error is detected, otherwise returns False. The default parity is odd, to use even parity we need to pass ‘even’ to the function.

def lrc(data, parity=’odd’): This makes a codeword from the data using LRC scheme and returns the codeword. The default parity is odd, to use even parity we need to pass ‘even’ to the function.

def detect_error_lrc(codeword, parity=’odd’): This detects error in the codeword made using LRC scheme and returns True if error is detected, otherwise returns False. The default parity is odd, to use even parity we need to pass ‘even’ to the function.

def checksum(data): This makes a codeword from the data using Checksum scheme and returns the codeword. The data is divided into 4-bit words.

def detect_error_checksum(codeword): This detects error in the codeword made using Checksum scheme and returns True if error is detected, otherwise returns False. The codeword is divided into 4-bit words.

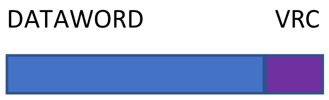
def crc(data, poly='111010101'): This makes a codeword from the data using CRC scheme and returns the codeword. The default polynomial used is CRC-8 ($x^8+x^7+x^6+x^4+x^2+1$). We can send other CRC polynomials to the function if needed.

def detect_error_crc(data, poly='111010101'): This detects error in the codeword made using CRC scheme and returns True if error is detected, otherwise returns False. The default polynomial used is CRC-8 ($x^8+x^7+x^6+x^4+x^2+1$). We can send other CRC polynomials to the function if needed.

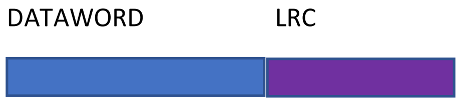
The sender program (**sender.py**) takes the name of a test file containing a sequence of 0's and 1's as input. It then divides up the data into datawords of customizable size, makes a codeword using an appropriate scheme from errordetect.py and injects error into the codeword using a function from the channel file (**channel.py**). Then socket is used to send the codeword with or without error to the receiver. The receiver program (**receiver.py**) receives the codeword, then calls the appropriate error detection function from errordetect.py and if there is an error detected, then the codeword is discarded and the fact that an error has been detected is sent to the sender. If no error is detected, the dataword is stored and the fact that no error was detected is sent to the sender. When transmission is complete, the connection is closed.

Packet Structure:

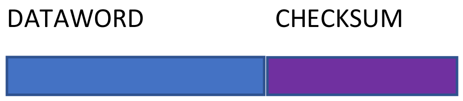
The size of the codewords is different, depending on which scheme is used. The size of the dataword is same in each case (predetermined). As the number of redundant bits added for each scheme is different, the size of the codeword is also different.



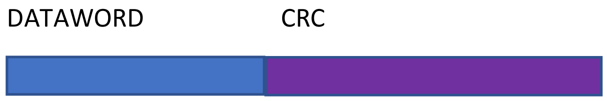
1 redundant bit is added to the dataword such that the parity is odd.



4 redundant bits are added to the dataword. The dataword is broken up into groups of 4-bit words and a parity bit is calculated for each column.

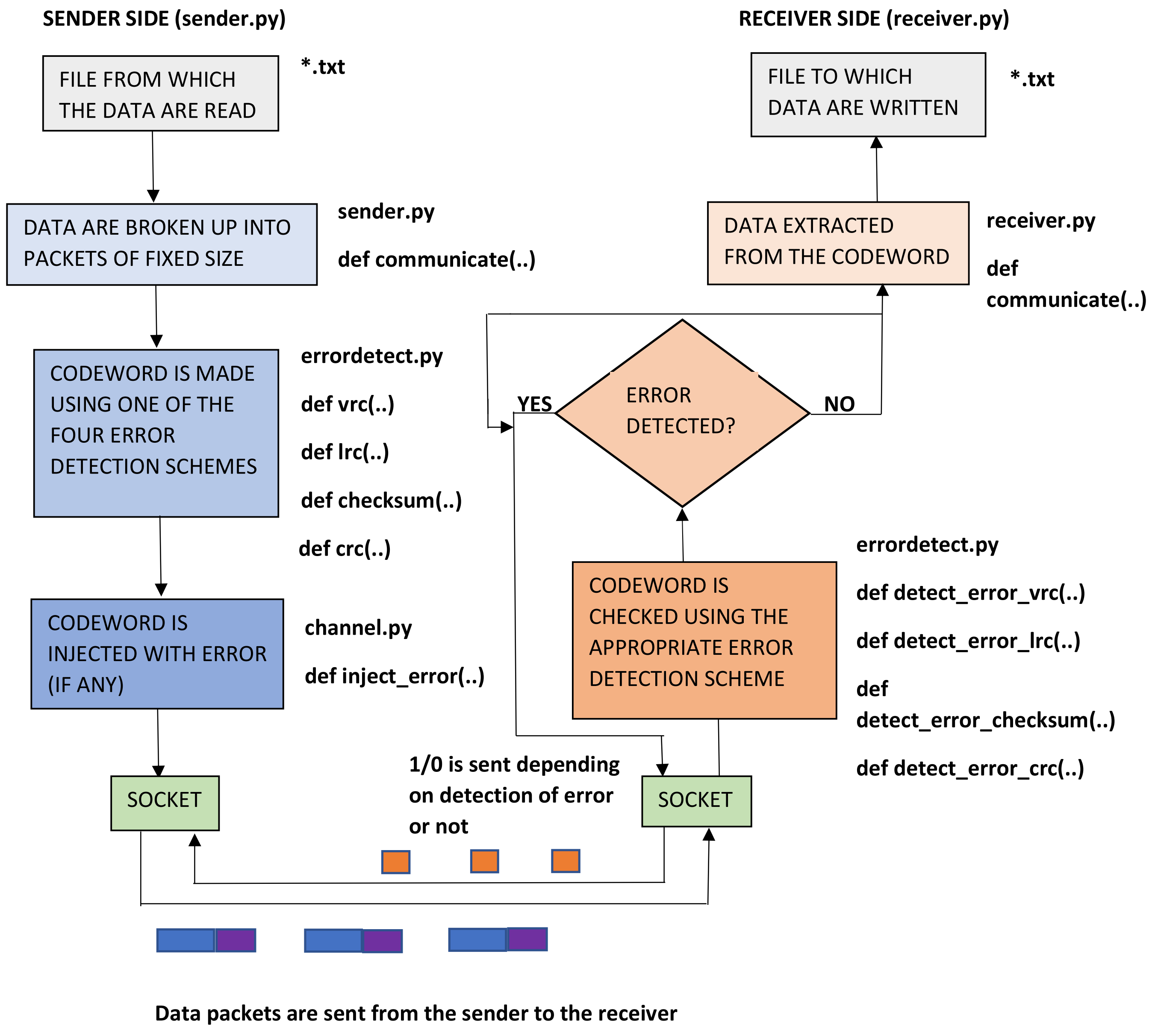


4 redundant bits are added to the dataword. The dataword is broken up into groups of 4-bit words and the sum of all the words is calculated. The checksum is found by complementing the sum using 1's complement. The checksum is then added to the dataword.



The number of redundant bits added to the dataword depends upon the degree of the CRC polynomial. For CRC-8 polynomial, 8 redundant bits are added. For CRC-16 polynomial, 16 redundant bits are added.

