

Computer Networks						
Assignment 1						
		BCSE III				
	Name: Mourya Saha					
	Roll: 002310501036					
	Group: A2					

Problem Statement:

Design and implement an error detection module which has two schemes namely Checksum and Cyclic Redundancy Check (CRC).

Design:

Purpose of the Program

This project implements and evaluates two common error detection techniques used in computer networks: Checksum and CRC. The primary goal is to understand their mechanisms, and to compare their effectiveness in detecting different types of data corruption that can occur during transmission.

The program simulates a client-server communication. The sender reads data from a file, applies the chosen error detection algorithm (Checksum or CRC) to generate a codeword, and transmits it to the receiver. The sender can also be configured to inject errors into the data before sending, to test the receiver's detection capabilities. The receiver, upon receiving the codeword, uses the same algorithm to check for errors and reports whether the data is corrupt or not.

Structure Diagram

The project is structured into several Python modules, each with a specific responsibility:

- **sender.py**: The client application. It reads a file, divides the data into chunks, computes the error detection code (**Checksum or CRC**), and sends the resulting codeword to the receiver over a TCP socket. It can also inject errors into the codeword with a certain probability.
- **reciever.py**: The server application. It listens for incoming TCP connections, receives the codewords from the sender, and uses the appropriate verification function (**verify_checksum or verify_crc**) to check for errors. It keeps a count of correctly identified transmissions (both valid and corrupt).
- **checksum.py**: A module that implements the logic for generating and verifying a simple 16-bit checksum.
- **crc.py**: A module that implements CRC codeword generation and verification. It supports multiple standard CRC polynomials (**CRC-8, CRC-10, CRC-16, CRC-32**).
- **injecterror.py**: A utility module with functions to inject different kinds of errors into a binary string, including random bit flips, burst errors, and errors designed to be undetectable by CRC.
- **test_cases.py**: A script for basic unit testing of the checksum and crc modules.
- **test_schemes.py**: A script designed to compare the error detection capabilities of Checksum and the different CRC polynomials by subjecting them to a variety of controlled error types.

Input and Output Format

- **Input:** The `sender.py` script takes a file path and the desired error detection method (`checksum` or `crc`) as command-line arguments. If `crc` is chosen, the specific polynomial (e.g., "`CRC-16`") must also be provided. The data within the input file is treated as a stream of characters.
- **Output:** The `sender.py` script prints the codewords it sends to the console. The `reciever.py` script prints the status of the connection and, at the end of the transmission, a summary of how many messages were correctly identified as valid or corrupt. The `test_schemes.py` script outputs a detailed comparison of how well each error detection scheme performs against various injected errors.
- **Network Protocol:** The sender and receiver communicate over a **TCP socket**. The message format is a newline-terminated string with the following structure:

method:polynomial:error_injected:codeword

- **method:** "`checksum`" or "`crc`"
- **polynomial:** The CRC polynomial used (e.g., "`CRC-16`"), or empty for `checksum`.
- **error_injected:** "`1`" if an error was intentionally injected by the sender, "`0`" otherwise.
- **codeword:** The data with the appended error detection code.

Implementation

Checksum

The checksum algorithm is implemented in `checksum.py`.

