Computer Networks Lab Report- Assignment 1

TITLE:

Name: Debarghya Maitra

Class: BCSE 3rd Year

Group: A3

Submission Date: 7/8/2022

Problem Statement: Design and implement an error detection module. Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. Please note that you may need to use these schemes separately for other applications (assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame (decide the size of the frame) from the input. Based on the schemes, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases (not limited to).
(a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given in next page).
(b) Error is detected by checksum but not by CRC.
(c) Error is detected by VRC but not by CRC.
[Note: Inject error in random positions in the input data frame. Write a separate method for that.]

DESIGN:

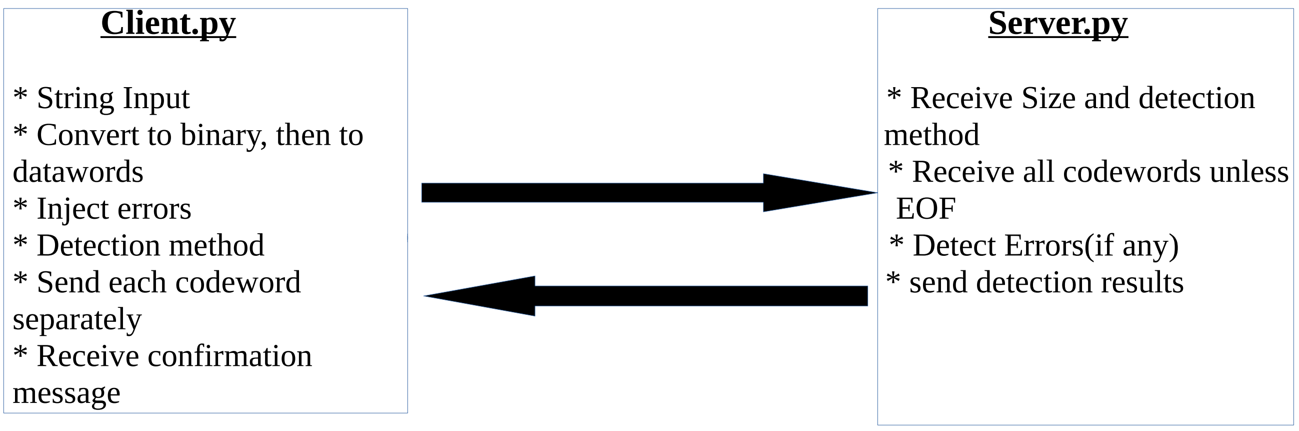
A bit, on travel, is subjected to electromagnetic interference due to noises. Thus, the data transmitted may be prone to errors. In easy words, any bit can be flipped. This happens because the voltage present in noise signals either has direct impact on the voltage present in the data signal or creates 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 capable enough to detect the error and ask the sender to retransmit the data packet.

In this assignment, we have implemented the following error detection techniques:
1. Vertical Redundancy check(VRC)
2. Longitudinal Redundancy check(LRC)
3. Checksum
4. Cyclic Redundancy check(CRC)

I have implemented the error detection module in seven program files:
*** server.py** : data receiver
*** client.py** : data sender
*** lrc.py, vrc.py, checksum.py, crc.py** : contains the classes of respective error detection methods.
*** helper.py** : definitions of basic CRC methods

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

1. **Client.py**: The following are the tasks performed in this Client program:
    a. Takes a string from stdin from user.
    b. Converts the string into binary data stream; and then breaks the stream into datawords of length 8.
    c. In case of LRC, VRC, the calculated redundant bits forms a packet and sent separately, for checksum, the calculated 1s complement sum is sent separately.
    d. In case of CRC, the redundant bits of remainder are added to each dataword and the respective codewords are sent through sockets.
    e. Before sending data, some errors are injected at random positions.
    e. Lastly, it receives a message from server if the data received was errorneous or not.
2. **Server.py**: The following are the tasks performed in this Server program:
    a. After connecting to the client, server receives the total data size, method of error detection, and all the codewords/datawords until EOF is received through sockets.
    b. Checks for errors based on the method specified.
    c. Sends the client a message indicating the status of received data.
3. **lrc.py, vrc.py, checksum.py, crc.py:** Contains classes for respective error detection methods which contains both encoding and decoding methods for a list of datawords.
4. **Helper.py:** Contains the definitions of types of CRC methods mapped to respective polynomials and a function to convert those polynomials into binary divisors.