Structure Diagram:


SENDER SIDE (sender.py)
RECEIVER SIDE (receiver.py)
FILE FROM WHICH THE DATA ARE READ
*.txt
DATA ARE BROKEN UP INTO PACKETS OF FIXED SIZE
sender.py
def communicate(..)
CODEWORD IS MADE USING ONE OF THE FOUR ERROR DETECTION SCHEMES
errordetect.py
def vrc(..)
def lrc(..)
def checksum(..)
def crc(..)
CODEWORD IS INJECTED WITH ERROR (IF ANY)
channel.py
def inject_error(..)
SOCKET
1/0 is sent depending on detection of error or not
SOCKET
FILE TO WHICH DATA ARE WRITTEN
*.txt
DATA EXTRACTED FROM THE CODEWORD
receiver.py
def communicate(..)
YES
ERROR DETECTED?
NO
CODEWORD IS CHECKED USING THE APPROPRIATE ERROR DETECTION SCHEME
errordetect.py
def detect_error_vrc(..)
def detect_error_lrc(..)
def detect_error_checksum(..)
def detect_error_crc(..)

Data packets are sent from the sender to the receiver

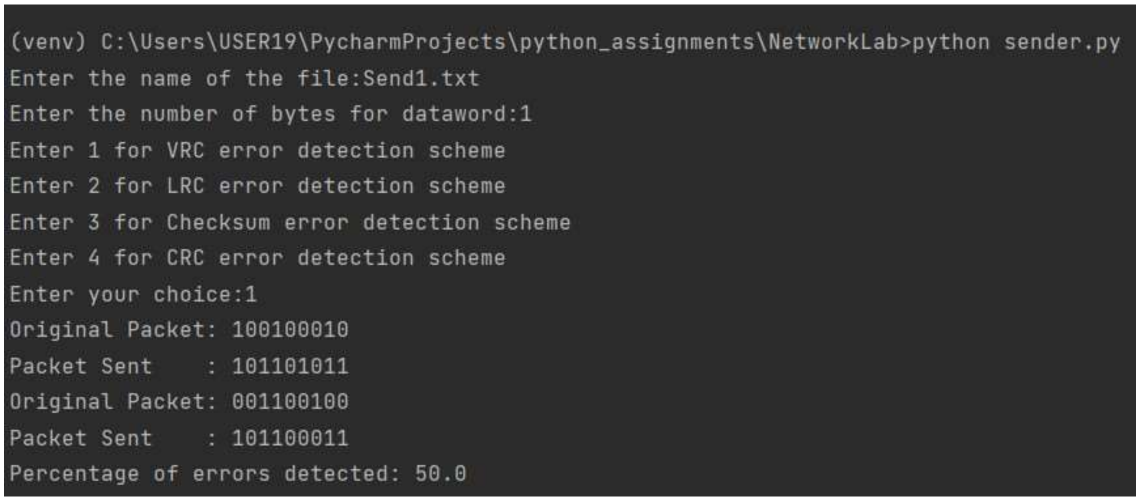
Input Format:

The name of the test file in which the sequence of 0's and 1's is stored is given as input to the sender program. The size of the dataword (in bytes) is also to be determined and the error detection scheme to be used for the run is to be chosen from the list.

Output Format:

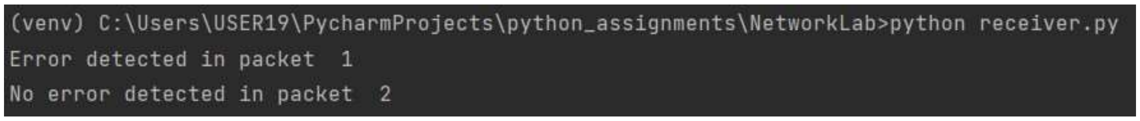
The original packet and the packet having error(s) are displayed on the sender terminal. On the receiver terminal, whether error has been detected or not is displayed. Lastly, the percentage of detected errors to the total number of errors is displayed on the sender terminal.

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:1
Original Packet: 100100010
Packet Sent    : 101101011
Original Packet: 001100100
Packet Sent    : 101100011
Percentage of errors detected: 50.0
```

RECEIVER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
Error detected in packet  1
No error detected in packet  2
```

IMPLEMENTATION

sender.py

The Sender class contains a constructor to initialize the ip and port. It has a communicate(..) function which takes as argument the filename from where the data are to be read, the size of the dataword and the choice of the error detection scheme.

def __init__(self,ip,port)

Method Description: The constructor initializes the ip and the port.

Code Snippet:

```
def __init__(self,ip,port):
   self.ip=ip
   self.port=port
```

def communicate(self,fn,fsize,ch)

Method Description: This function first sends the agreed upon dataword size and error detection scheme to the receiver. It then breaks up the data into the appropriately-sized datawords (if the length of the last dataword is less than the predetermined size, a sequence of 0’s is appended to the right of the data to get to the appropriate size), and calls the chosen error detection scheme function from the error detection module to make the codeword. It then randomly injects error into the codeword using the inject_error(..) function from channel.py and sends it to the receiver using socket. It then receives a packet from the receiver indicating whether an error was detected or not. Using this, it calculates the percentage of errors detected to the total number of errors injected.

Code Snippet:

```
def communicate(self,fn,fsize,ch):
  f=open(fn,'r')
  s=f.read()
  f.close()
  if len(s)%fsize !=0:
    st='0'*(fsize-(len(s)%fsize))
    s=s+st
  i=0
  count=0
  tcount=0
  m=bin(ch)[2:]
  conn.send(m.encode('utf-8'))
  m=bin(fsize)[2:]
  conn.send(m.encode('utf-8'))
  while i<len(s):
    if ch==1:
      m=errordetect.vrc(s[i:(i + fsize)])
    elif ch==2:
      m=errordetect.lrc(s[i:(i + fsize)])
    elif ch==3:
      m=errordetect.checksum(s[i:(i + fsize)])
    elif ch==4:
      m= errordetect.crc(s[i:(i + fsize)])
    print('Original Packet:',m)
    m2=channel.inject_error(m)
    print('Packet Sent    :',m2)
    conn.send(m2.encode('utf-8'))
    d=conn.recv(BUFFER_SIZE)
    er=d.decode('utf-8')
    if m2!=m and er=='1':
      count+=1
    if m!=m2:
      tcount+=1
    i=i+fsize
  m='11111111'
  conn.send(m.encode('utf-8'))
```

```
        print('Percentage of errors detected:',(count*100)/tcount)
```

main()

Method Description: The main() function sets up the connection between the sender and the receiver using socket programming and also accepts the user's choice of dataword size and error detection scheme.

Code Snippet:

```
if __name__=="__main__":
    TCP_IP = '0.0.0.0'
    TCP_PORT = 2004
    BUFFER_SIZE = 1024
    file=input('Enter the name of the file:')
    fsize=8*int(input('Enter the number of bytes for dataword:'))
    while True:
        print('Enter 1 for VRC error detection scheme')
        print('Enter 2 for LRC error detection scheme')
        print('Enter 3 for Checksum error detection scheme')
        print('Enter 4 for CRC error detection scheme')
        ch=int(input('Enter your choice:'))
        if 1 <= ch <= 4:
            break
        print('Invalid Input!!!Enter a valid number')
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((TCP_IP, TCP_PORT))
    s.listen(5)
    (conn, (ip, port)) = s.accept()
    obj=Sender(ip,port)
    obj.communicate(file,fsize,ch)
```

channel.py

The channel file contains functions to inject single-bit error, burst error (of given length) and random error (the bits are flipped at random) to the codeword. The inject_error function calls the appropriate function when a codeword is passed as an argument to it.

def inject_random_error(data)

Method Description: This function injects error to the data by flipping bits at random and returns the resultant dataword.

