Implementation of CRC for

Error Detection and Single-Bit Error Correction

CSE 2213 : Data and Telecommunication Lab

Batch : 29 / 2nd Year 2nd Semester 2024

Date

23 May, 2025

Submitted by

Atiya Fahmida (Roll-49)
Jubair Ahammad Akter (Roll-59)

Course Instructors

Dr. Md. Mustafizur Rahman (Professor)
Mr. Palash Roy (Lecturer)

Department of Computer Science and Engineering

University of Dhaka

Table of Contents

1 Introduction

1.1 What is CRC ?

Cyclic Redundancy Check (CRC) is an code used in digital data communication and data storage systems for error-detecting to identify sudden accidental changes to raw or data. It generates binary data from the given data using ASCII code and then add or append x zeros (where x= $\text{key}_s ize - 1) intheend(Supposedata = 10010011, so, appendeddata = 10010011000[letkey_size = 4], then\text{F(x)} = \text{x}^{10} + x^7 + x^4 + x^3)$ and dividing it by a fixed **Generator Polynomial** (Suppose the key=1011, then $G(x) = x^3 + x + 1$), which is shared by both sender and receiver.

The remainder of this division is called the **CRC code** and is appended to the main data during transmission. The receiver again genarates the same division and verifies that the result is zero for deciding errorless data otherwise there must be error somewhere in the data.

1.2 Necessity of CRC

Cyclic Redundancy Check (CRC) is widely used due to its simplicity, effectiveness, and reliability in ensuring data integrity during transmission.

- Detects single-bit, multi-bit, and burst errors
- Can solve also single-bit error
- More reliable than parity and checksum methods
- Commonly used in Ethernet, hard disk drives (HDDs), and communication protocols like HDLC and CAN bus
- Ensures data integrity but does not correct errors without additional techniques (e.g., Hamming code, Forward Error Correction)

2 Objective(s)

- To understand the fundamental theory and operation of CRC (Cyclic Redundancy Check).
- To implement CRC encoding and decoding using various polynomial division and compare them.
- To evaluate CRC's effectiveness in ensuring data integrity.
- To implement single-bit error correction using CRC.
- To simulate data transmission and detect errors using CRC.
- To analyze the pros and cons of CRC.

3 Algorithms / Pseudocode

3.1 CRC Generation and Detection

1. Read input data as binary string.
2. Append r-1 zeros to the data (where r = length of generator).
3. Use XOR to divide the data by generator (bitwise).
4. The remainder is the CRC which is appended to original data.
5. We will work for every generator polynomial (e.g., for CRC-8, CRC-10, CRC-16, CRC-32).

3.2 Single-Bit Error Correction

1. At receiver, perform same division and check the reminder.
2. After CRC detects an error, iterate over each bit.
3. Flip one bit and recompute CRC.
4. If CRC remainder becomes 0, correct bit is found.
5. Replace bit and accept the corrected data.
6. Did the same work for burst error, but this can not be solved or errorless by CRC. Only sinlge-bit is solvable

