COMPUTER NETWORKS

LAB REPORT

ASSIGNMENT 1

DEBJIT DHAR

BCSE UG 3

ROLL:002210501106

GROUP: A3

SUBMISSION: 09/09/2024

Problem Statement: Design and implement error detection techniques within a simulated network environment.

Sender Program: 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 dataword from the input. Based on the schemes, codewords will be prepared. Sender will send the codewords to the Receiver.

Error injection module: Inject error in random positions in the input data frame. Write a separate method for that. Sender program will randomly call this method before sending the codewords to the receiver.

Receiver Program: Receiver will check if there is any error detected. Based on the detection it will accept or reject the dataword.

- **Checksum (16-bit)**: Checksum of a block of data is the complement of the one's complement of the 16-bit sum of the block. The message (from the input file) is divided into 16-bit words. The value of the checksum word is set to 0. All words are added using 1's complement addition. The sum is complemented and becomes the checksum. So, if we transmit the block of data including the checksum field, the receiver should see a checksum of 0 if there are no bit errors.
- **CRC**: CRC generator polynomials will be given as input (CRC-8, CRC-10, CRC-16 and CRC-32). Show how good is the selected polynomial to detect single-bit error, two isolated single-bit errors, odd number of errors, and burst errors.

○ CRC-8: x 8 + x 7 + x 6 + x 4 + x 2 + 1 (Use: General, Bluetooth wireless communication)
○ CRC-10: x 10 + x 9 + x 5 + x 4 + x + 1 (Use: General, Telecommunication)
○ CRC-16: x 16 + x 15 + x 2 + 1 (Use: USB)
○ CRC-32: x 32 + x 26 + x 23 + x 22 + x 16 + x 12 + x 11 + x 10 + x 8 + x 7 + x 5 + x 4 + x 2 + x + 1 (Use: Ethernet IEEE802.3)

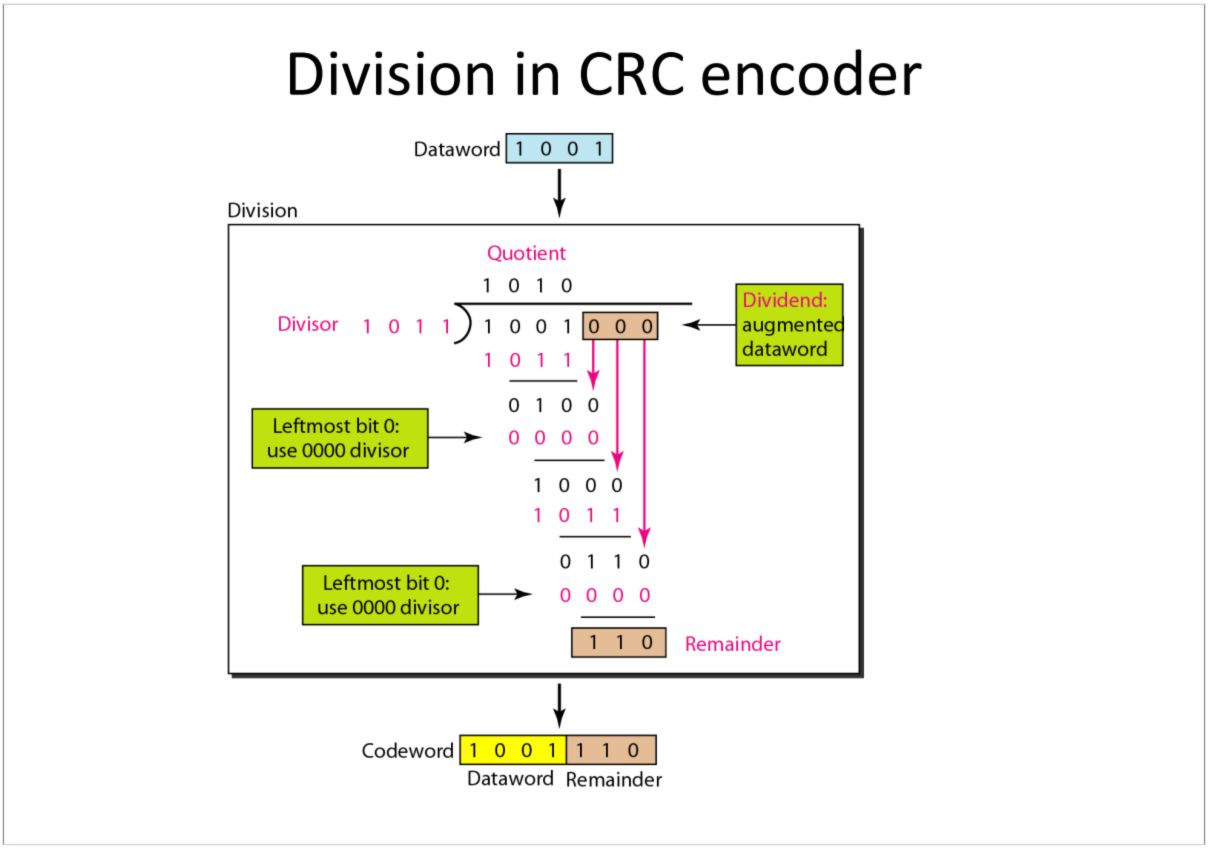
● **Error types**: single-bit error, two isolated single-bit errors, odd number of errors, and burst errors. Test the above two schemes for the error types and CRC polynomials mentioned above for the following cases (not limited to).
○ Error is detected by both CRC and Checksum.
○ Error is detected by checksum but not by CRC.
○ Error is detected by CRC but not by Checksum.