Code Snippet:

```
def inject_random_error(data):
    errdata = ''
    for i in range(len(data)):
        x = int(random() * 2)
        if x == 1:  # flip the bit
            if data[i] == '0':
```

```
            errdata += '1'
         else:
            errdata += '0'
      else:
         errdata += data[i]
   return errdata
```

def inject_single_bit_error(data)

Method Description: This function injects error to the data by choosing a bit at random and flipping it. It concatenates the bits before the chosen bit, the flipped bit and the rest of the bits after it. Error is injected if the chosen bit is present in the data. It then returns the resultant dataword.

Code Snippet:

```
def inject_single_bit_error(data):
   x = int(random()*(2*len(data)))
   if x<len(data):
      if data[x]=='0':
         data=data[0:x]+'1'+data[(x+1):]
      else:
         data=data[0:x]+'0'+data[(x+1):]
   return data
```

def inject_burst_error(data,l)

Method Description: This function injects a burst error of length l to the data. First, the bit where the burst error starts is chosen at random. Also, whether an error is to be injected or not is also decided. If the error is to be injected, we first concatenate the bits before the starting bit with the flipped starting bit. Then we add the intermediate bits between the starting and the ending bits. We flip these bits at random. Then we add the flipped ending bit and the rest of the original bits after it.

Code Snippet:

```
def inject_burst_error(data, l):
   x = int(random()*2*(len(data) - l))
   if x<(len(data)-l):
      if data[x]=='0':
         errdata=data[0:x]+'1'
      else:
         errdata=data[0:x]+'0'
      i=1
      while i<(l-1):
         y = int(random() * 2)
         if y == 1:  # flip the bit
            if data[x+i] == '0':
               errdata += '1'
            else:
               errdata += '0'
         else:
            errdata +=data[x+i]
```

```
        i+=1
      if data[x+l-1]=='0':
        errdata+='1'
      else:
        errdata+='0'
      errdata+=data[(x+l):]
    else:
      errdata=data
    return errdata
```

def inject_error(data)

Method Description: This function injects error to the data by calling one of the above functions and returns the resultant dataword.

Code Snippet:

```
  def inject_error(data):
    x=int(random()*3)
    # print('Press 1 for single-bit error, 2 for burst error, 3 for random errors')
    # x=int(input('Enter your choice:'))
    # x=2
    if x==0:
      data=inject_single_bit_error(data)
    elif x==1:
      data=inject_burst_error(data,int(random()*len(data)/2)+2)
    elif x==2:
      data=inject_random_error(data)
    return data
```

receiver.py

The Receiver class contains a communicate(..) function which accepts as argument the name of the file where the received data is to be written.

def communicate(self,fn)

Method Description: This function first receives the dataword size and the error detection scheme. It then receives the codewords until the transmission is complete. It calls the appropriate function from the error detection module to check the codeword for error(s). If error is detected, the codeword is discarded and 1 is sent back to the sender. If no error is detected, the dataword is extracted and written into the file. 0 is sent back to the sender.

Code Snippet:

```
def communicate(self,fn):
  f=open(fn,'w')
  m=''
  data = (r.recv(BUFFER_SIZE)).decode('utf-8')
  ch = int(data,2)
  data=(r.recv(BUFFER_SIZE)).decode('utf-8')
  fsize=int(data,2)
```

```
        x=1
        while True:
            data=(r.recv(BUFFER_SIZE)).decode('utf-8')
            if data=='11111111':
                break
            if ch==1:
                b=errordetect.detect_error_vrc(data)
            elif ch==2:
                b=errordetect.detect_error_lrc(data)
            elif ch==3:
                b=errordetect.detect_error_checksum(data)
            else:
                b=errordetect.detect_error_crc(data)
            if b:
                print('Error detected in packet ',x)
                m='1'
            else:
                print('No error detected in packet ',x)
                m='0'
                f.write(data[0:fsize])
            r.send(m.encode('utf-8'))
            x+=1
        f.close()
```

main()

Method Description: It contains the code for establishing the socket connection and closing the connection once the transmission is complete.

Code Snippet:

```
if __name__ == '__main__':
    host = socket.gethostname()
    port = 2004
    BUFFER_SIZE = 1024
    r = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    r.connect((host, port))
    obj=Receiver()
    obj.communicate('Received.txt')
    r.close()    # close connection
```

errordetect.py

The error detection module contains the implementation of the four error detection schemes and is imported by both the sender and receiver programs.

def vrc(data, parity='odd')

Method Description: This function accepts the dataword and the parity (if no parity is given, odd parity is assumed) and returns the codeword. It counts the number of 1's in the dataword and adds the parity bit depending on the chosen parity (for odd parity, if the number of 1’s in the dataword is odd, then parity bit is 0, otherwise it is 1. Opposite for even parity).

Code Snippet:

```
def vrc(data,parity='odd'):
  n=0
  for ch in data:
    if ch == '1':
      n=n+1
  if n%2 ==0 and parity=='odd':    # even number of 1's
    codeword=data+'1'
  elif parity=='odd':
    codeword=data+'0'
  elif n%2==0:
    codeword=data+'0'
  else:
    codeword=data+'1'
  return codeword
```

def detect_error_vrc(codeword, parity=’odd’)

Method Description: This function accepts the codeword and the parity (if no parity is given, odd parity is assumed) and returns True if it detects error, otherwise it returns False. It counts the number of 1’s in the dataword. For odd parity, if the number of 1’s in the codeword is odd, it returns True, otherwise returns False. Opposite for even parity.

Code Snippet:

```
def detect_error_vrc(codeword,parity='odd'):
  n = 0
  for ch in codeword:
    if ch == '1':
      n = n + 1
  if n%2==0 and parity=='odd':
    return True
  elif parity=='odd':
    return False
  elif n%2==0:
    return False
  else:
    return True
```

def lrc(data, parity=’odd’)

Method Description: This function accepts the dataword and parity (if no parity is given, it assumes odd parity) and returns the codeword. The dataword is divided into segments of 4 bits each and parity bit is calculated for each column. The resultant 4 redundant bits are added to the dataword to form the codeword. The number of 1’s for each column is calculated by keeping a list having 4 elements, one for each column. Every 4th position results in increment to the same element if the bit is 1, no increment is done if the bit is 0.

Code Snippet:

```
def lrc(data,parity='odd'):
  codeword=data
  n=[0,0,0,0]
  i=0
  for ch in data:
    if ch=='1':
      n[i%4]=n[i%4]+1
    i=i+1
  if parity=='odd':
    for x in n:
      if x%2==0:
        codeword=codeword+'1'
      else:
        codeword=codeword+'0'
  else:
    for x in n:
      if x%2==0:
        codeword=codeword+'0'
      else:
        codeword=codeword+'1'
  return codeword
```

def detect_error_lrc(codeword, parity='odd')

Method Description: This function accepts the codeword and parity (if no parity is given, it assumes odd parity) and returns True if an error is detected, otherwise returns False. The codeword is divided into segments of 4 bits each. The number of 1's for each column is calculated by keeping a list having 4 elements, one for each column. Every 4th position results in increment to the same element if the bit is 1, no increment is done if the bit is 0. If any list element is even (for odd parity), returns False, otherwise returns True. Reverse for even parity.

