**CYCLIC REDUNDANCY CHECK (CRC)**

**Submitted by**

**Linet M Shaji**

**RegNo: P191314**

**1st M. Sc. Computer Science**

**CUTN, Thiruvarur**

**Introduction**

The goal of this project is to implement **Cyclic Redundancy Check client-server program** **using python**.

Redundancy Check is a method of detecting accidental changes/errors in the communication channel. CRC uses **Generator Polynomial**which is available on both sender and receiver side. An example generator polynomial is of the form like x3 + x + 1. This generator polynomial represents key 1011.

This project contains code for both client and server. **Client side generates Encoded Data from Data and Generator Polynomial (or Key).** **Server Side** calculates the **CRC of the received data**. If the CRC calculated by the destination device does not match the one calculated by the client, then the received data contains an error.CRC or Cyclic.

**Cyclic Redundancy Check (CRC)**

Redundancy Check is a method of detecting accidental changes/errors in the communication channel. CRC uses **Generator Polynomial**which is available on both sender and receiver side. An example generator polynomial is of the form like x3 + x + 1. This generator polynomial represents key 1011.

n: Number of bits in data to be sent from client side

k: Number of bits in the key obtained from generator polynomial

**Sender Side (Generation of Encoded Data from Data and Generator Polynomial (or Key)):**

1. The binary data is first augmented by adding k-1 zeros in the end of the data
2. Use ***modulo-2 binary division*** to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same

.

**Receiver Side (Check if there are errors introduced in transmission)**

Perform modulo-2 division again and if remainder is 0, then there are no errors.

In this article we will focus only on finding the remainder i.e. check word and the code word.

**Modulo 2 Division:**

The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

* + - In each step, a copy of the divisor (or data) is XORed with the k bits of the dividend (or key).
    - The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
    - When there are no bits left to pull down, we have a result. The (n-1)-bit remainder which is appended at the sender side.

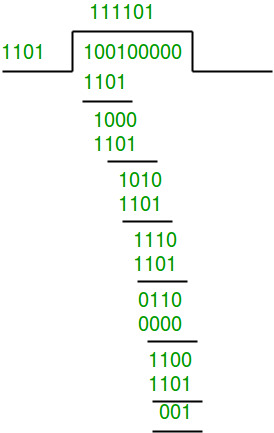
**Illustration:**

**Example 1 (No error in transmission):**

Data word: 100100

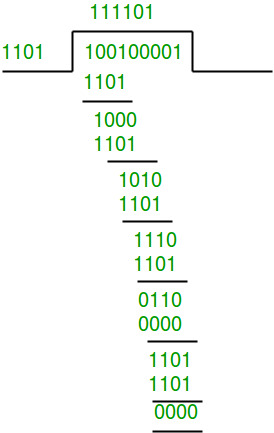
Key-1101[ or generator polynomial x^3+x^2+1]

**Sender side:**

  
Therefore, the remainder is 001 and hence the encoded data send by the client is 100100001

**Receiver Side**

Data received from the client 100100001



Therefore, the remainder is all zeros. Hence, data received has no error.

**About the code**

**Client side**

import socket

def xor(a, b):

# initialize result

result = []

# Traverse all bits, if bits are

# same, then XOR is 0, else 1

for i in range(1, len(b)):

if a[i] == b[i]:

result.append('0')

else:

result.append('1')

return ''.join(result)

# Performs Modulo-2 division

def mod2div(divident, divisor):

# Number of bits to be XORed at a time.

pick = len(divisor)

# Slicing the divident to appropriate

# length for particular step

tmp = divident[0 : pick]

while pick < len(divident):

if tmp[0] == '1':

# replace the divident by the result

# of XOR and pull 1 bit down

tmp = xor(divisor, tmp) + divident[pick]

else: # If leftmost bit is '0'

# If the leftmost bit of the dividend (or the

# part used in each step) is 0, the step cannot

# use the regular divisor; we need to use an

# all-0s divisor.

tmp = xor('0'\*pick, tmp) + divident[pick]

# increment pick to move further

pick += 1

# For the last n bits, we have to carry it out

# normally as increased value of pick will cause

# Index Out of Bounds.

if tmp[0] == '1':

tmp = xor(divisor, tmp)

else:

tmp = xor('0'\*pick, tmp)

checkword = tmp

return checkword

# Function used at the sender side to encode data by appending remainder of modular division

# at the end of data.

def encodeData(data, key):

l\_key = len(key)

# Appends n-1 zeroes at end of data

appended\_data = data + '0'\*(l\_key-1)

remainder = mod2div(appended\_data, key)

# Append remainder in the original data

codeword = data + remainder

return codeword

# Create a socket object

s = socket.socket()