DESIGN

Theory:

CRC:

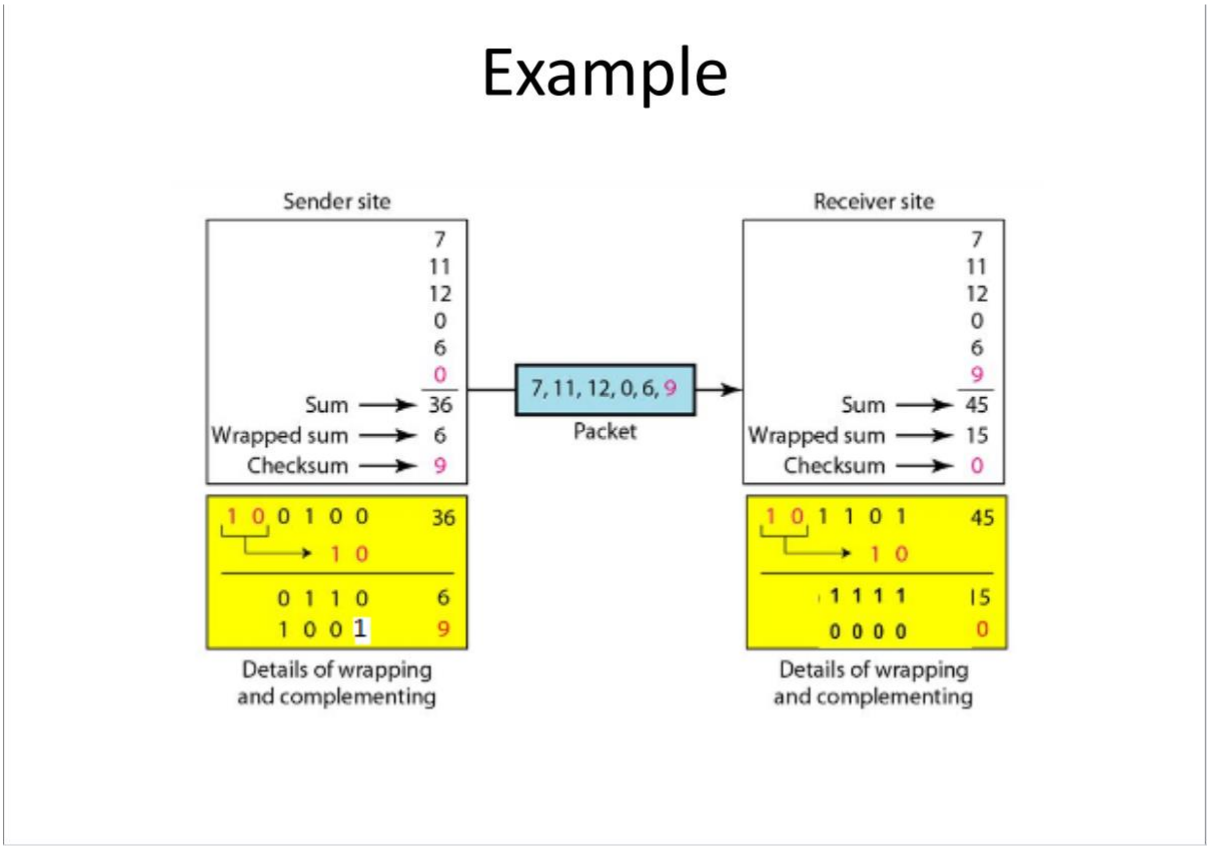
Cyclic Redundancy Check (CRC) is an error-detection scheme commonly used in digital networks and storage devices to detect accidental changes to raw data. When data is sent or stored, a CRC value (a specific number of bits) is computed and appended to the data. This value is derived from the data using a predetermined polynomial. When the data is received or retrieved, the CRC is recalculated and compared to the original CRC value. If the values match, the data is assumed to be error-free; if not, an error is detected. The simplicity and effectiveness of CRC make it widely used in ensuring data integrity in various systems.



