COMPUTER NETWORKS LAB

REPORT ASSIGNMENT-1

NAME: SOHAM LAHIRI

CLASS: BCSE UG-III 5TH SEMESTER

ROLL NO: 002210501107

GROUP: A3

SUBMISSION DATE: 02/09/2024

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.

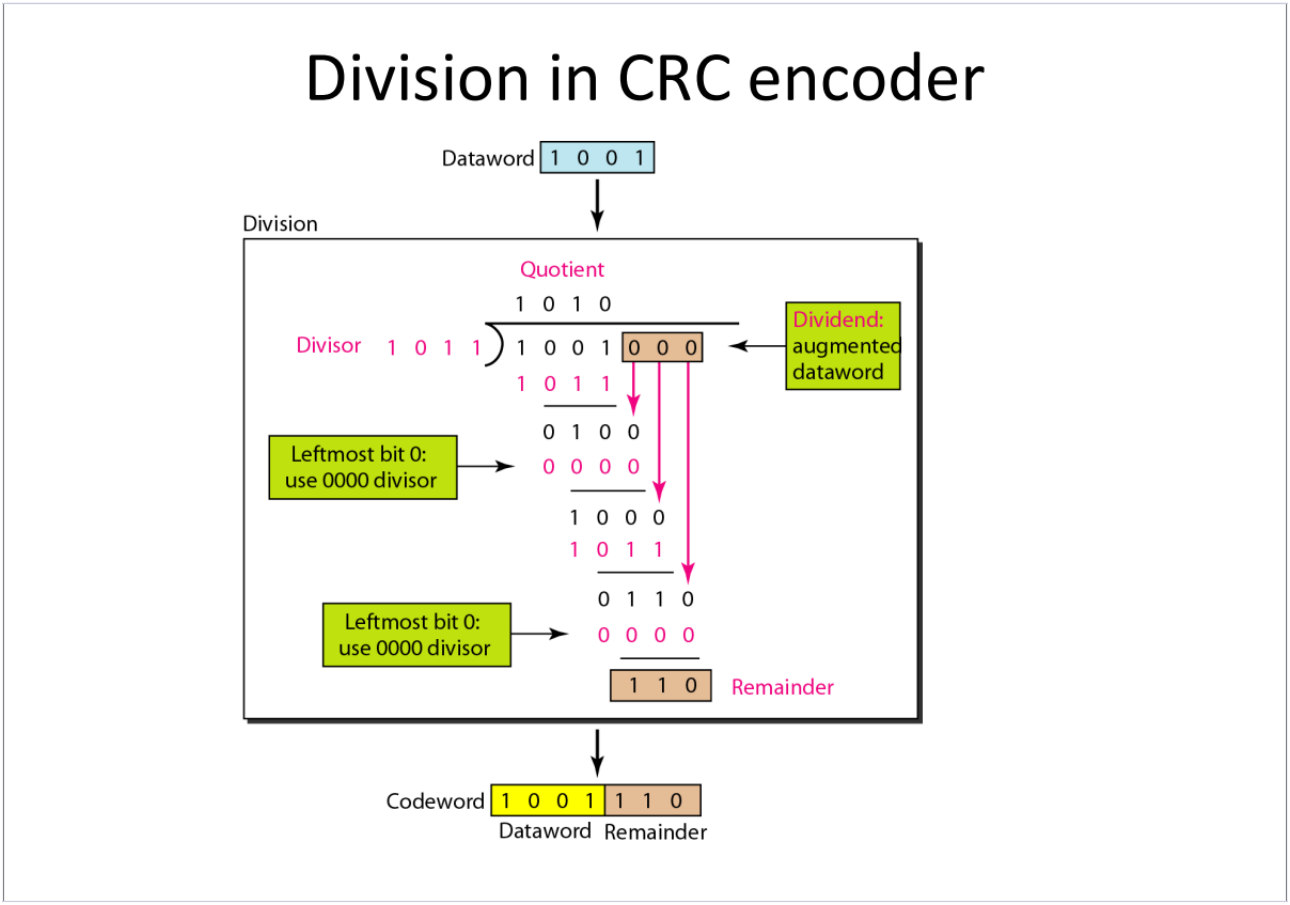
DESIGN

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.