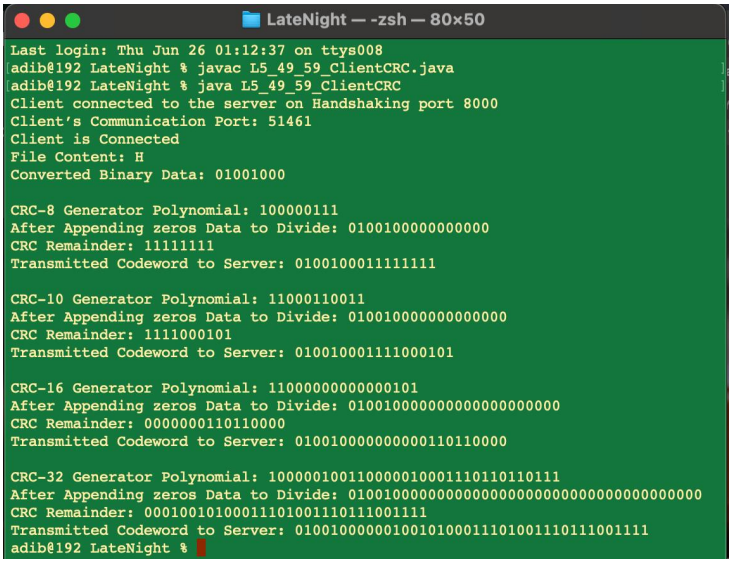
4 Implementation

4.1 Client Side Code

```java
import java.io.*;
import java.net.*;

public class L5_49_59_ClientCRC {
    // CRC polynomials (binary, no leading 1)
    private static final String CRC8_POLY = "100000111";       // CRC-8: x^8 + x^2 + x^1 + x^0
    private static final String CRC10_POLY = "11000110011";    // CRC-10: x^10 + x^9 + x^5 + x^4 + x^1 + x^0
    private static final String CRC16_POLY = "11000000000000101"; // CRC-16: x^16 + x^15 + x^2 + x^0
    private static final String CRC32_POLY = "100000100110000010001110110110111"; // CRC-32

    public static void main(String[] args) throws IOException {
        String inputFile = "input.txt";
        String data = readFile(inputFile);
        System.out.println("Client connected to the server on Handshaking port 8000");
        Socket socket = new Socket("localhost", 8000);
        int clientPort = socket.getLocalPort();
        System.out.println("Client's Communication Port: " + clientPort);
        System.out.println("Client is Connected");
        System.out.println("File Content: " + data);
        String binaryData = toBinaryString(data);
        System.out.println("Converted Binary Data: " + binaryData);

        // For each CRC type
        String[] crcTypes = {"CRC-8", "CRC-10", "CRC-16", "CRC-32"};
        for (String crcType : crcTypes) {
            String poly = getPoly(crcType);
            int polyLen = poly.length();
            System.out.println("\n" + crcType + " Generator Polynomial: " + poly);
            String dataToDivide = binaryData + "0".repeat(polyLen - 1);
            System.out.println("After Appending zeros Data to Divide: " + dataToDivide);
            String remainder = mod2div(dataToDivide, poly);
            System.out.println("CRC Remainder: " + remainder);
            String codeword = binaryData + remainder;
            System.out.println("Transmitted Codeword to Server: " + codeword);

            // Send codeword and CRC type to server
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            out.println(codeword);
            out.println(crcType);
        }
        socket.close();
    }

    private static String readFile(String filename) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(filename));
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) sb.append(line);
        br.close();
        return sb.toString();
    }

    private static String toBinaryString(String s) {
        StringBuilder sb = new StringBuilder();
        for (char c : s.toCharArray()) {
            sb.append(String.format("%8s", Integer.toBinaryString(c)).replace(' ', '0'));
        }
        return sb.toString();
    }

    private static String getPoly(String type) {
        switch (type) {
            case "CRC-8": return CRC8_POLY;
            case "CRC-10": return CRC10_POLY;
            case "CRC-16": return CRC16_POLY;
            case "CRC-32": return CRC32_POLY;
            default: throw new IllegalArgumentException("Unknown CRC type");
        }
    }

    // Modulo-2 division
    private static String mod2div(String dividend, String divisor) {
        int pick = divisor.length();
        String tmp = dividend.substring(0, pick);
        int n = dividend.length();
        while (pick < n) {
            if (tmp.charAt(0) == '1')
                tmp = xor(divisor, tmp) + dividend.charAt(pick);
            else
                tmp = xor("0".repeat(pick), tmp) + dividend.charAt(pick);
            tmp = tmp.substring(1);
            pick++;
        }
        if (tmp.charAt(0) == '1')
            tmp = xor(divisor, tmp);
        else
            tmp = xor("0".repeat(pick), tmp);
        return tmp.substring(1);
    }

    private static String xor(String a, String b) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < b.length(); i++)
            result.append(a.charAt(i) == b.charAt(i) ? '0' : '1');
        return result.toString();
    }
}
```