Checksum:

A checksum is a value used to verify the integrity of data during transmission or storage. It is computed by performing a mathematical operation on a block of data, resulting in a small, fixed-size value—typically a sequence of numbers or characters. When data is transmitted or stored, the checksum is calculated and sent or stored along with the data. Upon retrieval or receipt, the checksum is recalculated and compared to the original checksum. If the two values match, the data is considered to be intact; if they don't, this indicates that the data may have been altered or corrupted during transmission or storage.

Checksums are widely used in various applications, including file transfers, network communications, and data storage systems, as a simple method to detect errors. Although checksums are effective at detecting many common types of errors, they are not foolproof. More robust error-detection methods, like Cyclic Redundancy Check (CRC) or cryptographic hashes, are sometimes used when higher levels of data integrity assurance are required.



Example

Sender Program:

This program is designed to simulate the process of error detection and correction in data communication. It reads binary data from a file and then simulates errors during transmission by modifying the data using different error types. The program supports two error detection schemes: Checksum and Cyclic Redundancy Check (CRC). Based on the user's choice, it computes the appropriate checksum or CRC remainder, appends it to the data, and then sends the modified data (with simulated errors) over a network socket to a receiver. This demonstrates how error detection methods help identify and potentially correct errors that occur during data transmission.

Input Format:

- **Filename:** The path to the file containing the binary string data.
- **Choice:** The user's choice between Checksum (1) or CRC (2) as the error detection method.
- **Clip Size (Checksum Only):** If the user selects Checksum, they are prompted to enter a clip size (bit-length for segmentation).
- **Divisor (CRC Only):** If the user selects CRC, they are prompted to enter a binary string divisor for the CRC calculation.
- **Host and Port:** The IP address and port number for the network communication.

Output Format:

- **Transmitted Data:** The binary string data with the simulated error, alongside the computed checksum or CRC remainder.
- **Serialized Data:** The program serializes the transmitted data, the error detection value (checksum or CRC), and other relevant information for network transmission.

Flowchart:

Receiver Program Overview

This program is meticulously crafted to emulate the process of error detection within data communication systems. It achieves this by receiving binary data over a network and verifying its integrity through two well-established error detection techniques: **Checksum** and **Cyclic Redundancy Check (CRC)**. By listening for incoming data, deserializing it, and performing rigorous checks, the program aims to identify and report any transmission errors.

How the Receiver Program Operates

1. **Network Communication:**
   - The receiver program initializes by opening a network socket, setting it up to listen attentively for incoming connections on a designated IP address and port.
   - Upon establishing a connection, the program diligently receives serialized data transmitted by the sender.
2. **Data Deserialization:**
   - The received data undergoes a process of deserialization, transforming it back into its original components. This includes the binary string, the error detection value (either checksum or CRC remainder), the chosen error detection method, and any other pertinent information.
3. **Error Detection:**
   - Depending on the sender's selection of error detection method (Checksum or CRC), the receiver program computes the corresponding checksum or CRC remainder for the received binary string.
   - The program then compares this freshly computed value with the value received. A match signifies that the data was transmitted flawlessly, while any discrepancy indicates the presence of transmission errors.
4. **Result Output:**
   - After completing the error detection process, the program promptly outputs a message stating whether the received data is **error-free** or if it **contains errors**.