Client.py
* String Input
* Convert to binary, then to datawords
* Inject errors
* Detection method
* Send each codeword separately
* Receive confirmation message
Server.py
* Receive Size and detection method
* Receive all codewords unless EOF
* Detect Errors(if any)
* send detection results

IMPLEMENTATION:

Code Snippet of Server.py:

```
import socket
from Crypto.Util.number import long_to_bytes, bytes_to_long
import checksum
import lrc
import vrc
import crc
s = socket.socket()
s.bind(('localhost', 9999))
s.listen(1)
print("Waiting for connections")
while True:
    c, addr = s.accept()
    print("Connection from {}".format(addr))
    size = bytes_to_long(c.recv(1024))
    method = c.recv(1024).decode('utf-8')
    data = c.recv(1024).decode('utf-8')
    li = []
    while 1:
        text = c.recv(1024).decode('utf-8')
        if 'EOF' in text:
            li.append(text.replace("EOF", ""))
            break
        li.append(text)
    received_text = ''.join(li)[:size]
    print("Data size:{}, detection method:{}".format(size, method))
    print("Received Data={}".format(received_text))
    if method == "CRC":
        if crc.CRC.checkRemainder(li, data):
            c.send(b"Received Text, No errors found by CRC")
        else:
            c.send(b"Error detected by CRC")
    elif method == "VRC":
        if vrc.VRC.check_vrc(li, data):
            c.send(b"Received Text, No errors found by VRC")
        else:
            c.send(b"Error detected by VRC")
    elif method == "LRC":
        if lrc.LRC.check_lrc(li, data):
            c.send(b"Received Text, No errors found by LRC")
        else:
            c.send(b"Error detected by LRC")
    elif method == "Checksum":
        if checksum.Checksum.check_checksum(li, data):
            c.send(b"Received Text, No errors found by Checksum")
        else:
            c.send(b"Error detected by Checksum")
    else:
        raise Exception("An error occurred while parsing data")
    c.close()
```
Code snippet of Client.py:
```
import socket
from Crypto.Util.number import bytes_to_long, long_to_bytes
import random
import time
import checksum
import lrc
import vrc
import crc
import helper
HOST = 'localhost'
PORT = 9999
PKT_SIZE = 8
c = socket.socket()
random.seed(time.time())
def inject_error(text: str, number: int) -> str:
    if number == 0:
        return text
    for i in range(number):
        x = random.randint(0, len(text)-1)
        if text[x] == '0':
            text = text[:x]+'1'+text[x+1:]
        else:
            text = text[:x]+'0'+text[x+1:]
    return text
c.connect((HOST, PORT))
text = input("Enter data:").encode('utf-8')
res = input("Do you want to insert errors?(Y/n)")
method = input("Detection Method:(CRC/VRC/LRC/Checksum)")
enc_text = bin(bytes_to_long(text))[2:]
actual_len = len(enc_text)
if actual_len/8 == 0:
    pass
else:
    extra = '0'*(8-(actual_len % 8))
    enc_text = enc_text + extra
chunks = [enc_text[i:i+PKT_SIZE] for i in range(0, len(enc_text), PKT_SIZE)]
c.send(long_to_bytes(actual_len))
time.sleep(1)
if method == "CRC":
    c.send(b"CRC")
    time.sleep(1)
    crc_method = input("Give the CRC divisor method:")
    divisor = helper.convToBinary(crc_method)
    c.send(divisor.encode('utf-8'))
    chunks = crc.CRC.encodeData(chunks, divisor)
elif method == "VRC":
    c.send(b"VRC")
    time.sleep(1)
    c.send(vrc.VRC.generate_vrc(chunks).encode('utf-8'))
elif method == "LRC":
    c.send(b"LRC")
    time.sleep(1)
    c.send(lrc.LRC.generate_lrc(chunks).encode('utf-8'))
elif method == "Checksum":
    c.send(b"Checksum")
    time.sleep(1)
    c.send(checksum.Checksum.generate_checksum(chunks).encode('utf-8'))
else:
    print("No such method!")
    c.close()
    exit(1)
for i in chunks:
    time.sleep(1)
    if res == 'y' or res == 'Y':
        j = inject_error(i, random.randint(0, 2))
        c.send(j.encode('utf-8'))
    else:
        c.send(i.encode('utf-8'))
c.send(b'EOF')
print(c.recv(1024).decode('utf-8'))
```