FIGURE 1 – Client Side (Java) - Sending data with CRC

4.2 Client Side Terminal



```
LateNight — -zsh — 80×50
Last login: Thu Jun 26 01:12:37 on ttys008
adib@192 LateNight % javac L5_49_59_ClientCRC.java
adib@192 LateNight % java L5_49_59_ClientCRC
Client connected to the server on Handshaking port 8000
Client's Communication Port: 51461
Client is Connected
File Content: H
Converted Binary Data: 01001000

CRC-8 Generator Polynomial: 100000111
After Appending zeros Data to Divide: 0100100000000000
CRC Remainder: 11111111
Transmitted Codeword to Server: 0100100011111111

CRC-10 Generator Polynomial: 11000110011
After Appending zeros Data to Divide: 0100100000000000000
CRC Remainder: 1111000101
Transmitted Codeword to Server: 0100100011111000101

CRC-16 Generator Polynomial: 11000000000000101
After Appending zeros Data to Divide: 0100100000000000000000000
CRC Remainder: 0000000110110000
Transmitted Codeword to Server: 0100100000000000110110000

CRC-32 Generator Polynomial: 100000100110000010001110110110111
After Appending zeros Data to Divide: 0100100000000000000000000000000000000000
CRC Remainder: 00010010100011101001110111001111
Transmitted Codeword to Server: 0100100000010010100011101001110111001111
adib@192 LateNight %
```

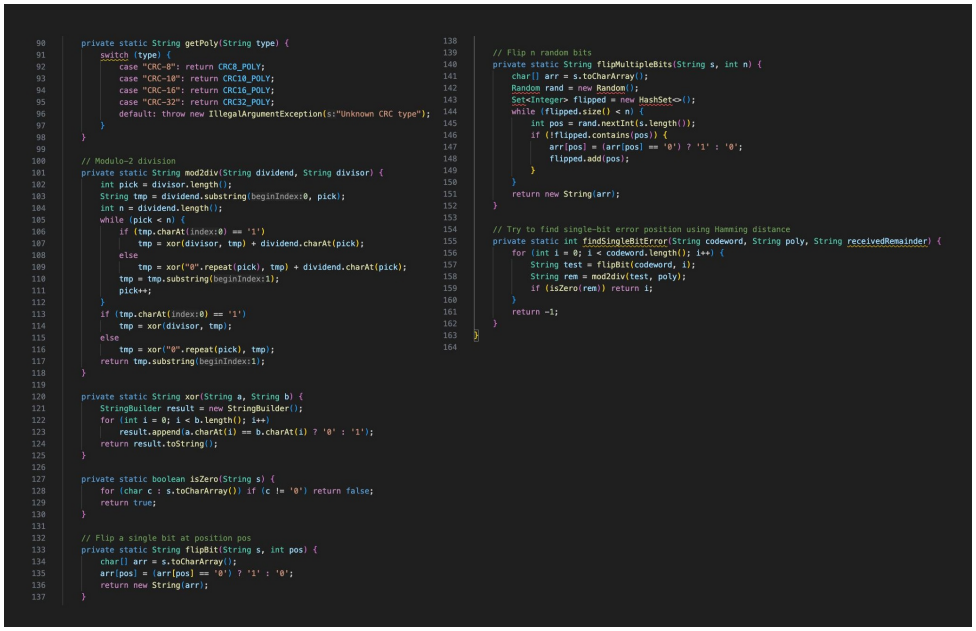
FIGURE 2 – Client Side (Terminal) - Sending data with CRC