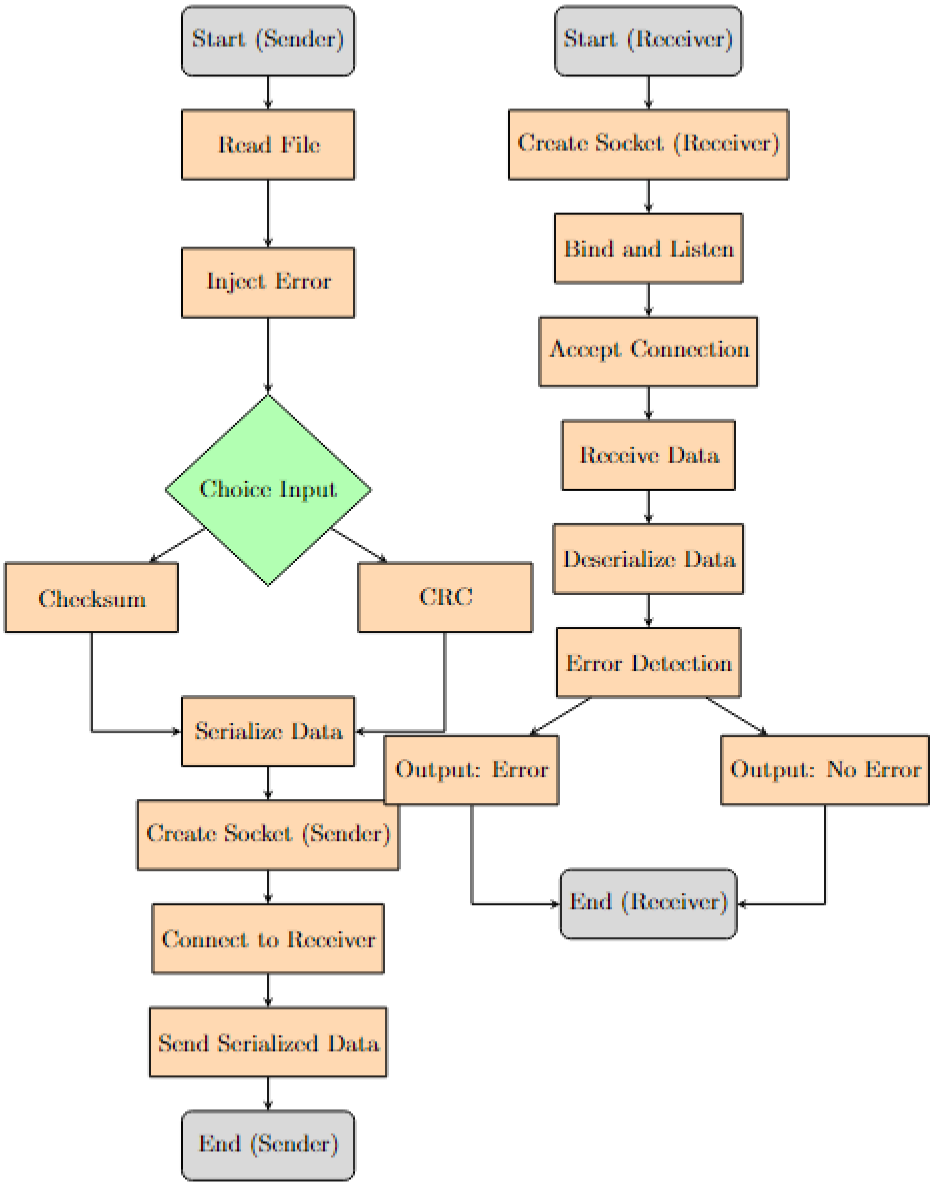
Input Format:

- **Host and Port:** Specify the IP address and port number on which the receiver will actively listen for incoming data.

Output Format:

- **Error Status:** The program delivers a concise message that reveals the integrity of the data. It will either confirm **"No Error"** if the data is intact or announce **"Error"** if any discrepancies are detected.

Structural Diagram



Implementation

Implementable code at: https://github.com/Debjit-Dhar/Networks

Sender Program

```
def read_file(file):
    with open(file,'r') as f:
        s=f.read()
    return s
```

Purpose: This function reads a binary string from a text file.

Parameters:

- file: The path to the file containing the binary data.

Process:

- Opens the file in read mode.
- Reads the content of the file.
- Returns the content as a string.

```
def checksum(s,clip):
    l=[]
    s='0'*(len(s)%clip)+s
    for i in range(0,len(s),clip):
        l.append(s[i:i+clip])
    #print(l)
    decimal=list(map(lambda x:int(x,2),l))
    s_dec=sum(decimal)
    #print(s_dec)
    bin_s_dec=str(bin(s_dec)[2:])

    clip_val=int(bin_s_dec[len(bin_s_dec)-clip:],2)
    #print(clip_val)
    clip_val_before=int(bin_s_dec[:len(bin_s_dec)-clip],2)

    clip_sum=clip_val+clip_val_before
    clip_sum_bin=str(bin(clip_sum)[2:])
    if(clip-len(clip_sum_bin)>0):
        clip_sum_bin='0'*(clip-len(clip_sum_bin))+clip_sum_bin
    clip_sum_bin_complement = int(''.join('1' if bit == '0' else '0' for
bit in clip_sum_bin),2)
    decimal.append(clip_sum_bin_complement)
    return decimal
```

Purpose: This function calculates a checksum for error detection.

Parameters:

- s: The binary string.
- clip: The bit-length for segmentation of the binary string.

Process:

- **Padding:** Adds zeros at the beginning of the string if its length isn't a multiple of clip.
- **Segmentation:** Divides the string into segments of size clip.
- **Decimal Conversion:** Converts each segment to its decimal equivalent.
- **Sum Calculation:** Sums all the decimal values.
- **Binary Conversion:** Converts the sum back to binary.
- **Handling Overflow:** Separates the least significant bits (size of clip) and the more significant bits. These are added together to handle overflow.
- **Complement:** The complement of the final sum is taken to form the checksum.
- **Return:** The checksum and all decimal segments are returned.

```python
def crc(s, divisor):
    orig_s=s
    s += '0' * (len(divisor) - 1)
    s = list(s)
    orig_divisor=divisor
    divisor = list(divisor)
    for i in range(len(s) - len(divisor) + 1):
        if s[i] == '1':  # Only perform XOR if the bit is 1
            for j in range(len(divisor)):
                s[i + j] = str(int(s[i + j]) ^ int(divisor[j]))
    remainder = ''.join(s[-(len(divisor) - 1):])

    return remainder
```

Purpose: This function computes the Cyclic Redundancy Check (CRC) for error detection.