Code Snippet:

```
def detect_error_lrc(codeword,parity='odd'):
  n=[0,0,0,0]
  i=0
  for ch in codeword:
    if ch=='1':
      n[i%4]=n[i%4]+1
    i=i+1
  if parity=='odd':
    for x in n:
      if x%2==0:
        return True
    return False
  else:
    for x in n:
      if x%2==1:
        return True
    return False
```

def checksum(data)

Method Description: This function accepts a dataword and returns the codeword using checksum scheme. The dataword is broken up into segments having 4 bits each. The segments are then added using binary addition. The checksum is produced by performing 1's complement on the binary sum of the segments (flipping all the bits from 0 to 1 and vice versa). The checksum is concatenated with the dataword to form the codeword.

Code Snippet:

```
def checksum(data):
  codeword=data
  s=0
  i=0
  while i<len(data):
    s=s+int(data[i:i+4],2)
    i=i+4
  sum=bin(s)[2:]
  while len(sum)!=4:
    if len(sum)<4:
      sum='0'*(4-len(sum))+sum
    elif len(sum)>4:
      sum=bin(int(sum[(len(sum)-4):],2)+int(sum[0:(len(sum)-4)],2))[2:]
  for x in sum:
    if x=='0':
      codeword+='1'
    else:
      codeword+='0'
  return codeword
```

def detect_error_checksum(codeword)

Method Description: This function accepts a codeword and returns True if it detects an error, otherwise returns False. The codeword is broken up into segments having 4 bits each. The segments are then added using binary addition. The checksum is produced by performing 1's complement on the binary sum of the segments (flipping all the bits from 0 to 1 and vice versa). If the checksum is 0, there is no error. So, if there is at least one 0 in the sum of the segments, return True, otherwise return False.

Code Snippet:

```
def detect_error_checksum(codeword):
  s=0
  i=0
  while i<len(codeword):
    s=s+int(codeword[i:i+4],2)
    i=i+4
  sum=bin(s)[2:]
```

```
while len(sum)!=4:
   if len(sum)<4:
      sum='0'*(4-len(sum))+sum
   elif len(sum)>4:
      sum=bin(int(sum[(len(sum)-4):],2)+int(sum[0:(len(sum)-4)],2))[2:]
for x in sum:
   if x=='0':   # on complementing, it will become a 1, so result is non-zero
      return True    # error detected
return False
```

def crc(data, poly='111010101')

Method Description: This function accepts a dataword and a CRC polynomial (if no polynomial is given, it assumes CRC-8 to be the polynomial) and returns the codeword. The dataword is concatenated with a number of 0's (same as the degree of the CRC polynomial). Binary division is then performed on it. We first find the remainder by XORing the polynomial with the first respective bits of the dataword. The result becomes the new dataword along with the bits afterward. The process continues till the end of the dataword. The remainder is concatenated to the dataword to produce the codeword.

Code Snippet:

```
def crc(data,poly='111010101'):
   codeword=data
   data+='0'*(len(poly)-1)
   a=data[0:len(poly)]
   rem=''
   i=len(poly)
   while True:
      rem=bin(int(a,2)^int(poly,2))[2:]
      if (i+(len(poly)-len(rem)))>len(data):
         a=rem+data[i:]
         break
      a=rem+data[i:i+(len(poly)-len(rem))]
      i=i+(len(poly)-len(rem))
   if len(a)<(len(poly)-1):
      a='0'*(len(poly)-len(a)-1)+a
   codeword+=a
   return codeword
```

def detect_error_crc(codeword, poly='111010101')

Method Description: This function accepts a codeword and a CRC polynomial (if no polynomial is given, it assumes CRC-8 to be the polynomial) and returns True if it detects error, otherwise returns False. Binary division is performed on the codeword. We first find the remainder by XORing the polynomial with the first respective bits of the codeword. The result becomes the new dividend along with the bits afterward. The process continues till the end of the codeword. If the remainder at the end is 0, no error so return False, otherwise return True.

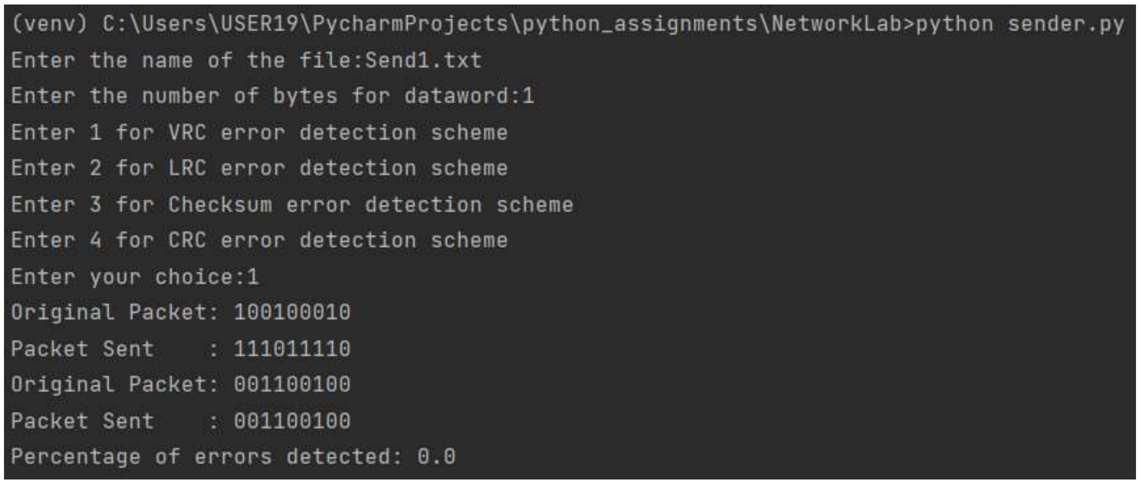
Code Snippet:

```
def detect_error_crc(codeword,poly='111010101'):
    a=codeword[0:len(poly)]
    i=len(poly)
    while True:
        rem = bin(int(a, 2) ^ int(poly, 2))[2:]
        if (i + (len(poly) - len(rem))) > len(codeword):
            a = rem + codeword[i:]
            break
        a = rem + codeword[i:i + (len(poly) - len(rem))]
        i = i + (len(poly) - len(rem))
    if int(a,2)==0:
        return False
    else:
        return True
```

TEST CASES

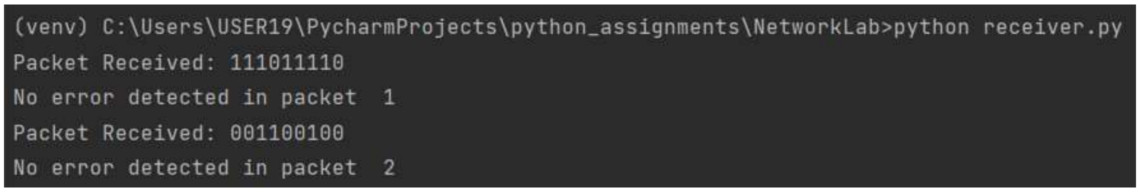
Sample Test 1: To check the working of the socket connection *(received packets are displayed by the receiver to ensure this)*