Method Description:

* **inject_error(text, number)** : Flips bit `number` times of random positions in `text`.

Code Snippet of vrc.py:

```
class VRC:
    @classmethod
    def generate_vrc(cls, chunks):
        res = ""
        for chunk in chunks:
            if chunk.count("1") % 2 == 0:
                res = res + "0"
            else:
                res = res + "1"
        return res
    @classmethod
    def check_vrc(cls, chunks, vrc):
        i = 0
        for chunk in chunks:
            if chunk == '':
                continue
            if (chunk.count("1") % 2 == 0 and vrc[i] == "0") or (chunk.count("1") % 2 != 0 and
vrc[i] == "1"):
                i = i+1
                continue
            else:
                return False
        return True
```

Method Description:

generate_vrc(): Takes a list of datawords, finds the parity of each and returns the parity ofeach dataword collectively.

check_vrc(): Takes a list of datawords and the collective vrc bits and checks for errors.

Code snippet of lrc.py:

```
class LRC:
    @classmethod
    def generate_lrc(cls, chunks):
        res = ""
        for i in range(len(chunks[0])):
            count = 0
            for chunk in chunks:
                if chunk[i] == '1':
                    count = count+1
            if count % 2 == 0:
                res = res+"0"
            else:
                res = res+"1"
        return res
    @classmethod
    def check_lrc(cls, chunks, lrc):
        for i in range(len(chunks[0])):
            count = 0
            for chunk in chunks:
                if chunk == '':
                    continue
                if chunk[i] == '1':
                    count = count+1
            if (count % 2 == 0 and lrc[i] == '0') or (count % 2 != 0 and lrc[i] == '1'):
                continue
            else:
                return False
        return True
```

Method Description:

generate_lrc(): Takes a list of datawords, finds the parity of each column and returns the parity bits collectively.

check_lrc(): Takes a list of datawords and the collective lrc bits and checks for errors.

Code snippet of Checksum.py:

```
class Checksum:
    @classmethod
    def generate_checksum(cls, chunks):
        res = ""
        size = len(chunks[0])
        for chunk in chunks:
            res = bin(int(res, 2)+int(chunk, 2)) if res != "" else chunk
        res = res[2:]
        res = bin(int(res[-size:], 2)+int(res[:-size], 2))
        return ''.join('1' if x == '0' else '0' for x in res[2:])
    @classmethod
    def check_checksum(cls, chunks, checksum):
        res = ""
        size = len(chunks[0])
        for chunk in chunks:
            res = bin(int(res, 2)+int(chunk, 2)) if res != "" else chunk
        res = res[2:]
        res = bin(int(res[-size:], 2)+int(res[:-size], 2)+int(checksum, 2))
        if res.count("1") == size:
            return True
        else:
            return False
```

Method descriptions:

generate_checksum(): Calculates the 1s complement sum of a list of datawords and returns the result.