- **generate_checksum(data):** This function takes a binary string, divides it into 16-bit chunks, and calculates the checksum for each chunk. The process involves summing the 4-bit words within each chunk, folding any overflow bits (end-around carry), and then taking the one's complement of the final sum. The complemented checksum is appended to the original data chunk.
- **verify_checksum(data):** This function takes a codeword (data + checksum). It performs the same summation and carry-folding operation as the generation function. If the received data is error-free, the result of this summation will be a string of all '1's.

```
chunk = 16
bit_size = 4

def generate_checksum(data: str) -> str:
    n = len(data)
    if n % chunk != 0:
        data = '0'*(chunk - n % chunk) + data
        n = len(data)

    words = [data[i:i+chunk] for i in range(0, n, chunk)]
    checksum_strings = []

    for word in words:
        numbers = [int(word[i:i+bit_size], 2) for i in range(0, chunk, bit_size)]
```

```

checksum = sum(numbers)

while checksum >= (1 << bit_size):
    # fold carries
    checksum = (checksum >> bit_size) + (checksum & ((1 << bit_size)-1))

checksum_string = format(checksum, '0{}b'.format(bit_size))
# complement
checksum_string_complemented = ''.join('0' if b == '1' else '1' for b in
checksum_string)
checksum_strings.append(word + checksum_string_complemented)

return ''.join(checksum_strings)

def verify_checksum(data: str) -> bool:
    n = len(data)
    word_size = chunk + bit_size
    if n % word_size != 0:
        data = '0'*(word_size - n % word_size) + data
        n = len(data)

    words = [data[i:i+word_size] for i in range(0, n, word_size)]

    for word in words:
        numbers = [int(word[i:i+bit_size], 2) for i in range(0, word_size, bit_size)]
        checksum = sum(numbers)

        while checksum >= (1 << bit_size):
            checksum = (checksum >> bit_size) + (checksum & ((1 << bit_size)-1))

        if checksum != (1 << bit_size) - 1: # must be all 1s
            return False

    return True

```

CRC (Cyclic Redundancy Check)

The CRC algorithm is implemented in **crc.py**.

- **xor_division(dividend, divisor)**: This is the core of the CRC calculation. It performs a modulo-2 binary division of the dividend (the data) by the divisor (the CRC polynomial) and returns the remainder.
- **generate_crc(data, polynomial)**: This function takes the data and a CRC polynomial. It appends a number of zero bits (equal to the degree of the polynomial) to the data and then uses **xor_division** to find the remainder. This remainder is the CRC code, which is appended to the original data to form the codeword.
- **verify_crc(data, polynomial)**: This function takes the received codeword and performs the **xor_division** with the same polynomial. If there are no errors, the remainder will be zero.

```

polynomials = {
    "CRC-8": "100000111",
    "CRC-10": "11000000011",
    "CRC-16": "1100000000000101",
    "CRC-32": "100000100110000010001110110110111",
}

polynomial = ""

def xor_division(dividend:str, divisor:str)->str:
    n = len(divisor)
    pick = n
    temp = dividend[:pick]

    while pick < len(dividend):
        if temp[0] == '1':
            temp = format(int(temp, 2)^int(divisor, 2), '0' + str(n) + 'b')
            temp = temp[1:]+dividend[pick]
            pick += 1

    if temp[0] == '1':
        temp = format(int(temp, 2)^int(divisor, 2), '0' + str(n) + 'b')

    return temp[1:]

def generate_crc(data:str, polynomial:str) -> str:
    if not data:
        return ""
    # global polynomial
    if polynomial in polynomials:
        polynomial = polynomials[polynomial]

    dividend = data + '0'*(len(polynomial)-1)

    rem = xor_division(dividend, polynomial)

    data_to_send = data+rem
    return data_to_send

def verify_crc(data:str, polynomial:str)->bool:
    if not data:
        return True
    # global polynomial
    if polynomial in polynomials:
        polynomial = polynomials[polynomial]

    dividend = data

    rem = xor_division(dividend, polynomial)
    return int(rem, 2)==0

```

Error Injection

The `injecterror.py` module provides several functions to simulate transmission errors:

