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.

```
    1 # Snippet from checksum.py
    2 def generate_checksum(data: str) -> str:
    3     # ... (padding and chunking logic)
    4     for word in words:
    5         numbers = [int(word[i:i+bit_size], 2) for i in range(0, chunk,
bit_size)]
    6         checksum = sum(numbers)
    7
    8         while checksum >= (1 << bit_size):
    9             # fold carries
   10             checksum = (checksum >> bit_size) + (checksum & ((1 << bit_size)-1))
   11
   12         checksum_string = format(checksum, '0{}b'.format(bit_size))
```

```
   13         # complement
   14         checksum_string_complemented = ''.join('0' if b == '1' else '1' for b in
checksum_string)
   15         checksum_strings.append(word + checksum_string_complemented)
   16
   17     return ''.join(checksum_strings)
```

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.

```
    1 # Snippet from crc.py
    2 def generate_crc(data:str, polynomial:str) -> str:
    3     if not data:
    4         return ""
    5     if polynomial in polynomials:
    6         polynomial = polynomials[polynomial]
    7
    8     dividend = data + '0'*(len(polynomial)-1)
    9     rem = xor_division(dividend, polynomial)
   10     data_to_send = data+rem
   11     return data_to_send
```

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.
- **undetectable_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.

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.

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 of 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.