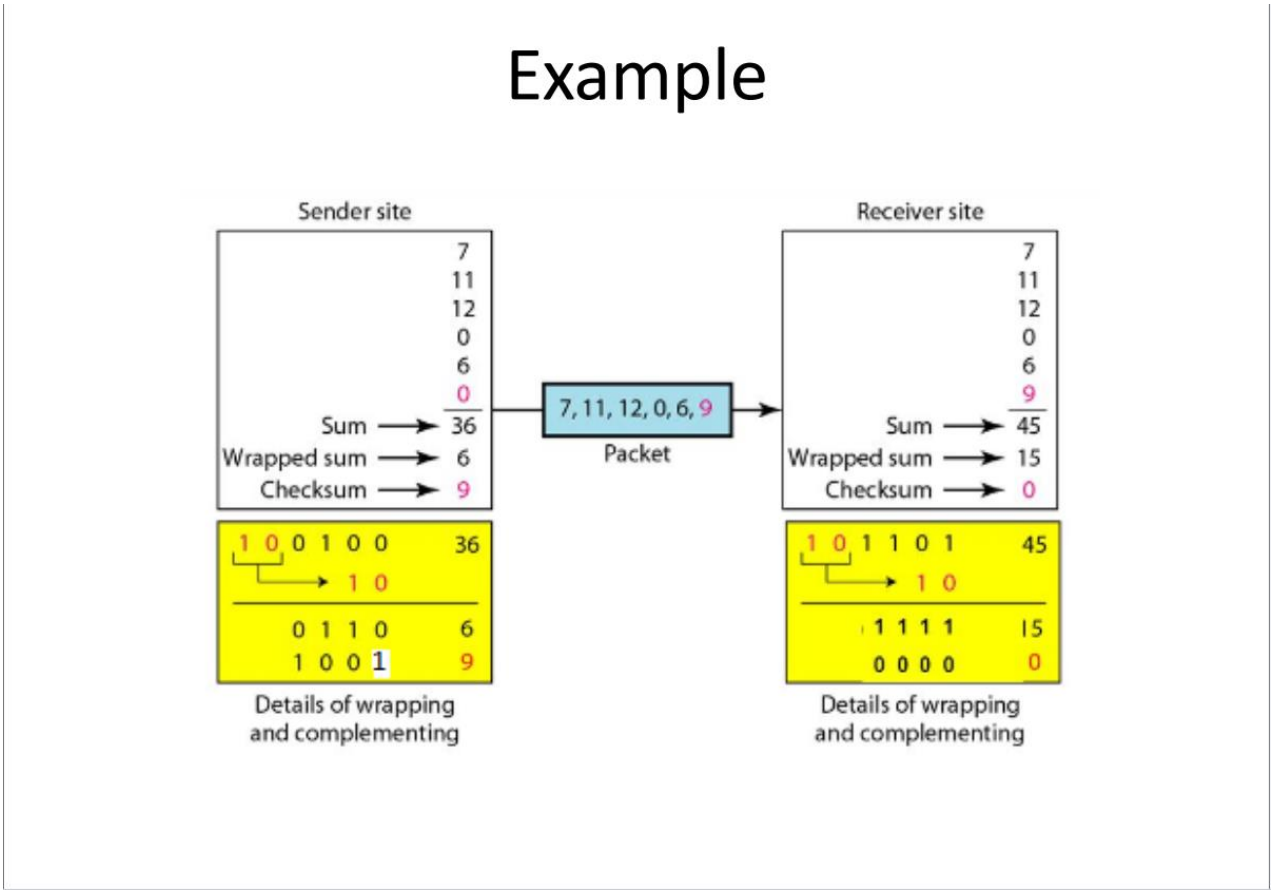
Example:



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.

Receiver Program:

This program is designed to simulate the process of error detection in data communication by receiving binary data over a network and verifying its integrity using two error detection schemes: Checksum and Cyclic Redundancy Check (CRC). The receiver program listens for incoming data, deserializes it, and then checks for transmission errors based on the selected error detection method. The goal is to identify whether the data received contains any errors and print the corresponding result.

How It Works:

1. **Network Communication**:
    - The program opens a network socket and listens for incoming connections on a specified IP address and port.
    - Once a connection is established, the program receives the serialized data sent by the sender program.
2. **Data Deserialization**:
    - The program deserializes the received data into its original components, including the binary string data, the error detection value (checksum or CRC remainder), the chosen error detection method, and any other relevant information.
3. **Error Detection**:
    - Based on the user's choice from the sender program (Checksum or CRC), the receiver computes the checksum or CRC remainder on the received binary string.
    - The program then compares the computed value with the received value to determine if there was an error during transmission.
    - If the computed value matches the received value, it indicates that the data was transmitted without errors. Otherwise, an error is detected.
4. **Result Output**:
    - The program outputs whether the received data is error-free or contains errors.