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:1
Original Packet: 100100010
Packet Sent    : 111011110
Original Packet: 001100100
Packet Sent    : 001100100
Percentage of errors detected: 0.0
```

RECEIVER TERMINAL

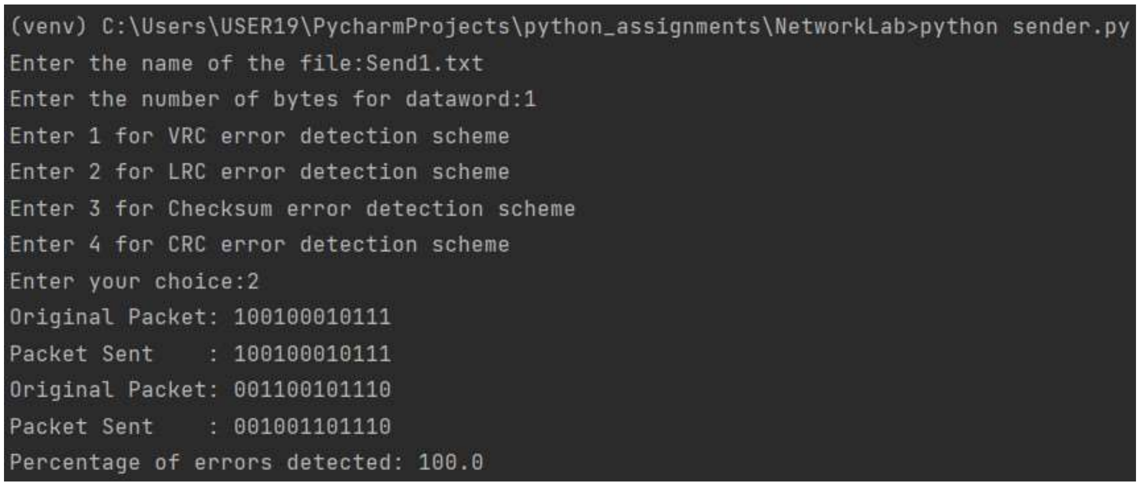


```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
Packet Received: 111011110
No error detected in packet  1
Packet Received: 001100100
No error detected in packet  2
```

We find that the packets are sent successfully over the socket. So, the portion of the program responsible for setting up the socket connection, using it and closing it once the transmission is complete, works correctly.

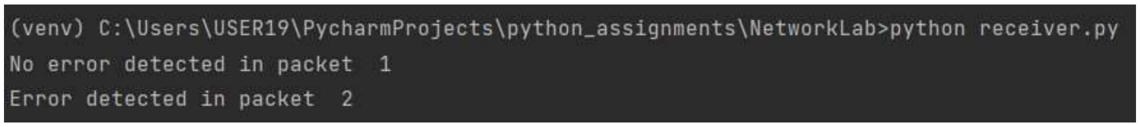
Sample Test 2: To check the working of LRC when two packets, one with error and the other without, are sent to the receiver

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:2
Original Packet: 100100010111
Packet Sent    : 100100010111
Original Packet: 001100101110
Packet Sent    : 001001101110
Percentage of errors detected: 100.0
```

RECEIVER TERMINAL

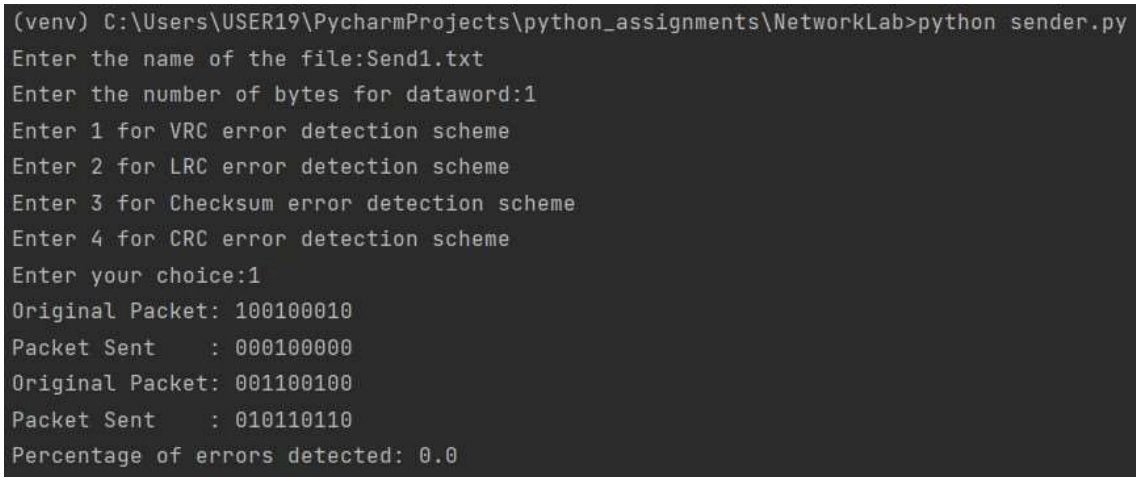


```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
No error detected in packet  1
Error detected in packet  2
```

In the above case, the receiver detected the error in packet 2 and also detected that there was no error in packet 1.

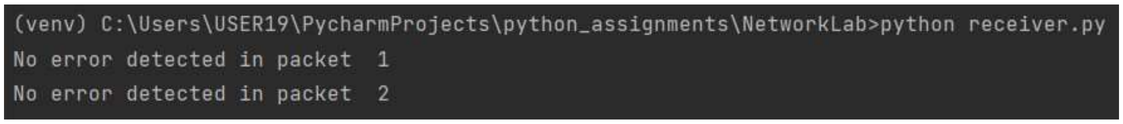
Sample Test 3: To check the working of VRC when two packets with random error are sent to the receiver

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:1
Original Packet: 100100010
Packet Sent    : 000100000
Original Packet: 001100100
Packet Sent    : 010110110
Percentage of errors detected: 0.0
```

RECEIVER TERMINAL

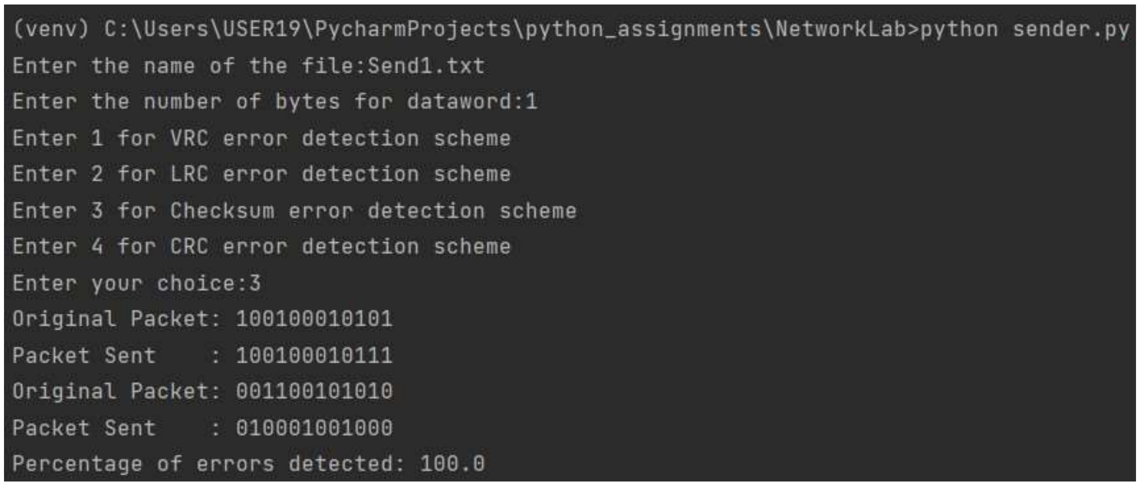