Parameters:

- s: The binary string to be transmitted.
- divisor: The binary divisor used for the CRC calculation.

Process:

- **Appending Zeros:** Adds zeros equal to the length of the divisor minus one to the end of the string.
- **Division:** Performs binary division (XOR operation) between the data and the divisor, simulating polynomial division.
- **Remainder:** The remainder from this division is the CRC code, which is appended to the data for transmission.
- **Return:** The remainder is returned as the CRC code

0=>Single Bit, 1=>Two Isolated Single Bit, 2=>Odd number of errors, 3=> Burst Errors

```python
import numpy as np
import time
def error_inject(s, etype=0):
    np.random.seed(int(time.time()))
    etype=np.random.randint(0, 8)
    print(etype)
    esize=np.random.randint(0,len(s))
    s = list(s)

    if etype == 0:
        np.random.seed(int(time.time()))
        ebit = np.random.randint(0, len(s))
        s[ebit] = '1' if s[ebit] == '0' else '0'

    elif etype == 1:
        np.random.seed(int(time.time()))
        ebit1 = np.random.randint(0, len(s))
        ebit2 = np.random.randint(0, len(s))
        while ebit1 == ebit2 or abs(ebit1 - ebit2) == 1:
            ebit1 = np.random.randint(0, len(s))
            ebit2 = np.random.randint(0, len(s))
        s[ebit1] = '1' if s[ebit1] == '0' else '0'
        s[ebit2] = '1' if s[ebit2] == '0' else '0'

    elif etype == 2:
        np.random.seed(int(time.time()))
        mid = len(s) // 2
        n_error = np.random.randint(0, mid) * 2 + 1
        random_list = np.random.choice(len(s), n_error, replace=False)
        for idx in random_list:
            s[idx] = '1' if s[idx] == '0' else '0'

    elif etype == 3:
        np.random.seed(int(time.time()))
        random_list = np.random.choice(len(s), esize, replace=False)
        print(random_list)
        for idx in random_list:
            s[idx] = '1' if s[idx] == '0' else '0'
    else:
        pass
    #s[3]='0'
    #s[19]='1'
    return ''.join(s)
```

Purpose: This function simulates the introduction of errors into the binary string to test the error detection schemes.

Parameters:

- s: The binary string.
- etype: The type of error to introduce (0-4).

Process:

- Different types of errors are simulated based on etype:
- **Type 0:** Single-bit error.
- **Type 1:** Two non-adjacent bit errors.
- **Type 2:** An odd number of random bit errors.
- **Type 3:** Random bit errors up to a random size.
- **Return:** The modified binary string with errors is returned.

First get s from text file, then call error_inject then call either crc or checksum then send the codeword along with the divisor or checksum as applicable

```
import socket
import time
import pickle

def send_binary_string(filename, choice,host, port):
    s=read_file(filename)
    s_error=error_inject(s,1)
    #augword=checksum or remainder(in case of crc)
    if choice==1:#checksum
        clip=int(input('Enter clip size'))
        augword=checksum(s,clip)
        serialized_data=pickle.dumps((s_error,clip,augword,choice))
    else:
        divisor=input('Enter divisor')
        augword=crc(s,divisor)
        serialized_data = pickle.dumps((s_error,divisor,augword,choice))
    # Create a socket object
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        # Connect to the receiver
        s.connect((host, port))

        # Send the serialized data
        s.sendall(serialized_data)
    ''' # Create a UDP socket object(For UDP Socket)
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as udp_socket:
        # Send the serialized data
        udp_socket.sendto(serialized_data, (host, port))'''
```

Purpose: This function coordinates reading the file, injecting errors, calculating the checksum or CRC, and then sending the data over a network.

Parameters:

- filename: The path to the binary data file.
- choice: The user's choice of error detection method (1 for Checksum, 2 for CRC).
- host: The IP address of the receiver.
- port: The port number on the receiver's end.