- `injecterror()`: Flips a specified number of random bits in the data.
- `injectbursterror()`: Simulates a burst error by flipping bits within a random contiguous block of the data.
- `injectodderror()`: Flips a random, odd number of bits.
- `undetactable_error()`: Creates a specific error pattern by XORing the data with the CRC polynomial. This error is guaranteed to be undetectable by that same CRC polynomial.

```
import random

def injecterror(data: str, errcnt:int = 1) -> str:
    """
    n = len(data)
    data_list = list(data)
    for _ in range(errcnt):
        index = random.randint(0, n-1)
        if data_list[index] == '1':
            data_list[index] = '0'
        else:
            data_list[index] = '1'
    return ''.join(data_list)

def injectbursterror(data: str, burstsize: int = 1) -> str:
    n = len(data)
    index = random.randint(0, n-burstsize)
    data_list = list(data)
    for i in range(index, index+burstsize):
        if random.random() < 0.4:
            if data_list[i] == '1':
                data_list[i] = '0'
            else:
                data_list[i] = '1'
    return ''.join(data_list)

def injectodderror(data: str) -> str:
    n = len(data)
    data_list = list(data)
    flip_count = random.choice([i for i in range(1, n, 2)])
    indices = random.sample(range(n), flip_count)

    for idx in indices:
        data_list[idx] = '0' if data_list[idx] == '1' else '1'

    return ''.join(data_list)

def undetectable_error(data: str, poly: str, shift: int = 0) -> str:
    n = len(data)
    m = len(poly)
```

```

# pad polynomial to length of data, shifted into place
mask = ['0'] * n
for i, bit in enumerate(poly):
    if bit == '1':
        pos = n - m - shift + i # align poly with right side then shift
        if 0 <= pos < n:
            mask[pos] = '1'

# XOR data with mask
corrupted = [
    '1' if (d != m) else '0'
    for d, m in zip(data, mask)
]

return ''.join(corrupted)

```

Sender.py

A Python script to send file data over a TCP socket using either checksum or CRC error detection methods.

Each chunk of the file is processed to generate a codeword (checksum or CRC), with optional error injection.

The codeword and metadata are sent to a receiver server.

Usage:

```
python sender.py <file_path> <method> [crc_polynomial]
```

Arguments:

file_path	Path to the input file to be sent.
method	Error detection method: "checksum" or "crc".
crc_polynomial	(Required if method is "crc") Polynomial to use for CRC calculation.

Modules:

socket	For network communication.
sys	For command-line argument handling.
checksum	Custom module for checksum generation.
crc	Custom module for CRC generation.
injecterror	Custom module for error injection.
random	For probabilistic error injection.

Functions:

main()	Handles argument parsing, file reading, codeword generation, error injection, and data transmission.
--------	--

Notes:

- Data is sent in chunks of 16 characters.
- With 20% probability, an error is injected into the codeword before sending.
- The message format sent to the receiver is:

```
"<method>:<polynomial>:<error>:<codeword>\n"
```

- Requires a receiver server listening on HOST:PORT.

```

import socket
import sys
import checksum
import crc
import injecterror
import random

```

```

HOST = '127.0.0.1'
PORT = 3000

def main():
    if len(sys.argv) < 3:
        print("Usage: python sender.py <file_path> <method> [crc_polynomial]")
        return

    file_path = sys.argv[1]
    method = sys.argv[2]

    chunk_size = 16 # characters per chunk
    polynomial = ""

    if method == "crc":
        if len(sys.argv) < 4:
            print("Usage: python sender.py <file_path> crc <crc_polynomial>")
            return
        polynomial = sys.argv[3]
        # crc.polynomial = polynomial # set global polynomial

    try:
        with open(file_path, 'r') as f, socket.socket(socket.AF_INET, socket.SOCK_STREAM)
as s:
        s.connect((HOST, PORT))
        print(f"Connected to {HOST}:{PORT}")

        while True:
            data = f.read(chunk_size)
            if not data:
                break

            if method == "checksum":
                codeword = checksum.generate_checksum(data)
            elif method == "crc":
                codeword = crc.generate_crc(data, polynomial)
            else:
                print(f"Error: Unknown method '{method}'")
                return

            # Inject error with 20% probability
            error = 0
            if random.random() < 0.2:
                codeword = injecterror.injecterror(codeword)
                error = 1

            message = f"{method}:{polynomial}:{error}:{codeword}\n"
            s.sendall(message.encode("utf-8"))
            print(f"Sent: {codeword}")

        print("File transfer complete.")

    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")

```



```
        return

if __name__ == "__main__":
    main()
```

Reciever.py

A TCP server for receiving and verifying messages using checksum or CRC error detection. This script listens for incoming connections on a specified host and port. It receives messages from a client, each message containing the error detection method, polynomial (for CRC), error flag, and codeword, separated by colons. Messages are processed line by line. The server verifies each message using the specified error detection method (checksum or CRC) and counts the number of correctly detected errors.