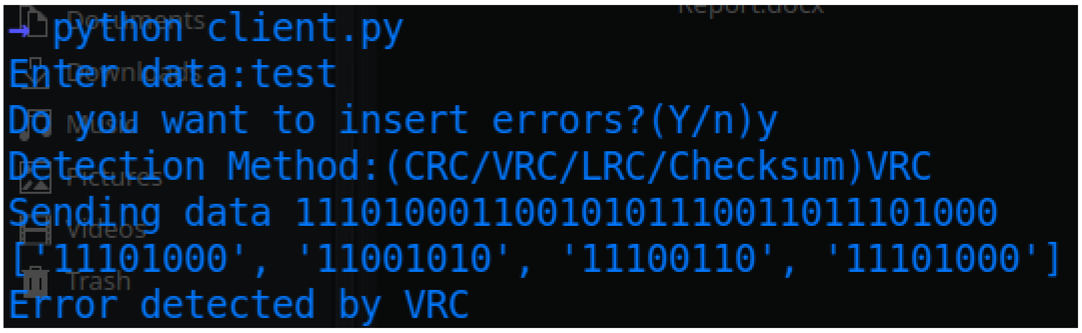
check_checksum(): Checks if the given checksum matches or not.

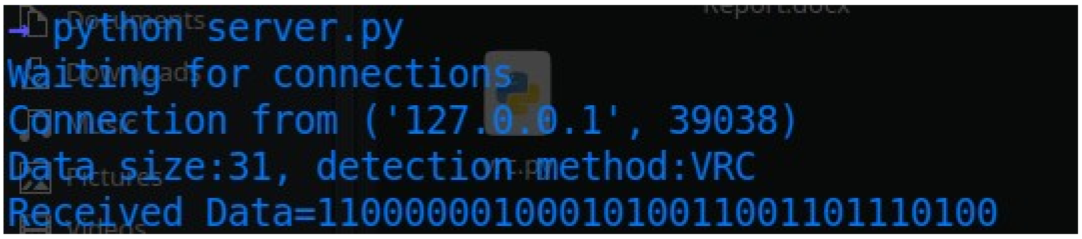
Code snippet of crc.py:

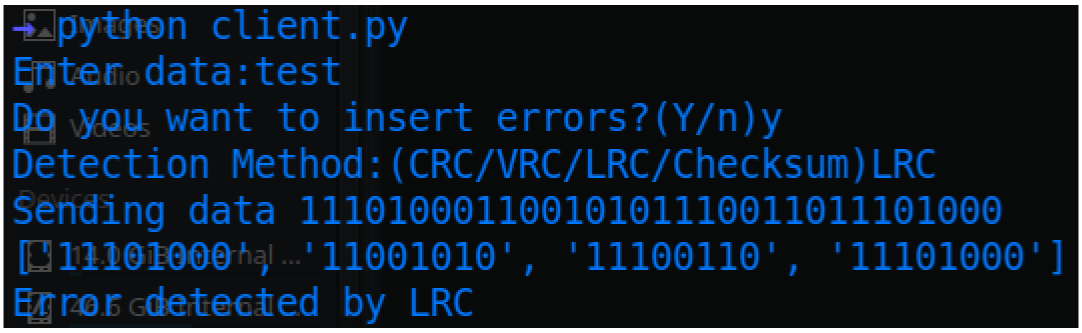
```
class CRC:
    @classmethod
    def xor(cls, a, b):
        result = []
        for i in range (1, len(b)):
            if a[i] == b[i]:
                result.append ('0')
            else:
                result.append ('1')
        return ''.join(result)
    @classmethod
    def mod2div(cls, dividend, divisor):
        pick = len(divisor)
        tmp = dividend[0 : pick]
        while pick < len(dividend):
            if tmp[0] == '1':
                tmp = cls.xor(divisor, tmp) + dividend[pick]
            else:
                tmp = cls.xor('0'*pick, tmp) + dividend[pick]
            pick += 1
        if tmp[0] == '1':
            tmp = cls.xor(divisor, tmp)
        else:
            tmp = cls.xor('0'*pick, tmp)
        checkword = tmp
        return checkword
    @classmethod
    def encodeData(cls, chunks, key):
        l_key = len(key)
        enc_chunks = []
        for data in chunks:
            appended_data = data + '0'*(l_key-1)
            remainder = cls.mod2div(appended_data, key)
            codeword = data + remainder
            enc_chunks.append(codeword)
        return enc_chunks
    @classmethod
    def checkRemainder (cls, chunks, key):
        l_key = len(key)
        for data in chunks:
            appended_data = data + '0'*(l_key-1)
            remainder = cls.mod2div(appended_data, key)
            if remainder.count('1') != 0:
                    return False
        return True
```
Method Description:
xor(): Calculates exclusive OR of given two binary strings
mod2div(): Modulo two division to calculate the remainder after dividing with a certain divisor.
EncodeData(): Encodes a list of datawords into a list of codewords.
CheckRemainder(): Checks if the remainder part in the codewords are correct or not.