Process:

- **File Read:** Reads the binary string from the file.
- **Error Injection:** Injects errors into the binary string.
- **Checksum or CRC:** Based on user choice, calculates either the checksum or CRC.
- **Serialization:** Packs the modified data, along with the checksum/CRC, into a serialized format using pickle.
- **Sending Data:** Opens a TCP socket, connects to the specified host and port, and sends the serialized data.

```
if __name__ == '__main__':
    # Change these to your server's address and port
    HOST = 'localhost'
    PORT = 65432
    filename='C:/Users/sudip/PycharmProjects/pythonProject1/My Python
Programs/DSA and OOPs and Machine Learning/CN Assignments/bin_data.txt'
    choice=int(input('Enter choice'))
    send_binary_string(filename,choice,HOST,PORT)
```

Receiver

```
def is_error_checksum(s, clip, checksum):  # True if error False if no
error
    l = []
    s = '0' * (len(s) % clip) + s
    for i in range(0, len(s), clip):
        l.append(s[i:i+clip])
    decimal = list(map(lambda x: int(x, 2), l))
    s_dec = sum(decimal)

    s_dec += checksum
    print(s_dec)
    bin_s_dec = str(bin(s_dec)[2:])

    clip_val = int(bin_s_dec[len(bin_s_dec)-clip:], 2)
    clip_val_before = int(bin_s_dec[:len(bin_s_dec)-clip], 2)

    clip_sum = clip_val + clip_val_before
    clip_sum_bin = str(bin(clip_sum)[2:])
    return not all(char == '1' for char in clip_sum_bin)
```

Purpose:

- This function checks for errors in a binary string s using a checksum method. It returns True if an error is detected and False if no error is found.

Parameters:

- s: The binary string to be checked.
- clip: The length of segments into which the binary string s is divided.
- checksum: The expected checksum value used to verify the integrity of the binary string.

Process:

- The function first pads the string s with leading zeros if the length of s is not a multiple of clip.

- The string is then divided into segments of length clip, and each segment is converted from binary to decimal.

- The sum of all these decimal values is calculated and added to the checksum.

- The binary representation of this sum (bin_s_dec) is split into two parts:

- clip_val: The last clip bits of the sum.

- clip_val_before: The remaining bits before the last clip bits.

- These two parts are summed together, and the binary representation of this sum (clip_sum_bin) is checked to see if all bits are '1'. If all bits are '1', the function returns False (no error); otherwise, it returns True (error).

```python
def is_error_crc(s, divisor, augword):  # True if error False if no error
    s += augword
    s = list(s)
    divisor = list(divisor)
    for i in range(len(s) - len(divisor) + 1):
        if s[i] == '1':  # Only perform XOR if the bit is 1
            for j in range(len(divisor)):
                s[i + j] = str(int(s[i + j]) ^ int(divisor[j]))
    remainder = ''.join(s[-(len(divisor) - 1):])
    r = int(remainder, 2)
    return r != 0
```

Purpose:

- This function checks for errors in a binary string s using the Cyclic Redundancy Check (CRC) method. It returns True if an error is detected and False if no error is found.

Parameters:

- s: The binary string to be checked.

- divisor: The binary divisor used in the CRC check.

- augword: The augmentation word, usually a string of zeros added to s before performing the division.

Process:

- The function appends augword to the end of the binary string s, effectively extending the length of s by the number of bits in augword.

- It then performs binary division of this extended string by the divisor. The division is done by XORing the segments of s with the divisor whenever the leading bit of the segment is '1'.

- The remainder of this division is calculated (the last few bits of the modified s).
- If the remainder is not zero, it indicates an error in the binary string, and the function returns True. Otherwise, it returns False.

```
import socket
import pickle

def receive_binary_string(host, port):
    # Create a socket object
    ''' # Create a UDP socket object
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as udp_socket:
        # Bind the socket to the address and port
        udp_socket.bind((host, port))

        print(f'Listening on {host}:{port}...')

        # Receive the serialized data (UDP is connectionless, so no
accept/listen)
        data, addr = udp_socket.recvfrom(4096)'''
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        # Bind the socket to the address and port
        s.bind((host, port))
        # Listen for incoming connections
        s.listen()

        # Accept a connection
        conn, addr = s.accept()
        with conn:
            print(f'Connected by {addr}')
            # Receive the serialized data
            data = b''
            while True:
                packet = conn.recv(4096)
                if not packet:
                    break
                data += packet

            # Deserialize the tuple
            s_error, other, augword, choice = pickle.loads(data)  # other
is the checksum or divisor as the case may be
            if choice == 1:  # Checksum
                is_error = is_error_checksum(s_error, other, int(augword[-
1]))
            else:  # CRC
                is_error = is_error_crc(s_error, other, augword)
            print(f'Received tuple of strings: {s_error, other, augword,
choice}')
    return is_error
```

Purpose:

- This function acts as a server that receives a serialized tuple over a network connection, deserializes it, and checks the binary string for errors using either the checksum or CRC method, based on the received data.

Parameters:

- host: The IP address or hostname where the server will listen for connections.
- port: The port number on which the server will listen for incoming connections.