Input Format:

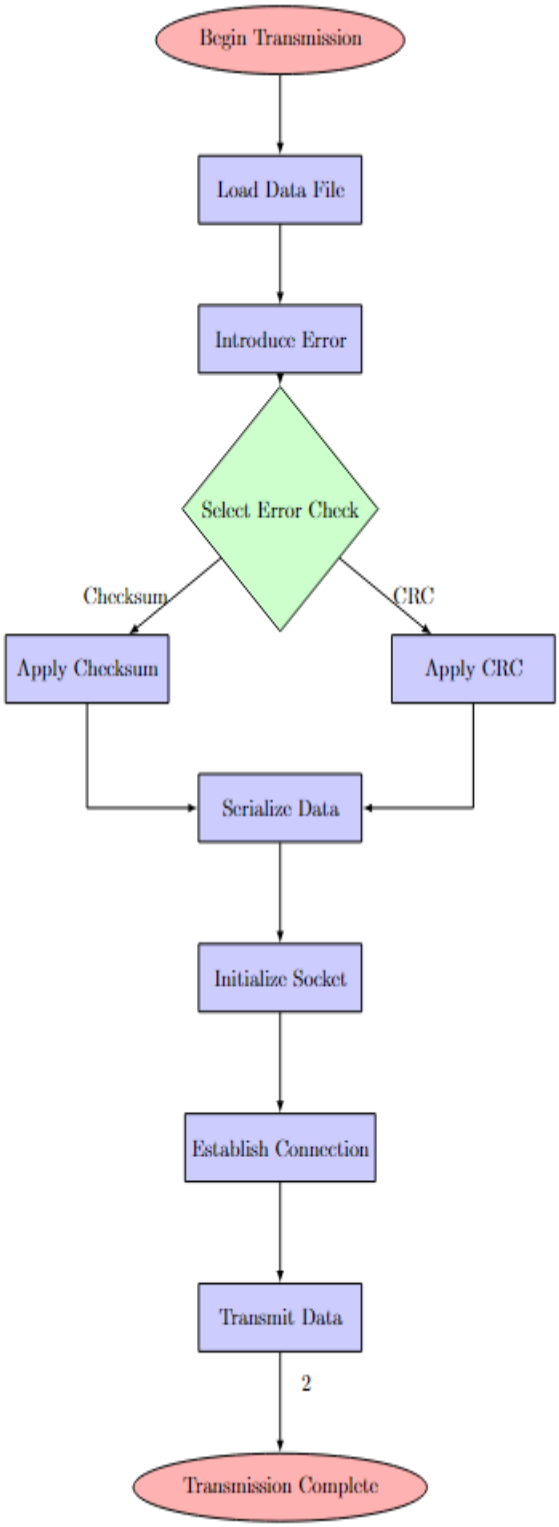
- **Host and Port**: The IP address and port number where the receiver listens for incoming data.