4.3 Server Side Code 1

```java
import java.io.*;
import java.net.*;
import java.util.*;

public class L5_49_59_ServerCRC {
    // CRC polynomials (binary, no leading 1)
    private static final String CRC8_POLY = "100000111";     // CRC-8: x^8 + x^2 + x^1 + x^0
    private static final String CRC10_POLY = "11000110011";   // CRC-10: x^10 + x^9 + x^5 + x^4 + x^1 + x^0
    private static final String CRC16_POLY = "11000000000000101"; // CRC-16: x^16 + x^15 + x^2 + x^0
    private static final String CRC32_POLY = "10000010011000001000011101110110111"; // CRC-32 IEEE 802.3

    Run | Debug | Run main | Debug main
    public static void main(String[] args) throws IOException {
        int handshakePort = 8000; // Handshake port
        ServerSocket serverSocket = new ServerSocket(handshakePort);
        System.out.println(x:"Server is connecting");
        System.out.println(x:"Waiting for the client");
        Socket clientSocket = serverSocket.accept();
        int clientPort = clientSocket.getPort();
        System.out.println("Client request is accepted at port no: " + clientPort);
        System.out.println("Server's Communication Port: " + handshakePort);

        BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), autoFlush:true);

        // For each CRC type
        String[] crcTypes = {"CRC-8", "CRC-10", "CRC-16", "CRC-32"};
        for (String crcType : crcTypes) {
            String codeword = in.readLine();
            String receivedType = in.readLine();
            if (!crcType.equals(receivedType)) {
                System.out.println("Expected CRC type " + crcType + " but got " + receivedType);
                continue;
            }
            String poly = getPoly(crcType);
            System.out.println("\n--- " + crcType + " ---");
            System.out.println("Generator Polynomial: " + poly);
            System.out.println("Received Codeword: " + codeword);
            System.out.println("CRC name: " + crcType);
            String remainder = mod2div(codeword, poly);
            System.out.println("Calculated Remainder: " + remainder);
            if (isZero(remainder)) {
                System.out.println(x:"No error detected in transmission.\n");
            } else {
                System.out.println(x:"Error detected in transmission!\n");
            }

            // Simulate single-bit error
            int flipPos = new Random().nextInt(codeword.length());
            String singleBitError = flipBit(codeword, flipPos);
            System.out.println(x:"Server Output with Error Simulation (Bit flipped)");
            System.out.println("Received Codeword: " + singleBitError);
            String remErr = mod2div(singleBitError, poly);
            System.out.println("Calculated Remainder: " + remErr);
            if (!isZero(remErr)) {
                System.out.println(x:"Error detected in transmission!");
                // Find error position by flipping each bit and checking CRC
                int foundPos = -1;
                for (int i = 0; i < singleBitError.length(); i++) {
                    String test = flipBit(singleBitError, i);
                    String rem = mod2div(test, poly);
                    if (isZero(rem)) {
                        foundPos = i + 1; // 1-based position
                        break;
                    }
                }
                if (foundPos != -1) {
                    System.out.println("Error in position: " + foundPos);
                } else {
                    System.out.println(x:"Error position could not be determined.");
                }
            }

            // Simulate burst error
            String burstError = flipMultipleBits(codeword, n:3); // flip 3 random bits
            System.out.println(x:"Server Output with Burst Error Simulation (Multiple bits flipped)");
            System.out.println("Received Codeword: " + burstError);
            String remBurst = mod2div(burstError, poly);
            System.out.println("Calculated Remainder: " + remBurst);
            if (!isZero(remBurst)) {
                System.out.println(x:"Error detected in transmission!");
            }
        }

        in.close();
        out.close();
        clientSocket.close();
        serverSocket.close();
    }

    private static String getPoly(String type) {
        switch (type) {
            case "CRC-8": return CRC8_POLY;
            case "CRC-10": return CRC10_POLY;
            case "CRC-16": return CRC16_POLY;
            case "CRC-32": return CRC32_POLY;
            default: throw new IllegalArgumentException(s:"Unknown CRC type");
        }
    }
```

FIGURE 3 – Server Side 1 (Java) - Receiving and validating CRC

4.4 Server Side Code 2



```java
    private static String getPoly(String type) {
        switch (type) {
            case "CRC-8": return CRC8_POLY;
            case "CRC-10": return CRC10_POLY;
            case "CRC-16": return CRC16_POLY;
            case "CRC-32": return CRC32_POLY;
            default: throw new IllegalArgumentException(s:"Unknown CRC type");
        }
    }

    // Modulo-2 division
    private static String mod2div(String dividend, String divisor) {
        int pick = divisor.length();
        String tmp = dividend.substring(beginIndex:0, pick);
        int n = dividend.length();
        while (pick < n) {
            if (tmp.charAt(index:0) == '1')
                tmp = xor(divisor, tmp) + dividend.charAt(pick);
            else
                tmp = xor("0".repeat(pick), tmp) + dividend.charAt(pick);
            tmp = tmp.substring(beginIndex:1);
            pick++;
        }
        if (tmp.charAt(index:0) == '1')
            tmp = xor(divisor, tmp);
        else
            tmp = xor("0".repeat(pick), tmp);
        return tmp.substring(beginIndex:1);
    }

    private static String xor(String a, String b) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < b.length(); i++)
            result.append(a.charAt(i) == b.charAt(i) ? '0' : '1');
        return result.toString();
    }

    private static boolean isZero(String s) {
        for (char c : s.toCharArray()) if (c != '0') return false;
        return true;
    }

    // Flip a single bit at position pos
    private static String flipBit(String s, int pos) {
        char[] arr = s.toCharArray();
        arr[pos] = (arr[pos] == '0') ? '1' : '0';
        return new String(arr);
    }

    // Flip n random bits
    private static String flipMultipleBits(String s, int n) {
        char[] arr = s.toCharArray();
        Random rand = new Random();
        Set<Integer> flipped = new HashSet<>();
        while (flipped.size() < n) {
            int pos = rand.nextInt(s.length());
            if (!flipped.contains(pos)) {
                arr[pos] = (arr[pos] == '0') ? '1' : '0';
                flipped.add(pos);
            }
        }
        return new String(arr);
    }

    // Try to find single-bit error position using Hamming distance
    private static int findSingleBitError(String codeword, String poly, String receivedRemainder) {
        for (int i = 0; i < codeword.length(); i++) {
            String test = flipBit(codeword, i);
            String rem = mod2div(test, poly);
            if (isZero(rem)) return i;
        }
        return -1;
    }
}
```