Process:

- The function creates a TCP socket and binds it to the specified host and port.
- It listens for incoming connections and accepts one connection.
- Once connected, it receives data in chunks of 4096 bytes until all data is received.
- The data is then deserialized using the pickle module. The received tuple contains four elements: s_error, other, augword, and choice.
- s_error: The binary string to be checked for errors.
- other: The checksum or divisor, depending on the method chosen.
- augword: The augmentation word (for CRC) or a string whose last character is the checksum (for checksum).
- choice: An integer indicating which error-checking method to use (1 for checksum, anything else for CRC).
- Depending on the value of choice, the function calls either is_error_checksum or is_error_crc to check for errors in the binary string.
- The result of the error check (True or False) is then returned.

```
if __name__ == '__main__':
    # Change these to your server's address and port
```

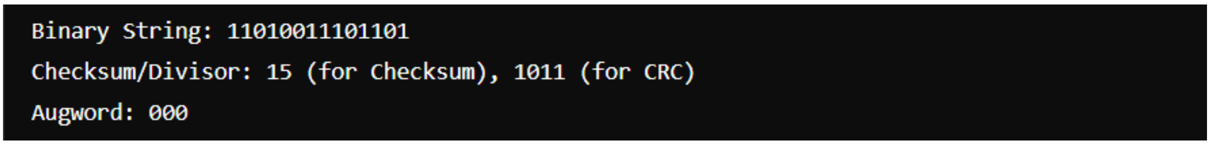
```
HOST = 'localhost'
PORT = 65432

if receive_binary_string(HOST, PORT):
    print('Error')
else:
    print('No Error')
```

TEST CASES

Case 1: Error Detected by Both CRC and Checksum

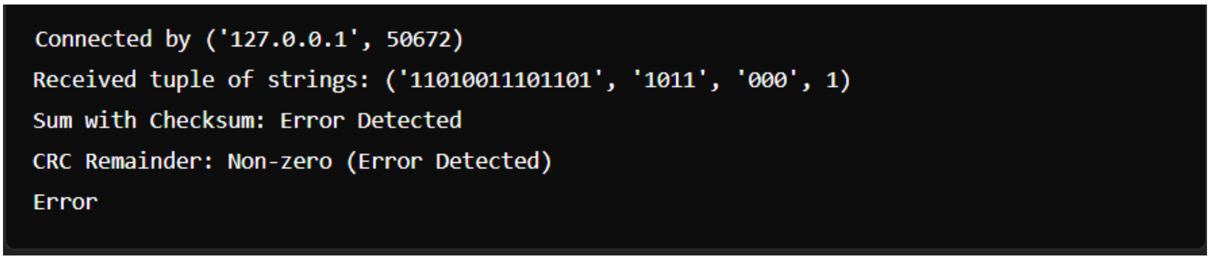
- Binary String (s_error): "11010011101100"
- Checksum (other): 15
- CRC Divisor (other): "1011"
- Augword: "000"
- Error Introduced: "11010011101101" (last bit flipped)

Sender Output:



```
Binary String: 11010011101101
Checksum/Divisor: 15 (for Checksum), 1011 (for CRC)
Augword: 000
```

Receiver Output:



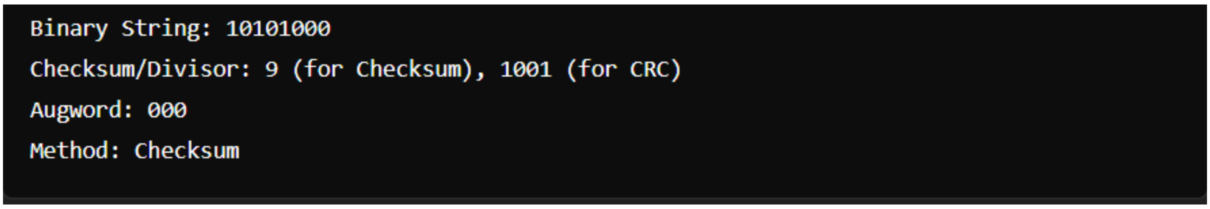
```
Connected by ('127.0.0.1', 50672)
Received tuple of strings: ('11010011101101', '1011', '000', 1)
Sum with Checksum: Error Detected
CRC Remainder: Non-zero (Error Detected)
Error
```

Case 2: Error Detected by Checksum but Not by CRC

- Binary String (s_error): "10101010"
- Checksum (other): 9
- CRC Divisor (other): "1001"
- Augword: "000"

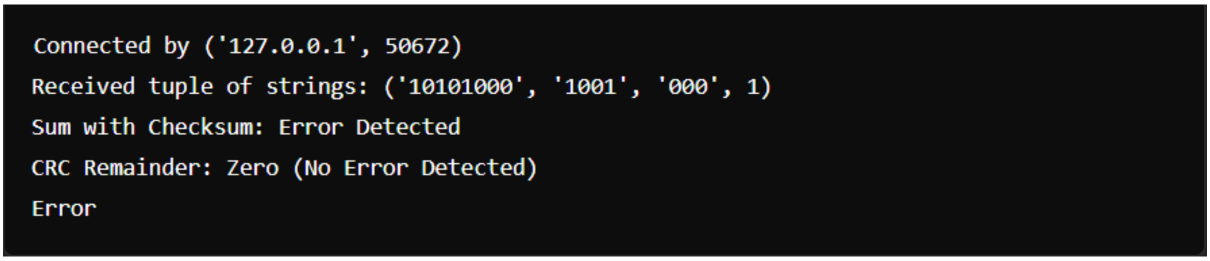
- Error Introduced: "10101000" (one bit flipped)