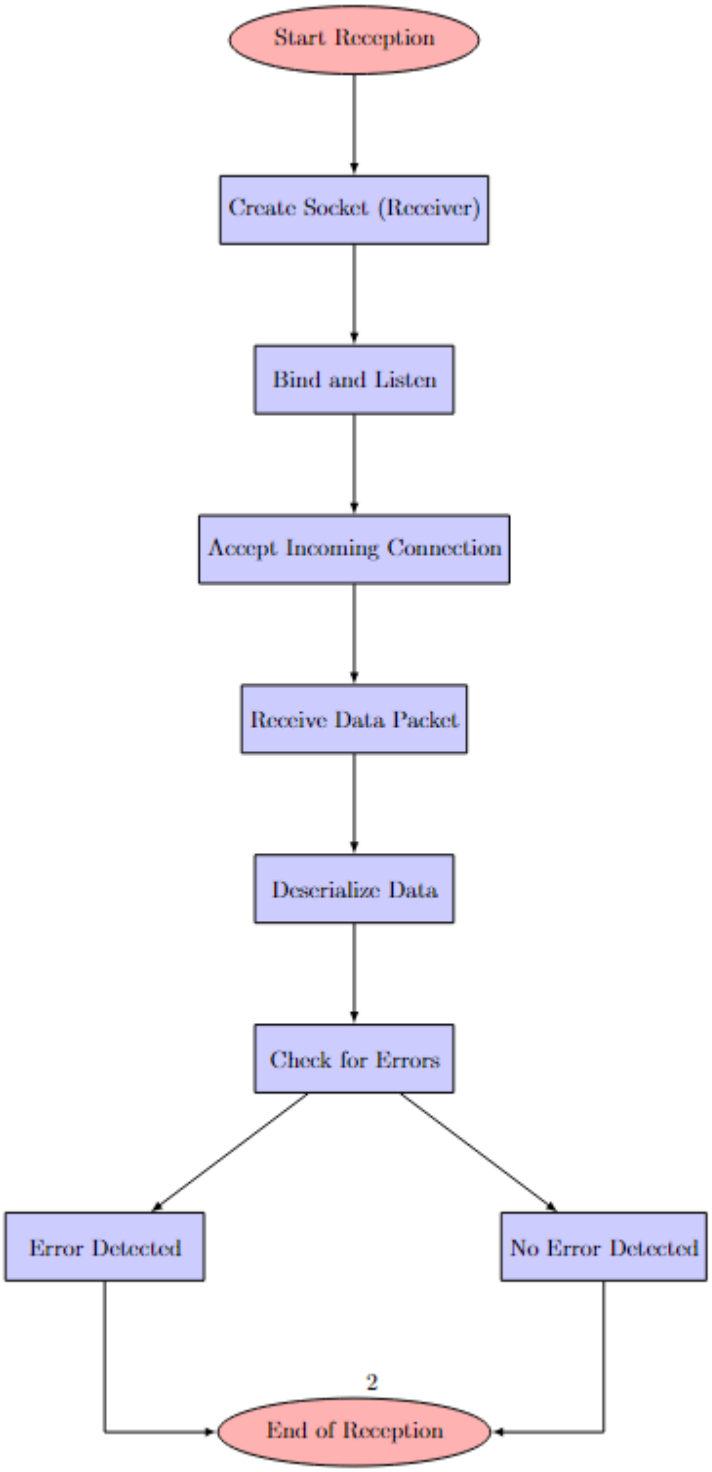
Output Format:

- **Error Status**: A message indicating whether the received data contains an error ("Error") or is error-free ("No Error").

This receiver program complements the sender program by verifying the integrity of the data transmitted over the network. It provides a practical demonstration of how error detection techniques like Checksum and CRC are used to ensure reliable communication in data transmission systems.

STRUCTURAL DIAGRAM




Start Reception
Create Socket (Receiver)
Bind and Listen
Accept Incoming Connection
Receive Data Packet
Deserialize Data
Check for Errors
Error Detected
No Error Detected
End of Reception

Implementation:

Sender Program:

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

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

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
```

Below convention is followed in error injection function:
0: Single Bit
1: Two Isolated Single Bit
2: Odd number of errors
3: Burst Errors

```
import numpy as np
import time
def error_inject(s, etype=4):
    #np.random.seed(int(time.time()))
    #etype=np.random.randint(0, 8)
    etype=4
    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
    #checksum no, crc yes
    #s[3]='0'
    #s[19]='1'

    #crc no,checksum yes
    s[15]='0'
    return ''.join(s)

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)