TEST CASES:

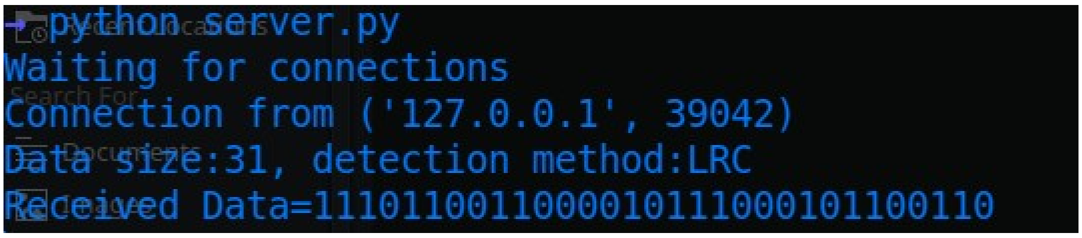
a) Errors Detected By All Methods:


```
→ python client.py
Enter data:test
Do you want to insert errors?(Y/n)y
Detection Method:(CRC/VRC/LRC/Checksum)VRC
Sending data 11101000110010101110011011101000
['11101000', '11001010', '11100110', '11101000']
Error detected by VRC
```

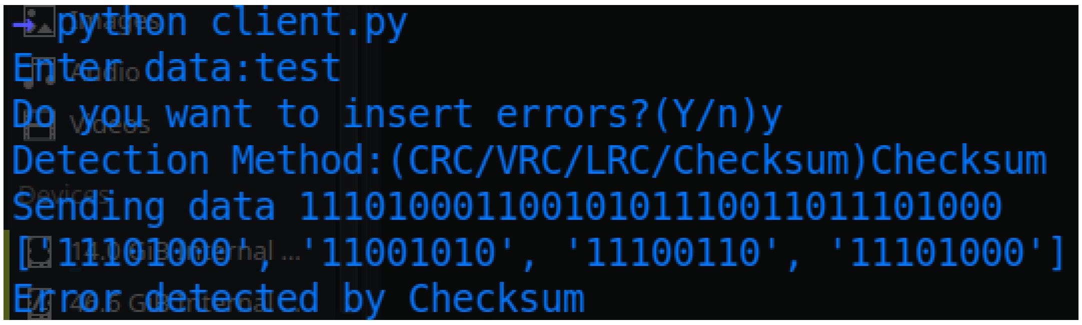

```
→ python server.py
Waiting for connections
Connection from ('127.0.0.1', 39038)
Data size:31, detection method:VRC
Received Data=1100000010001010011001101110100
```



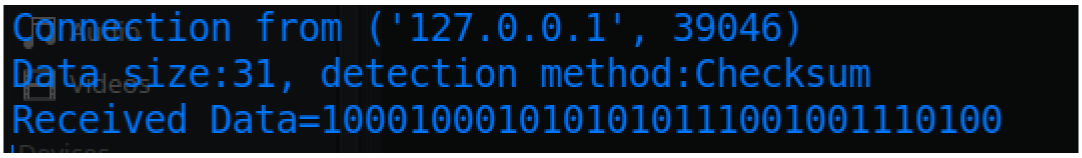
```
→ python client.py
Enter data:test
Do you want to insert errors?(Y/n)y
Detection Method:(CRC/VRC/LRC/Checksum)LRC
Sending data 1110100011001010111001101110100
['11101000', '11001010', '11100110', '11101000']
Error detected by LRC
```