Sender Output



```
Binary String: 10101000
Checksum/Divisor: 9 (for Checksum), 1001 (for CRC)
Augword: 000
Method: Checksum
```

Receiver Output:

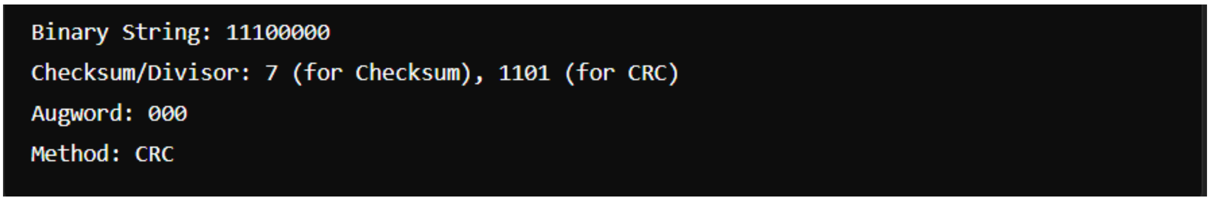


```
Connected by ('127.0.0.1', 50672)
Received tuple of strings: ('10101000', '1001', '000', 1)
Sum with Checksum: Error Detected
CRC Remainder: Zero (No Error Detected)
Error
```

Case 3: Error Detected by CRC but Not by Checksum

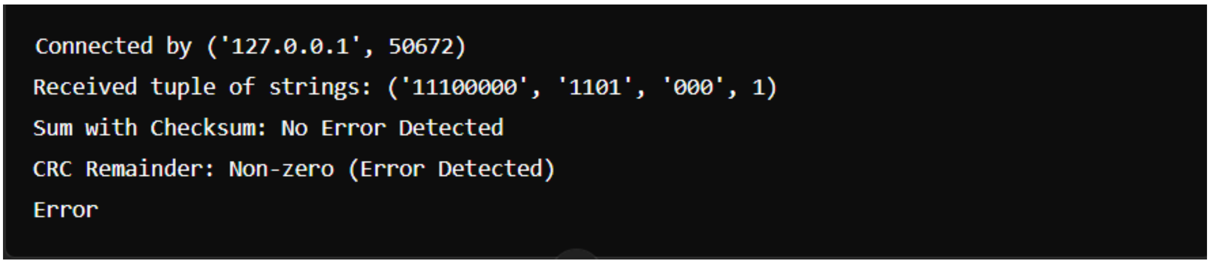
- Binary String (s_error): "11100100"
- Checksum (other): 7
- CRC Divisor (other): "1101"
- Augword: "000"
- Error Introduced: "11100000" (two bits flipped to cancel out in checksum)

Sender:



```
Binary String: 11100000
Checksum/Divisor: 7 (for Checksum), 1101 (for CRC)
Augword: 000
Method: CRC
```

Receiver:



```
Connected by ('127.0.0.1', 50672)
Received tuple of strings: ('11100000', '1101', '000', 1)
Sum with Checksum: No Error Detected
CRC Remainder: Non-zero (Error Detected)
Error
```

RESULTS

Error is injected 50% times.

Each type of error is injected 25% times.

ANALYSIS

a. When all four schemes are employed, the ability to detect errors improves notably. However, around 3.2% of the time, none of the schemes manage to identify the error.

b. The errors introduced by the system are sporadic and unpredictable, leading to potential experimental inconsistencies. In some instances, a bit may flip an even number of times, effectively canceling out any error, meaning the inject_error() function might not always introduce detectable errors. These situations are disregarded, with the program assuming that errors are being successfully injected into all codewords.

c. Due to the use of sockets for data transfer, incomplete socket buffer transmissions can result in unexpected outcomes. To mitigate this, the program pauses for 1 second before sending each data element, though this delay slows down the overall process.

COMMENTS

This assignment has greatly enhanced my understanding of different error detection techniques through both theoretical study and hands-on application. By delving into these methods, I have gained a deeper appreciation for their benefits and drawbacks. Furthermore, I have learned how the limitations of one approach can be mitigated by leveraging alternative strategies.

I would like to express my heartfelt thanks to my teachers Dr Sarbani Roy and Dr Nandini Mukherjee for their guidance and support throughout this journey. Their encouragement has played a key role in helping me better comprehend these concepts.