```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
No error detected in packet  1
No error detected in packet  2
```

In the above case, the errors have not been detected by the VRC.

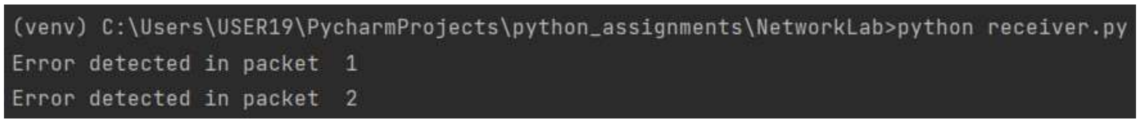
Sample Test 4: To check the working of Checksum when two packets with random error are sent to the receiver

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:3
Original Packet: 100100010101
Packet Sent    : 100100010111
Original Packet: 001100101010
Packet Sent    : 010001001000
Percentage of errors detected: 100.0
```

RECEIVER TERMINAL

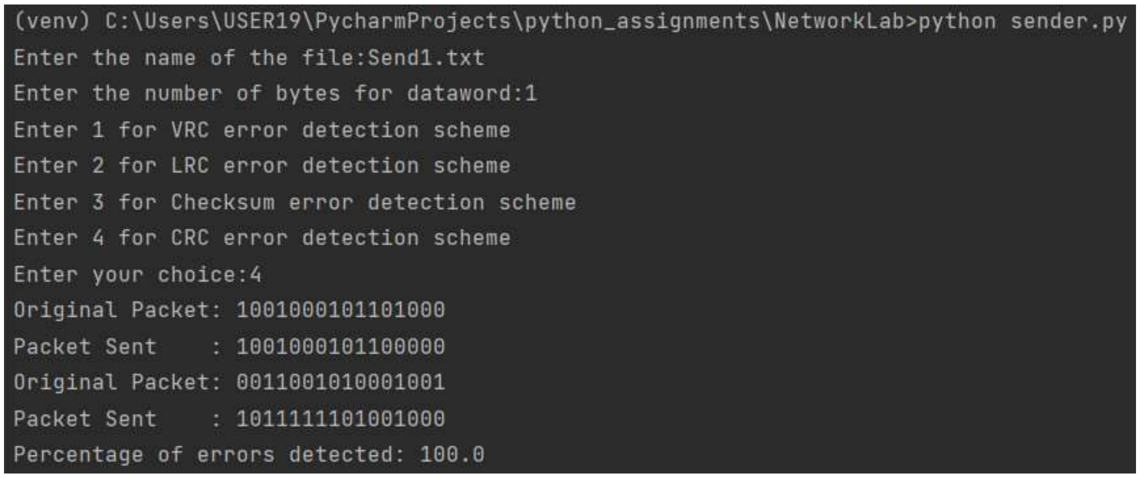


```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
Error detected in packet  1
Error detected in packet  2
```

In the above case, both the errors have been detected by Checksum scheme.

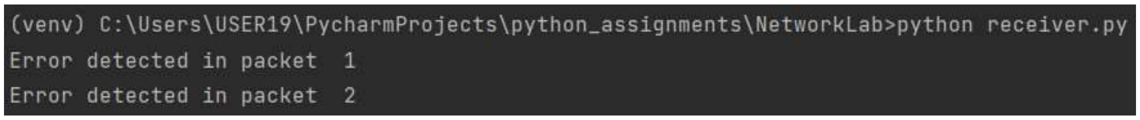
Sample Test 5: To check the working of CRC (CRC-8) when two packets with random error are sent to the receiver

SENDER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python sender.py
Enter the name of the file:Send1.txt
Enter the number of bytes for dataword:1
Enter 1 for VRC error detection scheme
Enter 2 for LRC error detection scheme
Enter 3 for Checksum error detection scheme
Enter 4 for CRC error detection scheme
Enter your choice:4
Original Packet: 1001000101101000
Packet Sent    : 1001000101100000
Original Packet: 0011001010001001
Packet Sent    : 1011111101001000
Percentage of errors detected: 100.0
```

RECEIVER TERMINAL



```
(venv) C:\Users\USER19\PycharmProjects\python_assignments\NetworkLab>python receiver.py
Error detected in packet  1
Error detected in packet  2
```

In the above case, both the errors have been detected by CRC.

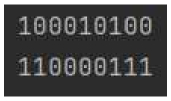
The test cases 1-5 also help in checking whether errors have been injected randomly and the test cases 2-5 also check that the functions of the error detection module are available for use of both the sender and the receiver programs.

The following test cases check whether the functions in the error detection module perform as expected.

Sample Test 6: To check the correctness of the VRC function producing the codeword

Test Data: 10001010 and 11000011

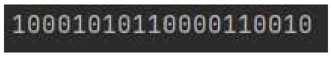
Output:



```
100010100
110000111
```

Sample Test 7: To check the correctness of the LRC function producing the codeword

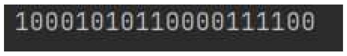
Test Data: 1000101011000011

Output: 

```
10001010110000110010
```

Sample Test 8: To check the correctness of the Checksum function producing the codeword

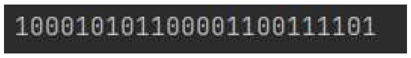
Test Data: 1000101011000011

Output: 

```
10001010110000111100
```

Sample Test 9: To check the correctness of the CRC function (CRC-8) producing the codeword

Test Data: 1000101011000011

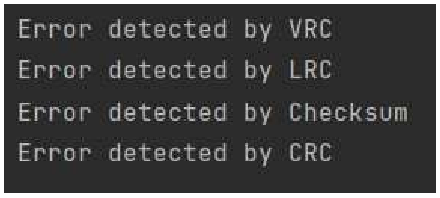
Output: 

```
100010101100001100111101
```

Sample Test 10: To check whether a single-bit error can be detected by the four schemes

Test Data: 1000111011000011 for VRC, 10001110110000110010 for LRC, 10001110110000111100 for Checksum, 100011101100001100111101 for CRC (CRC-8)

Output:



```
Error detected by VRC
Error detected by LRC
Error detected by Checksum
Error detected by CRC
```

Thus, a single-bit error **can be detected by all the four schemes**.

Sample Test 11: To check whether Checksum and CRC (CRC-8) can detect the following error

Test Data:

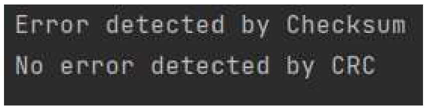
Original Dataword:0000000111010101

Redundant bits added due to Checksum: 1011

Redundant bits added due to CRC: 00000000

Dataword having error: 0000001110101010

Output:



```
Error detected by Checksum
No error detected by CRC
```

The above burst **error was detected by checksum and not detected by CRC** (CRC-8).

Sample Test 12: To check whether VRC and CRC (CRC-7) can detect the following error

Test Data:

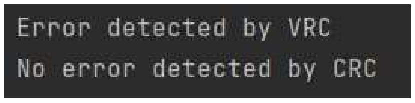
Original Dataword: 0000000000000000

Redundant bit added due to VRC: 1

Redundant bits added due to CRC: 0000000

Dataword having error: 0000000010001001

Output:



