ECS301: Principles of Communications

P42: Studying single parity check codes and its error correction

Akshat Singh
20031

Department of EECS
IISER Bhopal
India
Date: 03/11/2022

1 Introduction

Single Parity Check is a popular technique used to detect a single-bit error in the transmitted data; in this technique, the parity of any one bit (say '1') is kept either even or odd during transmission and is rechecked while receiving the data. Error is detected whenever the parity of the selected bit is different.
The limitation of this technique is that it can only be used to check single-bit errors, and whenever there is a burst error, this technique fails.

1.1 Error

Whenever data is transmitted from a device to another, it can not be guaranteed that the data originally transmitted will be the same as data received. Whenever this type of situation occurs when the received data is different from the transmitted data, it is called an error

Errors can be classified into 2 types -

1. **Single-Parity Error:** Error in only one bit of the data.
2. **Burst Error:** Error in multiple bits of the data.

This report will focus solely on the Single-Parity Errors and their detection and corrections.

1.2 Single-Bit error

The term single-bit error means that only one bit of the given data is changed from 0 to 1 or 1 to 0. These are one of the least likely type of errors in data transmission as most errors are burst errors.

1.3 Parity

Parity means the total count of any bit (0 or 1) is either even or odd.
They are of two types -

1. **Even Parity:** When number of 1's (or 0's) in the data is even, we call it even parity of the bit 1 (or 0).
2. **Odd Parity:** When number of 1's (or 0's) in the data is odd, we call it odd parity of the bit 1 (or 0).

2 Methods

There are multiple methods for detecting and correcting single-parity errors, here in this report only 2 of them will be discussed. They are -

2.1 Simple-parity check

This method, as the name suggests, is a very simple approach of finding parity error.
With this approach, first the number of 1's (or 0's) in the data to be transmitted are counted, and then their parity is checked. One of the parity (even or odd) is selected for that bit which is known to both sender and the receiver. In this report, the bit selected is 1 and its parity is considered even but the same can be applied for 0 and odd.
After counting the number of 1's, if it is odd, 1 is added to the data to make the parity of 1 even, this extra bit added is called the ***parity-bit***. Similarly, if the number of 1's in the data is even, the parity bit is set to 0 and added to the data.

For e.g.: Let's assume the data to be transmitted is **100101**, here the number of 1's are 3 which is odd. So the parity-bit will be 1 and the final data will be **1001011**.
At the receiver's end, the parity of 1 is checked, if the parity remains 1, there is no single-bit error in the data. But if the parity is odd, an error is reported in the data and it is transmitted again until there is no error. Following is the python implementation of this method including error during transmission, files will be attached to this pdf for testing -

```
1         import random
2
3         binaryNumber = '' # Global variable for binary number
4
5         # Checking even parity of 1, set as true initially number
      of 1 initially is even(0)
6         evenParity = True
7
8         # Sender's end
9         while True: # Loop to check if there is a wrong binary
      input
10
11            binaryNumber = input("Enter binary number ")
12
13            correctInput = True
14
15            for num in binaryNumber:
16                if (num == '0'):
17                    # if current bit is 0 do nothing
18                    continue
19                elif (num == '1'):
20                    # if current bit is 1, change parity from even
      to odd and odd to even
21                    evenParity = not evenParity
22                else:
23                    # if input is not 0 or 1, ask for input again
24                    correctInput = False
25                    break
26
27            if (correctInput == True):
```

```
                break

            print("Wrong Input! Try again")

        parityBit = '' # global variable for the bit to be added to
    the binary string

        # keeping the parity of 1 even
        if (evenParity):
            parityBit = '0'
        else:
            parityBit = '1'

        # adding the parity bit to binary number
        binaryNumber += parityBit

        print("Transmitted binary string is: " + binaryNumber)

        # Randomly adding noise to the string

        i = random.randint(0, len(binaryNumber) - 1)

        charList = list(binaryNumber)
        charList[i] = random.randint(0,1)
        binaryNumber = ''.join(str(e) for e in charList)




        # Receiver's end
        checkParity = True

        for bit in binaryNumber:
            if (num == '0'):
                continue
            else:
                checkParity = not checkParity

        print("Received binary string is: " + binaryNumber)

        if (checkParity):
            print("No errors in parity of 1")
        else:
            print("There is error in parity of 1")

```

The code is divided into three sections, each starting with a comment. Initially the random library is imported for adding noise in the second section and two global variables are initialised namely *binaryNumber* and *evenParity*. The data to be transmitted will be stored in the 'binaryNumber' variable as a string and the 'evenParity' will check the parity of 1 in the data as a boolean; it is initialised to True as the number of 1's initially are 0 which is even. The code is divided into three sections -

2.1.1 Sender's End

In this section, a while loop is run to take input from the user to get the binary string to be transmitted, this loop will count the number of 1's in the data and accordingly set the value of 'evenParity' as well as check if the data is binary or not, in case it is not binary the loop will run again asking for another input from the user.
Following this the code checks what the value of the parity bit will be according to the 'evenParity' variable and add this parity bit to the original data.