Modules required:

- socket: For network communication.
- checksum: For checksum verification.
- crc: For CRC verification.

Attributes:

- HOST (str): The IP address to bind the server.
- PORT (int): The port number to bind the server.

Workflow:

1. Bind and listen on HOST:PORT.
2. Accept a client connection.
3. Receive data in chunks, buffering until a newline is found.
4. For each message:
 - Parse method, polynomial, error flag, and codeword.
 - Verify codeword using the specified method.
 - Track correct detections.
5. Print summary of correct detections after transfer is complete.

```
import socket
import checksum
import crc

HOST = '127.0.0.1'
PORT = 3000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print(f"Server listening on {HOST}:{PORT}")

    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        buffer = ""
        message_count = 0
        correct_detection_count = 0

        while True:
```

```
data = conn.recv(1024)
if not data:
    print("File transfer complete.")
    break

buffer += data.decode("utf-8")

# Process messages separated by newline
while "\n" in buffer:
    message, buffer = buffer.split("\n", 1)
    if not message.strip():
        continue

    try:
        method, polynomial, error, codeword = message.split(":", 3)
        message_count += 1
    except ValueError:
        print(f"Invalid message format: {message}")
        continue

    is_valid = False
    if method == "checksum":
        is_valid = checksum.verify_checksum(codeword)
    elif method == "crc":
        is_valid = crc.verify_crc(codeword, polynomial)
    else:
        print(f"Unknown method: {method}")
        continue

    if is_valid == bool(error):
        correct_detection_count += 1

print(f"Correct detection: {correct_detection_count}/{message_count}")
```

Test Cases

The correctness and robustness of the implementation were verified using the following test cases, executed by the `test_cases.py` and `test_schemes.py` scripts.

1. Zero-Error Verification (``test_cases.py``):

- **Test Data:** A variety of binary strings, including "Hello" in binary, all '0's, all '1's, and an empty string.
- **Check:** For both Checksum and all CRC variants, generate a codeword and then immediately verify it.
- **Expected Outcome:** The verification function should always return True, confirming that the algorithms correctly identify error-free data.

2. Random Bit Errors (``test_schemes.py``):

- **Test Data:** A 32-bit binary string.
- **Check:** Inject 1, 2, and 5 random bit errors into the codeword.
- **Expected Outcome:** All schemes (Checksum and all CRCs) are expected to detect these errors.

3. Burst Errors (``test_schemes.py``):

- **Test Data:** A 32-bit binary string.
- **Check:** Inject burst errors of varying lengths (1, 2, 17, 20 bits).
- **Expected Outcome:** CRC is expected to be highly effective, especially for burst errors smaller than the polynomial degree. Checksum's performance may vary.

4. Odd-Numbered Errors (``test_schemes.py``):

- **Test Data:** A 32-bit binary string.
- **Check:** Inject a random, odd number of bit flips.
- **Expected Outcome:** All CRC polynomials can detect any odd number of single-bit errors. Checksum should also perform well.

5. Guaranteed Undetectable Errors (``test_schemes.py``):

- **Test Data:** A 32-bit binary string.
- **Check:** For each CRC polynomial, generate a codeword and then inject the corresponding "undetectable" error pattern.
- **Expected Outcome:** The `verify_crc` function for that specific polynomial should return True, demonstrating the theoretical limitation of CRC.