```
Error detected by VRC
No error detected by CRC
```

In the above case, the burst **error was detected by VRC but not detected by CRC** (CRC-7).

Thus, we have verified the correctness of the working of the four error detection schemes and our programs.

RESULTS

We have to evaluate the performance of the four error detection schemes in comparison to one another. For this, we take a long list of random strings and then convert each character to its ASCII value. The 8-bit binary representation of the ASCII values is stored in a file and this becomes the dataset as it is sufficiently long and random.

Performance metrics for evaluation: The data packets produced from the dataset are injected with different types of errors. All the four error detection schemes are utilized one by one, randomly, at different times (so that system variables are also random). The percentage of errors detected to the total number of errors injected into the packets is tabulated. We find the average percentage of detected errors for each scheme and compare them. We also repeat the tests for different sizes of datawords. The greater the percentage of detected errors, the more efficient is the scheme. We also observe the range of observations for a particular scheme to check the reliability, e.g., a scheme which fluctuates between a wide range of percentages cannot be relied upon in real-life.

Observation Tables:

Each observation percentage is correct to two decimal places.

SIZE OF DATAWORD: 1 byte (8 bits)

- Single-bit error

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	100%	100%	100%	100%	100%
2	100%	100%	100%	100%	100%
3	100%	100%	100%	100%	100%
4	100%	100%	100%	100%	100%
5	100%	100%	100%	100%	100%
AVERAGE	**100%**	**100%**	**100%**	**100%**	**100%**

- Burst error of length 2 (consecutive 2 bits are flipped)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	0%	100%	100%	100%	100%
2	0%	100%	100%	100%	100%
3	0%	100%	100%	100%	100%
4	0%	100%	100%	100%	100%
5	0%	100%	100%	100%	100%
AVERAGE	**0%**	**100%**	**100%**	**100%**	**100%**

- Burst error of length 3

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	54.90%	100%	100%	100%	100%
2	44.55%	100%	100%	100%	100%
3	50.09%	100%	100%	100%	100%
4	49.54%	100%	100%	100%	100%
5	46.69%	100%	100%	100%	100%
AVERAGE	**49.154%**	**100%**	**100%**	**100%**	**100%**

- Burst error having a random length less than half the codeword length + 2

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	38.81%	95.37%	94.06%	99.80%	100%
2	36.66%	96.81%	96.72%	99.81%	100%
3	43.37%	96.21%	94.17%	99.60%	100%
4	42.20%	94.80%	95.46%	100%	100%
5	39.69%	96.41%	94.18%	100%	100%
AVERAGE	**40.146%**	**95.92%**	**94.918%**	**99.842%**	**100%**

- Random error (flipping bits at random)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	50.10%	93.45%	92.77%	99.52%	100%
2	48.50%	94.41%	93.45%	99.23%	99.90%
3	51.93%	93.06%	93.54%	99.61%	100%
4	50.53%	93.64%	92.77%	99.61%	100%
5	49.32%	94.31%	93.06%	99.32%	100%
AVERAGE	**50.076%**	**93.774%**	**93.118%**	**99.458%**	**99.98%**

SIZE OF DATAWORD: 2 bytes (16 bits)

- Single-bit error

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	100%	100%	100%	100%	100%
2	100%	100%	100%	100%	100%
3	100%	100%	100%	100%	100%
4	100%	100%	100%	100%	100%
5	100%	100%	100%	100%	100%
AVERAGE	**100%**	**100%**	**100%**	**100%**	**100%**

- Burst error of length 2 (consecutive two bits get flipped)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	0%	100%	100%	100%	100%
2	0%	100%	100%	100%	100%
3	0%	100%	100%	100%	100%
4	0%	100%	100%	100%	100%
5	0%	100%	100%	100%	100%
AVERAGE	**0%**	**100%**	**100%**	**100%**	**100%**

- Burst error of length 3

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	46.82%	100%	100%	100%	100%
2	50.00%	100%	100%	100%	100%
3	51.55%	100%	100%	100%	100%
4	52.03%	100%	100%	100%	100%
5	44.32%	100%	100%	100%	100%
AVERAGE	**48.944%**	**100%**	**100%**	**100%**	**100%**

- Burst error having a random length less than half the codeword length + 2

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	43.40%	95.89%	95.59%	100%	100%
2	45.12%	95.82%	94.16%	99.58%	100%
3	42.47%	95.42%	94.28%	99.61%	100%
4	48.82%	96.75%	95.45%	100%	100%
5	43.65%	92.53%	93.63%	99.62%	100%
AVERAGE	**44.692%**	**95.282%**	**94.622%**	**99.762%**	**100%**

- Random error (flipping bits at random)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	50.67%	93.06%	93.45%	99.81%	100%
2	48.75%	94.99%	94.22%	99.81%	100%
3	50.67%	93.06%	94.03%	99.61%	100%
4	48.17%	93.45%	94.22%	99.81%	100%
5	48.36%	92.68%	92.48%	99.61%	100%
AVERAGE	49.324%	93.448%	93.68%	99.73%	100%

SIZE OF DATAWORD: 3 bytes (24 bits)

- Single-bit error

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	100%	100%	100%	100%	100%
2	100%	100%	100%	100%	100%
3	100%	100%	100%	100%	100%
4	100%	100%	100%	100%	100%
5	100%	100%	100%	100%	100%
AVERAGE	100%	100%	100%	100%	100%

- Burst error of length 2 (consecutive two bits are flipped)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	0%	100%	100%	100%	100%
2	0%	100%	100%	100%	100%
3	0%	100%	100%	100%	100%
4	0%	100%	100%	100%	100%
5	0%	100%	100%	100%	100%
AVERAGE	0%	100%	100%	100%	100%

- Burst error of length 3

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	49.05%	100%	100%	100%	100%
2	53.07%	100%	100%	100%	100%
3	51.76%	100%	100%	100%	100%
4	48.43%	100%	100%	100%	100%
5	49.19%	100%	100%	100%	100%
AVERAGE	50.30%	100%	100%	100%	100%

- Burst error having a random length less than half the codeword length + 2

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	47.29%	91.65%	94.32%	99.60%	100%
2	47.97%	95.13%	93.90%	100%	100%
3	45.96%	95.65%	93.90%	99.86%	100%
4	46.10%	94.59%	93.24%	99.86%	100%
5	47.33%	94.41%	94.60%	99.86%	100%
AVERAGE	46.93%	94.286%	93.992%	99.836%	100%

- Random error (flipping bits at random)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	48.28%	93.39%	94.63%	99.59%	100%