2.1.2 Adding Noise to the data

In this section, an index less than the length of the binary data is selected randomly and its value is randomly changed to either 0 or 1, this will make sure that the data has random noise and even the correct data can be transmitted.

2.1.3 Receiver's End

In this section, the parity of the received data is checked again; if the parity of 1 in this data is even then it is concluded that there are no errors, but if the parity of 1 is odd then it is concluded that there is an error in the received data.

The code above is just a simulation of what happens in real life, if an error like this is detected, the data can be sent again until no error is detected.

2.1.4 Outputs

Using another python script to run this code 'n' times, the number of steps taken to send the correct data was taken for binary numbers 1 to 10000. *For e.g.* The binary representation of 1021 If **1111111101** so we take $i = 1021$, send it with random noise, and it takes 3 steps for the program to send the correct data,then $k_{1021} = 3$. Similarly, taking the average using the formula -

$$\frac{\sum_{i=1}^{n} k_i}{n} \tag{1}$$

Upon calculating the average for 10,000 numbers 10 times, it was -
0.9847, 0.9971, 0.9932, 1.0215, 1.0096, 0.9952, 0.9807, 1.0142, 1.0062, 1.0093

2.1.5 Observations

The average of 10,000 numbers lies around 1, which implies that on an average, only 1 single-parity errors occur during transmission, so single-parity errors have very low chance of happening and correcting them would not take too long.

2.2 Hamming Codes

Hamming codes are very useful as they detect as well as correct the data so sending data multiple times is not required. They use multiple parity bits instead of a single parity bit and these bits are placed in the positions of powers of '2'.
The number of these redundant parity bits 'r' is calculated using the following relation -

$$2^r \geq n + r + 1 \tag{2}$$

Where 'n' is the number of bits in the sent *data*
'r' is the number of redundant parity bits
The minimum value of 'r' is picked from the relation for space efficiency but any value of 'r' for which the relation satisfies can be picked.

Let the Hamming code is $b_{n+k}b_{n+k1}.....b_3b_2b_1$ and parity bits p_k,p_{k1},....p_1 . We can place the 'k' parity bits in powers of 2 positions only. In remaining bit positions, we can place the 'n' bits of binary code.

Based on requirement, we can use either even parity or odd parity while forming a Hamming code. But, the same parity technique should be used in order to find whether any error present in the received data.

2.2.1 Finding Parity Bits

Follow this procedure for finding **parity bits** -

- Find the value of p_1, based on the number of ones present in bit positions b_3, b_5, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^0.
- Find the value of p_2, based on the number of ones present in bit positions b_3, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^1.
- Find the value of p_3, based on the number of ones present in bit positions b_5, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^2.
- Similarly find the other parity bits if there is a need of it.

2.2.2 Finding Check Bits

Follow this procedure for finding **check bits** -

- Find the value of c_1, based on the number of ones present in bit positions b_1, b_3, b_5, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^0.

- Find the value of c_2, based on the number of ones present in bit positions b_2, b_3, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^1.
- Find the value of c_3, based on the number of ones present in bit positions b_4, b_5, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have '1' in the place value of 2^2.
- Similarly find the values of other check bits

2.2.3 Codes

The following code runs this algorithm -

```
import random
from typing import List

# Using Hamming codes to detect and correct errors

# Function for calculating the redundant bits in the given binary string
# using formula 2^r >= m + r + 1
def RedundantBits(m:int):
    for r in range(m):
        if (2**r >= m + r + 1):
            return r # returns the number of redundant bits

# Finding a modified string with number of bits = n + r + 1
def PositionRedundantBits(givenString, r):
    j = 0
    k = 1
    m = m = len(givenString)
    par = ''


    for i in range(1, m + r + 1):
        if (2**j == i):
            par = par + '0'
            j += 1
        else:
            # Appending the data backwards because 0th position in binary is last in string
            par = par + givenString[-1 * k]
            k += 1

    # Reverse modified string
    return par[::-1] # returns the modified string with position of parity bits in reverse order

# Calculating and adding parity bits to their specified position
def CalculateParityBits(modifedString, r):
    n = len(modifedString)

    for i in range(r):
        binVal = 0
        for j in range(1, n + 1):
```

```
            if (j & (2**i) == ((2**i))):
                # if position has 1 in significant position, bitwise OR is applied
                binVal = binVal ^ int(modifedString[-1 * j])

        # Concatinating string from 0 to n - 2^r + parity bit + (n - 2^r + 1 to n)
        modifedString = modifedString[:n-(2**i)] + str(binVal) + modifedString[n - (2**i) + 1:]

    return modifedString # returns the modified string with correct parity bits at their specified places


# Detecting error in recieved string
def DetectError(modifiedString, numberOfRedBit):
    n = len(modifiedString)
    res = 0

    # Does the same thing as checking for parity bits
    for i in range(numberOfRedBit):
        val = 0
        for j in range(1, n + 1):
            if (j & (2**i) == (2**i)):
                val = val ^ int(modifiedString[-1 * j])

        res = res + val * (10**i)

    return int(str(res),2) # Returns the position of error bit or 0 if there is no error



# Function for adding single-bit error
def AddRandomNoise(binaryNumber):
    i = random.randint(0, len(binaryNumber) - 1)

    charList = list(binaryNumber)
    charList[i] = random.randint(0,1)
    binaryNumber = ''.join(str(e) for e in charList)
    return binaryNumber


############################################################
# Main Code
############################################################


data = ''
while True: # Loop to check if there is a wrong binary input
    data = input("Enter binary number ")


    correctInput = True

    for num in data:
        if (num == '0' or num == '1'):
```

```
            # if current bit is 0 do nothing
            continue
        else:
            # if input is not 0 or 1, ask for input again
            correctInput = False
            break

    if (correctInput == False):
        print("Wrong Input! Try again")
    else:
        break


m = len(data)
r = RedundantBits(m)
modifiedString = PositionRedundantBits(data, r)
modifiedString = CalculateParityBits(modifiedString, r)


print("Transmitted data is: " + modifiedString)

modifiedString = AddRandomNoise(modifiedString)

correct = DetectError(modifiedString, r)
if correct == 0:
    pass
else:
    listString = list(modifiedString)

    if (listString[len(listString) - correct] == 0):
        listString[len(listString) - correct] = '1'
    else:
        listString[len(listString) - correct] = '0'

    modifiedString = "".join(listString)

print("Recieved data is: " + modifiedString)