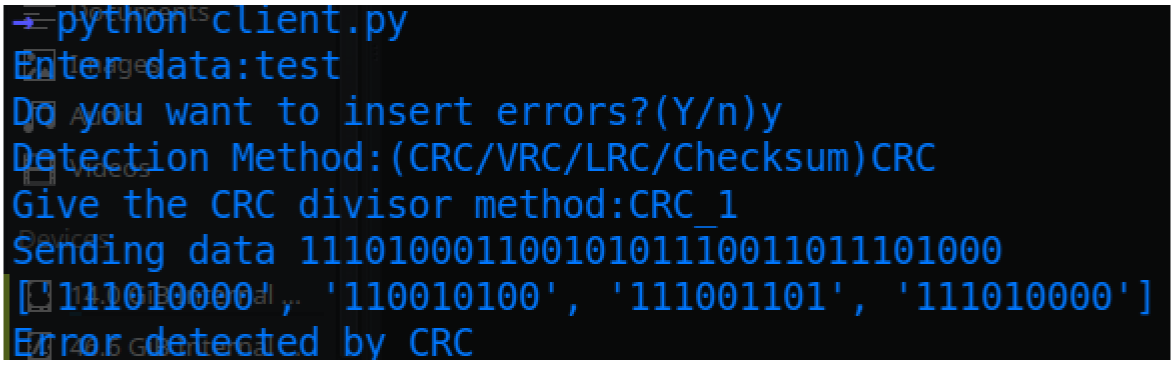
```
→ python server.py
Waiting for connections
Connection from ('127.0.0.1', 39042)
Data size:31, detection method:LRC
Received Data=1110111001100001011100010110011
```



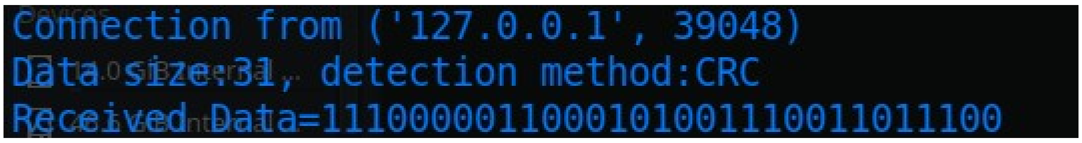
```
→ python client.py
Enter data:test
Do you want to insert errors?(Y/n)y
Detection Method:(CRC/VRC/LRC/Checksum)Checksum
Sending data 1110100011001010111001101110100
['11101000', '11001010', '11100110', '11101000']
Error detected by Checksum
```



```
Connection from ('127.0.0.1', 39046)
Data size:31, detection method:Checksum
Received Data=1000100010101010111001001110100
```

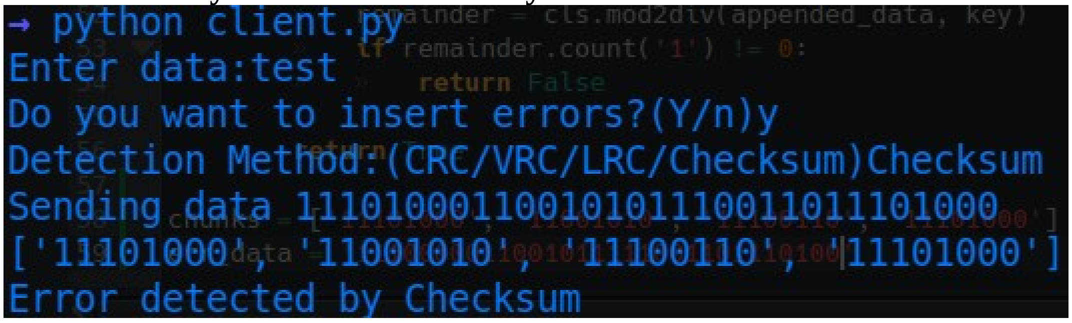


```
→ python client.py
Enter data:test
Do you want to insert errors?(Y/n)y
Detection Method:(CRC/VRC/LRC/Checksum)CRC
Give the CRC divisor method:CRC_1
Sending data 1110100011001010111001101110100
['111010000', '110010100', '111001101', '111010000']
Error detected by CRC
```