2	51.79%	94.76%	93.80%	99.79%	100%
3	49.79%	93.32%	92.77%	99.38%	99.93%
4	48.14%	93.53%	93.25%	99.79%	100%
5	50.27%	94.76%	93.59%	99.38%	100%
AVERAGE	**49.65%**	**93.952%**	**93.608%**	**99.586%**	**99.986%**

RANDOM COMBINATION OF THE ABOVE ERRORS TO MIMIC A REAL-WORLD SCENARIO

SIZE OF DATAWORD: 1 byte (8 bits)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	58.65%	95.39%	96.40%	99.71%	100%
2	58.63%	95.33%	95.64%	99.85%	100%
3	62.72%	94.68%	94.72%	99.56%	100%
4	60.61%	95.11%	95.20%	99.85%	100%
5	57.40%	95.47%	95.49%	100%	100%
AVERAGE	**59.602%**	**95.196%**	**95.49%**	**99.794%**	**100%**

SIZE OF DATAWORD: 2 bytes (16 bits)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	55.88%	93.73%	96.15%	99.42%	100%
2	59.05%	96.02%	93.64%	100%	100%
3	58.92%	96.78%	96.34%	99.45%	100%
4	62.53%	96.80%	94.30%	99.42%	100%
5	66.47%	96.10%	96.50%	99.71%	100%
AVERAGE	**60.57%**	**95.886%**	**95.386%**	**99.60%**	**100%**

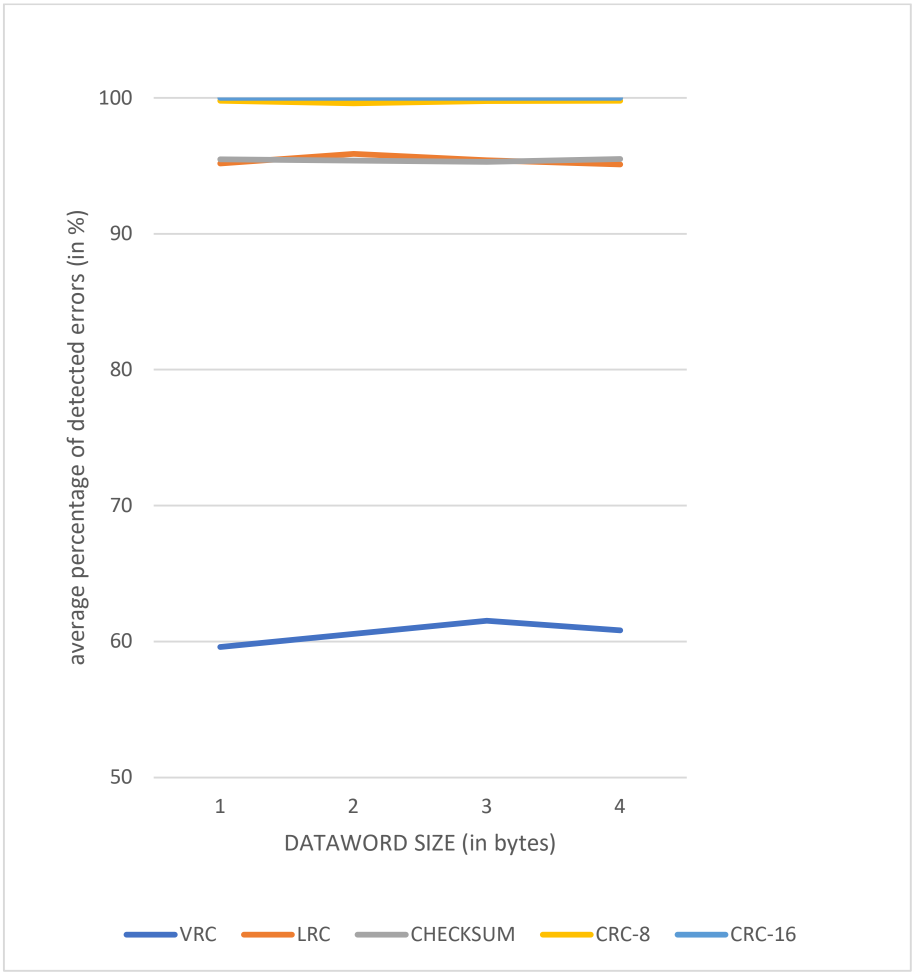
SIZE OF DATAWORD: 3 bytes (24 bits)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	64.59%	95.22%	95.53%	99.58%	100%
2	60.46%	95.16%	95.25%	100%	100%
3	60.18%	96.77%	95.35%	100%	100%
4	61.53%	94.93%	94.18%	99.79%	100%
5	60.86%	95.04%	96.18%	99.48%	100%
AVERAGE	**61.524%**	**95.424%**	**95.298%**	**99.77%**	**100%**

SIZE OF DATAWORD: 4 bytes (32 bits)

	VRC	LRC	CHECKSUM	CRC-8	CRC-16
1	60.0%	95.08%	94.93%	99.72%	100%
2	61.14%	95.30%	95.73%	99.86%	100%
3	59.47%	95.12%	95.23%	100%	100%
4	61.97%	95.73%	95.56%	100%	100%
5	61.53%	94.30%	96.15%	99.45%	100%
AVERAGE	**60.82%**	**95.106%**	**95.52%**	**99.806%**	**100%**

Graph:



The above graph shows the average percentage of detected errors for each error detection scheme for increasing size of dataword.

ANALYSIS

From the test cases and observations, we have found that all the error detection schemes can detect single-bit errors, irrespective of the dataword size. The VRC error detection scheme is not reliable as it can detect errors involving odd number of flipped bits and cannot detect errors involving even number of flipped bits. It is also not efficient as it has an around 50% accuracy for detecting random errors. LRC and Checksum both are quite efficient and reliable as they have around 95% accuracy for detecting random errors and have a narrow range in their observations. CRC error detection scheme is more efficient as it has close to 100% accuracy. However, if we use CRC, the number of redundant bits added to the dataword is more, so the data rate decreases. CRC has a limitation—it cannot detect errors if the codeword is changed in such a way that the difference between the new codeword and the original codeword is a multiple of the CRC polynomial used. These errors may be detected by VRC or by Checksum as seen in the test cases. Checksum cannot detect error if the codeword is changed in such a way that the sum of all the segments remains the same before and after. LRC cannot reliably detect burst errors having a length greater than 4 (in this implementation), however such errors might not be very common. So, CRC is the most efficient scheme, followed by Checksum and LRC, and VRC is the least efficient. However, VRC adds only one redundant bit, so if the channel only results in single-bit errors, then we are best off using VRC as the data rate is not much reduced. For burst errors having a length less than or equal to 4, LRC can be used. For burst errors having greater length, we may use Checksum if 95% accuracy is acceptable. However, if our work calls for high accuracy and this is of greater significance than the data rate, we use CRC. To highly minimize the possibility of an error going undetected, we can use CRC-16, but it adds more redundant bits, leading to a very long codeword. To compromise, we can use CRC-8, as it adds less redundant bits, but the performance (error detection) suffers a minimal loss. The different sizes of datawords do not result in significant difference in the performance of the schemes, however the accuracy of the VRC scheme jumps about a fair bit, which only shows its less reliability.

Possible Improvements: With the program as it is now, the receiver displays whether or not it has detected an error in a packet and also sends a packet to the sender containing 1 if an error was detected and 0 otherwise. As of now, the sender uses this value to calculate the percentage of detected errors to the total number of errors, but if the sender keeps track of the packets in which error was detected, then it can retransmit the respective data packets to the receiver afterwards. This can be a way to ensure the receiver receives all the data correctly in the absence of any error correction. Also, as of now, no data packet is getting lost in the channel, but in a real-world scenario, that may happen, so such a thing might be incorporated in the channel function.

COMMENTS

The lab assignment was interesting as it allowed us to see the different advantages and limitations of each error detection scheme in a practical manner and allowed us to verify the results we have read about in theory. I found the analysis of the different observations very engaging. Writing the programs was not particularly difficult after figuring out the design and the algorithms, however nor was it very easy. However, I feel that the programming was not the main focus of this assignment, but rather finding how the error detection schemes we have previously read about in our Data Communications class measure up in a simulated real-world environment. Comparing the performance of different CRC polynomials is a possibility. What to do after detecting the errors, retransmission or something else, can also be thought about. Introduction of error correction can be another possibility to be considered.