```

The comments above the code specifies the workings of the functions.

2.3 Output

Each time on entering a binary number, it sends and returns the same modified string which can be converted back to the original binary number using the same method to modify it. On running another script which converts the numbers 1 to 10,000 into binary and then runs through this whole code shows no error in the transmitted and received data.

2.3.1 Observations

Although this method seems the most reliable for finding and correcting single-bit error, it can be observed that for large strings of binary numbers, there will be a larger number of redundant bits. So calculating those redundant bits

will take longer time and will require more space, which is not efficient as we checked earlier from simple-parity check code that on an average data has to be transferred only 2 times (1 for first time and 1 for error), so unless small data is to be sent at lower latency, it should not be used.

2.4 Other Methods

There are various other methods like the quantum algorithm for single parity check code but they are out of scope of this report so nothing more will be talked about them.

3 Real Life Applications

Single-parity error detection and corrections codes are used in almost all of the fields of computer and networking either directly or as multi-bit error correcting codes which derived from these codes.
Here are a few fields in which they are used -

3.1 Internet

In a typical TCP/IP stack, many error detecting and correcting codes are used at different levels, form ethernet frame, IPv4 header to UDP and TCP have some form of error detection or correction code including single-parity check codes.

3.2 Deep-space telecommunications

Error Detection and correction codes are very important in deep-space telecommunications and were used in many missions like the voyager 1 and voyager 2. There is extreme dilution of signal power over interplanetary distances and limited power availability, and with long distances comes a lot of noise which can be corrected using error detecting and correcting codes.

3.3 Satellite broadcasting

Same as deep-space telecommunication, error detecting and correcting codes can be used here for correcting the noise over long distances.

3.4 Data storage

Error detection and correction codes are often used to improve the reliability of data storage media. A parity track capable of detecting single-bit errors was present on the first magnetic tape data storage in 1951. Many different types of error detecting and correcting codes are now used to improve the reliability of data storage media.

This type of error is generally caused by natural phenomena such as thermal noise, power noise, cross talk, attenuation, and other forms of electromagnetic interference. Here is a peculiar example of single-bit error -

In 2013, during a Super Mario 64 70 star speedrun race against MidBoss, DOTA_Teabag encountered a glitch, widely considered to be completely impossible in the game: an up-warp without any ceiling which could be grabbed. This means that Mario was teleported extremely high into the air, an act that is typically only possible under very particular conditions, in a situation where none of those conditions were being met. The clip caught the attention of prominent Super Mario 64 player pannenkoek12, who put a $1000 bounty for anyone who could figure out how to replicate the glitch, but no one could replicate the glitch by any means.
Later it was found out that during the race, an ionizing particle from outer space collided with DOTA_Teabag's N64, flipping the eighth bit of Mario's first height byte. Specifically, it flipped the byte from **11000101** to **11000100**, from "C5" to "C4". This resulted in a height change from C5837800 to C4837800, which by complete chance, happened to be the exact amount needed to warp Mario up to the higher floor at that exact moment, this was tested and verified by by pannenkoek12 himself who put up the bounty - using a script that manually flipped that particular bit at the right time, confirming the suspicion of a bit flip. To know more about the example, refer to this link or watch this video

4 Conclusion

Even though single-parity check codes are used less in the fields of communication and networking, they are important as the multiple-bit errors are built upon these single-parity check codes. These codes are still implemented on devices so that small errors can be checked and corrected efficiently instead of using sophisticated ways of multiple-bit detection and corrections.

5 Sources and References

1. https://en.wikipedia.org/wiki/Error_detection_and_correction
2. https://www.geeksforgeeks.org/hamming-code-implementation-in-python/
3. https://www.tutorialspoint.com/digital_circuits/digital_circuits_error_detection_correction_codes.htm
4. https://www.javatpoint.com/computer-network-error-detection