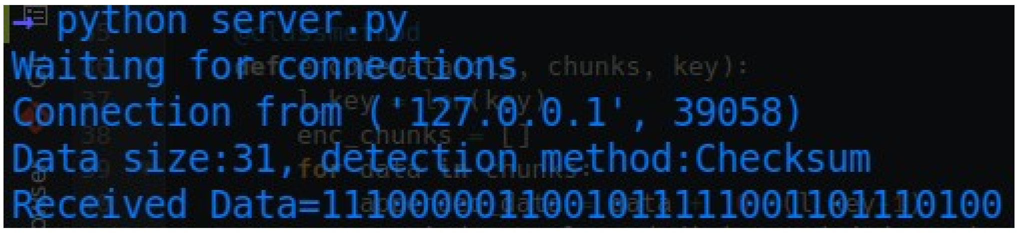


```
Connection from ('127.0.0.1', 39048)
Data size:31, detection method:CRC
Received Data=1110000011000101001110011011100
```

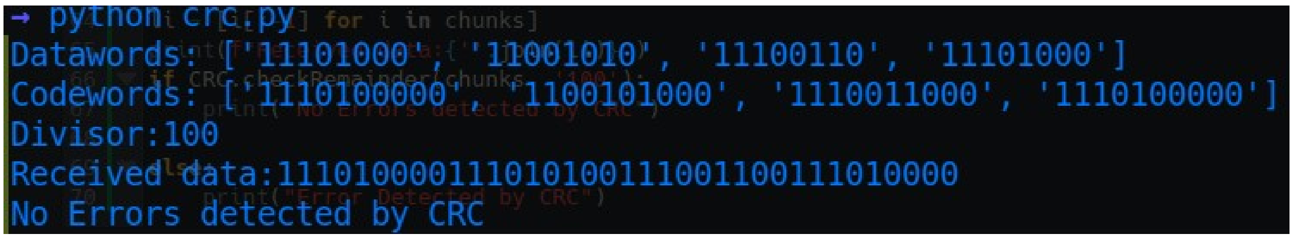
b) Error is detected by checksum but not by CRC:



```
→ python client.py
Enter data:test
Do you want to insert errors?(Y/n)y
Detection Method:(CRC/VRC/LRC/Checksum)Checksum
Sending data 1110100011001010111001101110100
['11101000', '11001010', '11100110', '11101000']
Error detected by Checksum
```

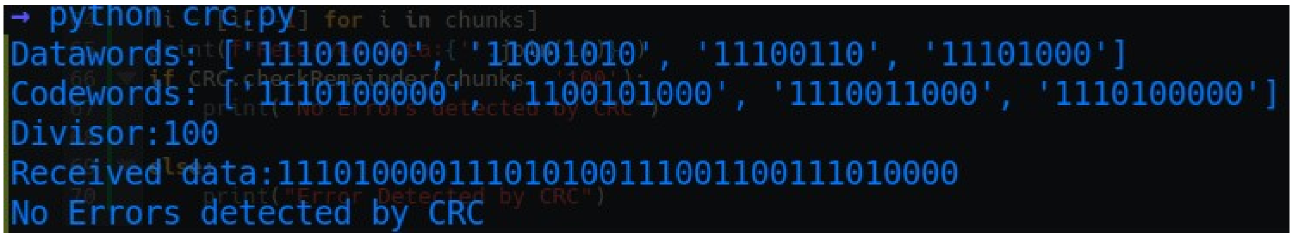


```
→ python server.py
Waiting for connections
Connection from ('127.0.0.1', 39058)
Data size:31, detection method:Checksum
Received Data=1110000011001011111001101110100
```