FIGURE 4 – Server Side 2 (Java) - Necessary functions for receiving and validating CRC

5 Result Analysis

CRC Polynomials Tested : CRC-8, CRC-10, CRC-16, CRC-32

- **Single-Bit Error Detection :** All four CRCs successfully detected single-bit errors.
- **Burst Error Detection :** It can be detected only. But find out the main data is difficult.
- **Single-Bit Correction :** Java code with brute-force bit flipping and CRC recheck was effective.

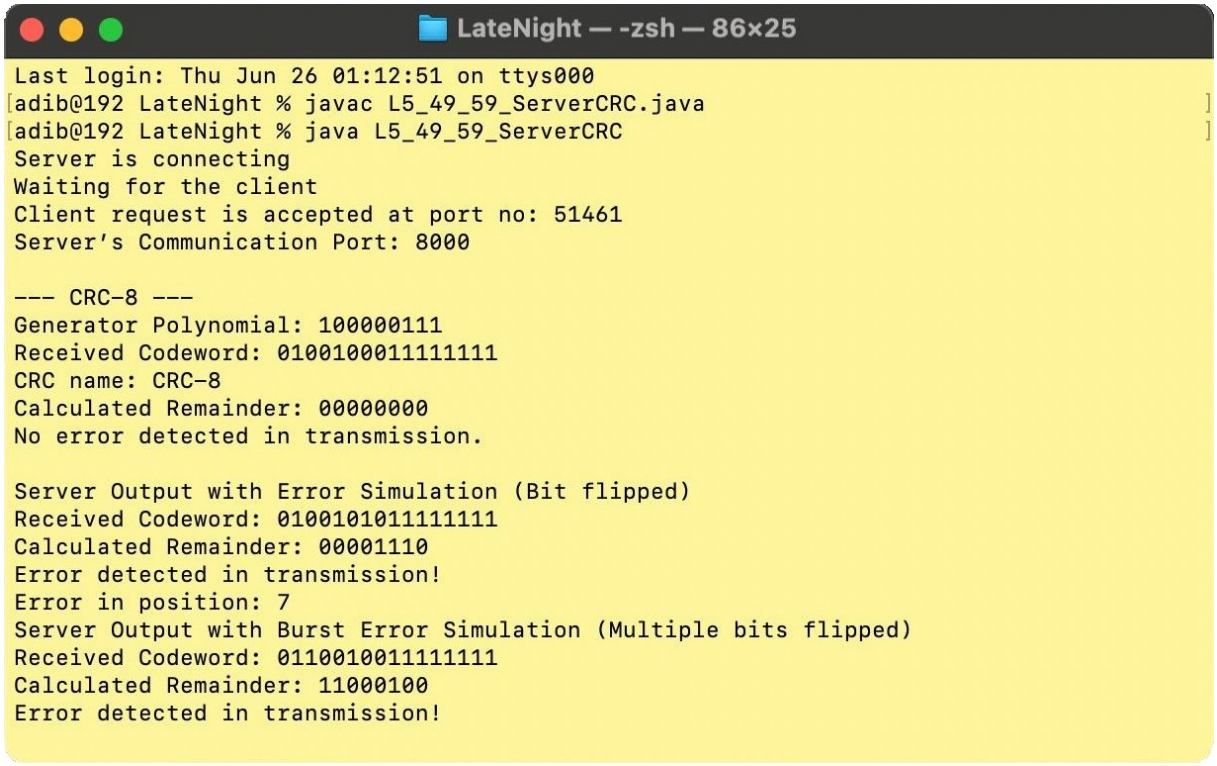


FIGURE 5 – Example Output - CRC-8 detecting burst error and correcting single-bit error