Testing if they work

Implemented in `test_cases.py`

```
import checksum
import crc
import injecterror

binary_string = ["0100100001100101011011000110110001101111", "11110000111100001111",
"0000000000000000", "1111111111111111", ""]
polynomials = {
    "CRC-8": "100000111",
    "CRC-10": "11000000011",
    "CRC-16": "11000000000000101",
    "CRC-32": "100000100110000010001110110110111",
}

for word in binary_string:
    codeword_checksum = checksum.generate_checksum(word)
    print("[Checksum] codeword: ", codeword_checksum)

    correct_data = checksum.verify_checksum(codeword_checksum)
    print("[Checksum] correct data recieved: ", correct_data)

for key in polynomials:
    print()
    for word in binary_string:
        codeword_crc = crc.generate_crc(word, key)
        print(f"[CRC][{key}] codeword: ", codeword_crc)

        correct_data = crc.verify_crc(codeword_crc, key)
        print(f"[CRC][{key}] correct data recieved: ", correct_data)
```

Testing error detection capabilities and some undetectable errors

Implemented in `test_schemes.py`

Main functionalities:

- Injects random single/multi-bit errors and checks detection.
- Injects burst errors of various lengths and checks detection.
- Injects odd-number errors and checks detection.
- Injects guaranteed undetectable errors for each CRC polynomial and checks detection.
- Reports which bits were flipped and whether each scheme detected the error.

```
import checksum
import crc
import injecterror

binary_string = "111100001111000000001111000011110000"
polynomials = {
    "CRC-8": "100000111",
    "CRC-10": "11000000011",
    "CRC-16": "11000000000000101",
```

```

    "CRC-32": "100000100110000010001110110110111",
}

errors = [1, 2, 17, 20] # burst error lengths

def show_bits_flipped(original: str, corrupted: str):
    """Return indices where bits differ (0 = rightmost/LSB)."""
    diffs = []
    for i, (a, b) in enumerate(zip(reversed(original), reversed(corrupted))):
        if a != b:
            diffs.append(i)
    return diffs

print("=== ERROR DETECTION COMPARISON ===")
print(f"Original data length: {len(binary_string)} bits")

# 1. Random single/multi-bit errors
print("\n--- RANDOM BIT ERRORS ---")
for count in [1, 2, 5]:
    err_data = injecterror.injecterror(binary_string, count)
    diffs = show_bits_flipped(binary_string, err_data)
    print(f"\nInjected {count} random errors at {diffs}")

    # checksum
    chk_data = checksum.generate_checksum(binary_string)
    chk_err = injecterror.injecterror(chk_data, count)
    print(f"[Checksum] Detected={not checksum.verify_checksum(chk_err)}")

    # CRCs
    for key, poly in polynomials.items():
        # crc.polynomial = poly
        crc_data = crc.generate_crc(binary_string, poly)
        crc_err = injecterror.injecterror(crc_data, count)
        print(f"[{key}] Detected={not crc.verify_crc(crc_err, poly)}")

# 2. Burst errors
print("\n--- BURST ERRORS ---")
for size in errors:
    print(f"\nBurst size = {size}")
    chk_data = checksum.generate_checksum(binary_string)
    chk_err = injecterror.injectbursterror(chk_data, size)
    print(f"[Checksum] Detected={not checksum.verify_checksum(chk_err)}")

    for key, poly in polynomials.items():
        # crc.polynomial = poly
        crc_data = crc.generate_crc(binary_string, poly)
        crc_err = injecterror.injectbursterror(crc_data, size)
        print(f"[{key}] Detected={not crc.verify_crc(crc_err, poly)}")

# 3. Odd-number errors