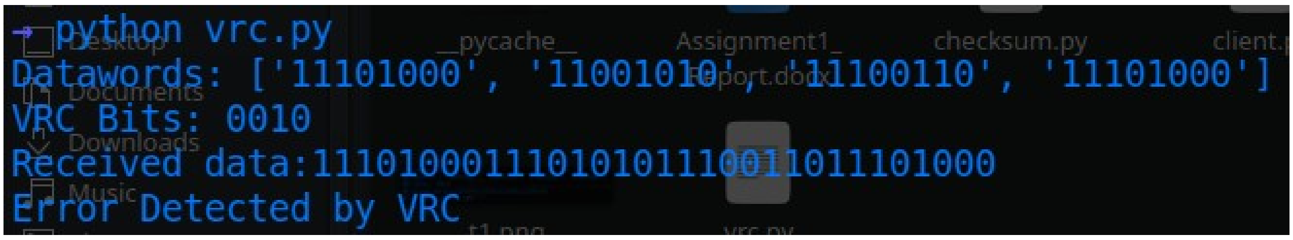


```
→ python crc.py
Datawords: ['11101000', '11001010', '11100110', '11101000']
Codewords: ['1110100000', '1100101000', '1110011000', '1110100000']
Divisor:100
Received data:111010000111010100111001100111010000
No Errors detected by CRC
```

c) Error is detected by VRC but not by CRC:



```
→ python crc.py
Datawords: ['11101000', '11001010', '11100110', '11101000']
Codewords: ['1110100000', '1100101000', '1110011000', '1110100000']
Divisor:100
Received data:111010000111010100111001100111010000
No Errors detected by CRC
```



```
→ python vrc.py
Datawords: ['11101000', '11001010', '11100110', '11101000']
VRC Bits: 0010
Received data:11101000111010101110011011101000
Error Detected by VRC
```

RESULTS:

For each type of error detection methods there are certain advantages as well as disadvantages discussed below:

a. Vertical Redundancy Check(VRC): Since parity for each data words are calculated separately, it can detect any single bit errors as well as odd number of burst errors, but can't detect any even number of burst errors, for example:

The dataword:11101001 has 5 1's hence VRC is 1, if errorneous dataword be like: 11111111, then it can easily detect the error as it has 8 1's. But if the errorneous dataword be like: 11101111, it will not detect errors as it has 7 1's.

b. Longitudinal Redundancy Check(LRC): There may be 2 cases where LRC cannot detect errors, if 2 datawords have flipped bits in the same position like 1110 and 1001 becomes 1111 and 1000, for the last bit LRC remains same; and another case would be if even number of datawords have error in the same position.

c. Checksum: Checksum may also have 2 type of cases where the 1's complement sum remains the same but there are errors in the sent data:

i) If bits in the same position are flipped, for example: 1110 and 1001 has checksum 0111 but 1111 and 1000 also has the same checksum.

ii) If two bits having 0 are flipped to 1 and in the more significant bit a 1 is turned to 0, the sum will still remain the same.

d. Cyclic Redundancy Check(CRC): 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 101010. After appending CRC bits (in this case 0 0) rdwe get 10101000. Assuming during the data transmission, the 3 bit is in error and the message received at the receiver is 10101100. On performing CRC check on 10101100, 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.
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.

ANALYSIS:

a. Error detection capabilities of the code is increased significantly when all 4 schemes are used. Approximately 3.2% of the time, error is not detected by any of the schemes.
b. The errors introduced are random and erratic, so the results obtained are subject to experimental errors. There can be cases where a same bit is flipped an even number of times and hence no error is introduced by the inject_error( ) function. Such cases have been ignored and the program assumes that the function is indeed able to introduce errors in all codewords.
c. Since sockets are used for data transfer, the incomplete transfer of socket buffers can cause some undesired results, hence before sending each data element, the process is suspended for 1second, which again slows down the whole process

COMMENTS:

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