if __name__ == '__main__':
    # Change these to your server's address and port
    HOST = 'localhost'
    PORT = 12345
    filename='C:/Users/User/Desktop/CSE NOTES/CSE NOTES-3RD YEAR 5TH
SEM/Computer Network Lab/Assignment-1/bin_data (1).txt'
    choice=int(input('Enter choice'))
    send_binary_string(filename,choice,HOST,PORT)
```

Explanation:

1. read_file(file)

Purpose:

- Reads the content of a file and returns it as a string.

How it works:

- It opens the specified file in read mode ('r'), reads its entire content into a string s, and returns this string.

2. checksum(s, clip)

Purpose:

- Computes a checksum value for the given binary string s with a specified clip size (bit length).

How it works:

- Pads the binary string s with leading zeros to make its length a multiple of clip.
- Splits the padded string into segments of length clip.
- Converts each segment into a decimal number.
- Sums all these decimal numbers and converts the sum back to binary.
- Splits this binary sum into two parts: clip_val (the last clip bits) and clip_val_before (the remaining bits).
- Adds these two parts together, and calculates the binary complement of the sum to get the checksum.
- The checksum is then appended to the list of decimal values, which is returned.

3. crc(s, divisor)

Purpose:

- Computes the Cyclic Redundancy Check (CRC) for the given binary string s using the provided divisor.

How it works:

- Appends enough zeros to s to match the length of the divisor minus one (to simulate CRC calculation).
- Performs bitwise XOR operations between the binary string and the divisor wherever the current bit of s is '1'.
- The remainder from this division is calculated and returned as the CRC code.

4. error_inject(s, etype=4)

Purpose:

- Injects errors into the binary string s based on the specified error type (etype).

How it works:

- Converts the string s into a list to allow modifications.
- Depending on etype, different types of errors are introduced:
  - **0**: Single bit error is introduced at a random position.
  - **1**: Two isolated single-bit errors are introduced at different random positions.
  - **2**: An odd number of errors are introduced at random positions.
  - **3**: A burst error of random length is introduced at random positions.
- The modified string with injected errors is returned as a string.

5. send_binary_string(filename, choice, host, port)

Purpose:

- Sends the binary string (with potential errors) and its computed checksum or CRC to a receiver over a network socket.

How it works:

- Reads the binary string from a file using read_file.
- Injects errors into the string using error_inject.
- Depending on the user's choice (choice), it either calculates a checksum or CRC:
  - **choice 1**: Calculates a checksum using the specified clip size.
  - **choice 2**: Calculates a CRC using the specified divisor.
- The binary string, along with the computed checksum/CRC and other necessary data, is serialized using pickle.
- Creates a socket, connects to the specified host and port, and sends the serialized data.

Main Execution Block:

- If the script is run directly, it prompts the user for a choice (checksum or CRC) and calls send_binary_string with the chosen parameters, including the filename, choice, host, and port for the connection

Implementation:

Receiver Program:

```
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)

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

import socket
import pickle
def receive_binary_string(host, port):
    # Create a socket object
    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

if __name__ == '__main__':
    # Change these to your server's address and port
    HOST = 'localhost'
    PORT = 12345

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

Explanation:

1. is_error_checksum(s, clip, checksum)

Purpose:

- Determines if there is an error in the received data when using a checksum.

How it works:

- Pads the binary string s with leading zeros to make its length a multiple of clip.
- Splits the padded string into segments of length clip and converts each segment into a decimal number.
- Sums these decimal values along with the received checksum.
- Converts the sum to a binary string.
- Extracts the clip_val (last clip bits) and clip_val_before (remaining bits).
- Adds these two parts together and checks if the resulting binary string (clip_sum_bin) consists entirely of '1's.
- Returns True if there is an error (i.e., if clip_sum_bin is not all '1's), otherwise returns False.

2. is_error_crc(s, divisor, augword)

Purpose:

- Determines if there is an error in the received data when using CRC (Cyclic Redundancy Check).

How it works:

- Appends the received augword (CRC remainder) to the binary string s.
- Performs bitwise XOR operations between the modified binary string and the divisor wherever the current bit of s is '1'.
- Calculates the remainder after this division and converts it to an integer.
- Returns True if the remainder is not zero (indicating an error), otherwise returns False.

3. receive_binary_string(host, port)

Purpose:

- Receives the binary string along with the checksum or CRC from the sender over a network socket and checks for errors.

How it works:

- Creates a socket and binds it to the specified host and port.
- Listens for incoming connections and accepts one when available.
- Receives the serialized data in chunks (using recv in a loop until all data is received).
- Deserializes the received data using pickle, unpacking it into the variables s_error, other, augword, and choice.
    - s_error: The binary string that may contain errors.

    - other: The checksum or divisor, depending on the chosen error-checking method.
    - augword: The checksum or CRC remainder that was sent along with the data.
    - choice: Indicates whether to use checksum (1) or CRC (2) for error checking.
- Depending on choice, it calls either is_error_checksum or is_error_crc to determine if an error is present.
- Prints the result and returns True if an error is detected, otherwise returns False.

Main Execution Block:

- When the script is run directly, it calls receive_binary_string to start the server and receive data.
- Based on the result from receive_binary_string, it prints "Error" if an error is detected, otherwise prints "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 individual error; Single Bit, Two Isolated Single Bit, Odd number of errors, Burst Errors, occur 25% times each.

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 significantly deepened my understanding of various error detection schemes through both research and practical implementation. By exploring and applying these techniques, I gained valuable insights into their strengths and limitations. Moreover, I learned how the shortcomings of one method can be effectively addressed by alternative approaches.

I would like to extend my sincere gratitude to our teacher for guiding us through this learning process. Their support and encouragement have been instrumental in helping me grasp these concepts more thoroughly.