```

```

print("\n--- ODD-NUMBER ERRORS ---\n")
odd_err = injecterror.injectodderror(binary_string)
diffs = show_bits_flipped(binary_string, odd_err)
print(f"Injected odd-error at positions: {diffs}")

chk_data = checksum.generate_checksum(binary_string)
chk_err = injecterror.injectodderror(chk_data)
print(f"[Checksum] Detected={not checksum.verify_checksum(chk_err)}")

for key, poly in polynomials.items():
    # crc.polynomial = poly
    crc_data = crc.generate_crc(binary_string, poly)
    crc_err = injecterror.injectodderror(crc_data)
    print(f"[{key}] Detected={not crc.verify_crc(crc_err, poly)}")

# 4. Guaranteed undetectable errors
print("\n--- GUARANTEED UNDETECTABLE ERRORS ---\n")
for key, poly in polynomials.items():
    # crc.polynomial = poly
    crc_data = crc.generate_crc(binary_string, poly)
    undet = injecterror.undetectable_error(crc_data, poly)
    diffs = show_bits_flipped(crc_data, undet)
    detected = not crc.verify_crc(undet, poly)
    print(f"[{key}] Detected={detected}, Flipped bits={diffs}")

```

Results

The `test_schemes.py` script was executed to compare the error detection capabilities. The results can be summarized in the following table:

Error Type	Checksum	CRC-8	CRC-10	CRC-16	CRC-32
1 Random Bit Error	Detected	Detected	Detected	Detected	Detected
2 Random Bit Errors	Detected	Detected	Detected	Detected	Detected
Odd Number of Bit Errors	Detected	Detected	Detected	Detected	Detected
Burst Errors (small)	Varies	Detected	Detected	Detected	Detected
Burst Errors (large)	Varies	Varies	Varies	Detected	Detected
Undetectable Error (CRC-8)	Varies	Not detected	Detected	Detected	Detected
Undetectable Error (CRC-16)	Varies	Detected	Detected	Not detected	Detected

Analysis

The results from the tests highlight the different strengths and weaknesses of the Checksum and CRC algorithms.

- Checksum:** The implemented checksum is simple and fast. It is effective at detecting single-bit errors and most random multi-bit errors. However, it has weaknesses. For example, it can fail to detect errors where two bits are in opposite directions, or where one word is substituted for another with same sum. Its ability to detect burst errors is not as reliable as CRC.

- **CRC:** CRC is a much more powerful error detection mechanism. Its performance is related to the choice of the generator polynomial.
 - All tested CRC polynomials were able to detect all single-bit, double-bit, and odd-numbered bit errors.
 - CRC is particularly effective at detecting burst errors. A CRC with a degree r can detect all burst errors of length less than or equal to r .
 - The `test_schemes.py` script successfully demonstrated the concept of undetectable errors. For any given CRC polynomial, there exists a specific error pattern (which is a multiple of the polynomial) that will result in a zero remainder, thus fooling the algorithm. However, the probability of such a specific error occurring naturally is very low. Longer polynomials have a lower probability of failing to detect errors.

Possible Improvements

- The `checksum.py` module uses fixed global variables for `chunk` and `bit_size`. This could be made more flexible by passing them as parameters to the functions.
- The user interface of the `sender.py` script could be improved with more interactive prompts or a graphical interface.
- The project could be extended to include error correction codes, such as Hamming codes, for a more comprehensive study.

Comments

Key Learnings & Reflections:

- **Practical Application:** The assignment effectively bridged the gap between networking theory and practice. Implementing the Checksum and CRC algorithms provided a deeper, more intuitive understanding of their operation than textbook descriptions alone.
- **Technical Challenge:** The implementation of CRC's modulo-2 division was moderately challenging and required a precise understanding of bitwise logic.
- **Comparative Analysis:** The most valuable aspect was designing targeted tests to compare the two schemes. This made their respective strengths and weaknesses—such as the checksum's vulnerability to compensating errors—tangibly clear.
- **Suggested Improvement:** To further enhance the learning experience, a visualization tool that illustrates how errors are introduced and detected by each algorithm would be a valuable addition.
- **Overall Impression:** This was an engaging and effective assignment that successfully solidified my understanding of error detection codes.