# Define the port on which you want to connect

port = 12345

# connect to the server on local computer

s.connect(('127.0.0.1', port))

# Send data to server 'Hello world'

input\_string = input("Enter data you want to send->")

data =(''.join(format(ord(x), 'b') for x in input\_string))

print(data)

key = "1001"

ans = encodeData(data,key)

print(ans)

s.sendall(ans.encode('utf-8'))

# receive data from the server

print("Recieved data from server")

print(s.recv(1024))

# close the connection

s.close()

**Server Side**

import socket

def xor(a, b):

# initialize result

result = [ ]

# Traverse all bits, if bits are

# same, then XOR is 0, else 1

for i in range(1, len(b)):

if a[i] == b[i]:

result.append('0')

else:

result.append('1')

return ''.join(result)

# Performs Modulo-2 division

def mod2div(divident, divisor):

# Number of bits to be XORed at a time.

pick = len(divisor)

# Slicing the divident to appropriate

# length for particular step

tmp = divident[0 : pick]

while pick < len(divident):

if tmp[0] == '1':

# replace the divident by the result

# of XOR and pull 1 bit down

tmp = xor(divisor, tmp) + divident[pick]

else: # If leftmost bit is '0'

# If the leftmost bit of the dividend (or the

# part used in each step) is 0, the step cannot

# use the regular divisor; we need to use an

# all-0s divisor.

tmp = xor('0'\*pick, tmp) + divident[pick]

# increment pick to move further

pick += 1

# For the last n bits, we have to carry it out

# normally as increased value of pick will cause

# Index Out of Bounds.

if tmp[0] == '1':

tmp = xor(divisor, tmp)

else:

tmp = xor('0'\*pick, tmp)

checkword = tmp

return checkword

# Function used at the receiver side to decode

# data received by sender

def decodeData(data, key):

l\_key = len(key)

# Appends n-1 zeroes at end of data

appended\_data = data.decode('utf-8') + '0'\*(l\_key-1)

remainder = mod2div(appended\_data, key)

return remainder

# Creating Socket

s = socket.socket()

print ("Socket successfully created")

# reserve a port on your computer in our

# case it is 12345 but it can be anything

port = 12345

s.bind(('', port))

print ("socket binded to %s" % (port))

# put the socket into listening mode

s.listen(5)

print ("socket is listening")

while True:

# Establish connection with client.

c, addr = s.accept()

print('Got connection from', addr)

# Get data from client

print("Recieved data from client")

data = c.recv(1024)

print(data)

if not data:

break

key = "1001"

ans = decodeData(data, key)

print("Remainder after decoding is->"+ans)

print("Error not found")

# If remainder is all zeros then no error occured

temp = "0" \* (len(key) - 1)

if ans == temp:

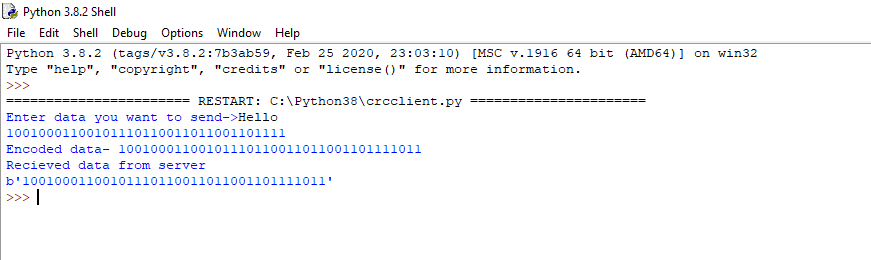
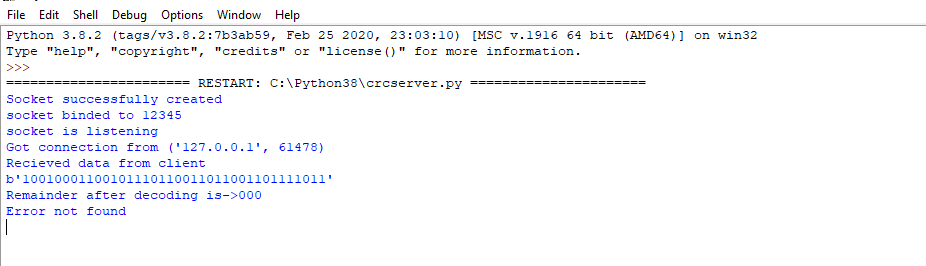
c.sendall(data )

else:

c.sendall("Error in data")

c.close()

**Screenshots**

**Conclusion**

Working on the implementation of this project given better understanding on Cyclic Redundancy Check, how they work and using python for this implementation also helped me to understand the python programming concepts better, knowing what to use for what and how to go about certain programming concepts in python. We learnt to efficiently validate our inputs and assert our outputs using python.