FIGURE 6 – Example Output - CRC-10 detecting burst error and correcting single-bit error

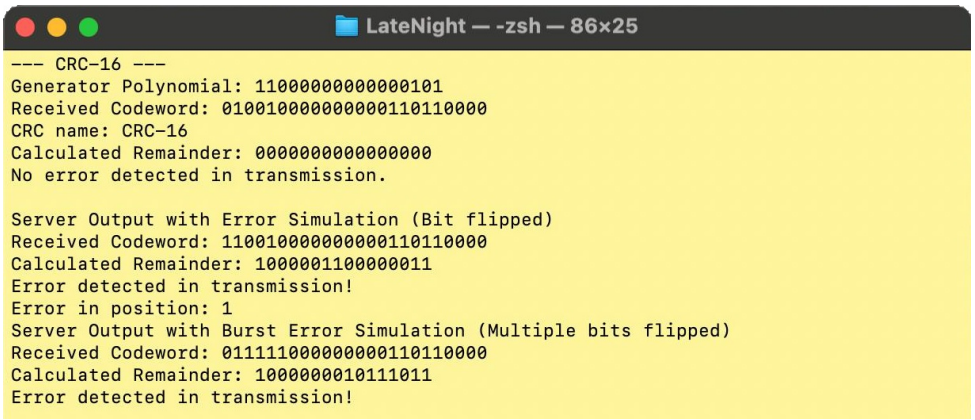


FIGURE 7 – Example Output - CRC-16 detecting burst error and correcting single-bit error

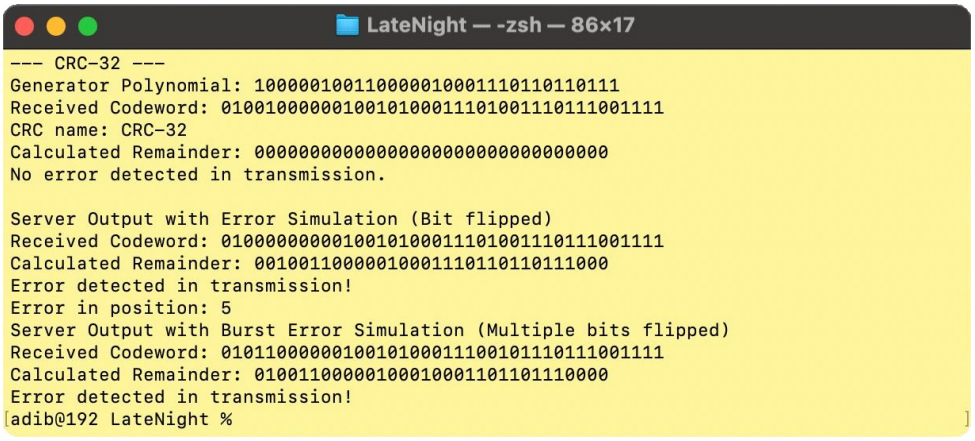


FIGURE 8 – Example Output - CRC-32 detecting burst error and correcting single-bit error

6 Discussion

6.1 Why Use Different CRC Generators ?

- **CRC-8 :** Fast but weak against burst errors.
- **CRC-10 :** Slightly better than CRC-8.
- **CRC-16 :** Good for protocols like USB, Modbus.
- **CRC-32 :** Best for large data streams (e.g., file transfers).

6.2 Limitations

- CRC only detects errors ; it doesn't correct them.
- Multiple-bit errors may not be uniquely detectable.
- Hamming logic only works for 1-bit errors.

6.3 Detection of Single-Bit Errors

If data is sent without error we will get reminder zero. But if a single bit is fliped, then we will flip only a single bit every time by loop of the main codeword, we get from client. So if only a single bit is fliped, then one time the data will be correct and we will get CRC reminder zero.

6.4 Why Multiple-Bit Errors Cannot Be Corrected Using CRC

CRC is primarily designed for error detection, not correction. This limitation arises from the implementation process itself :

- **Error Detection Mechanism :** CRC is based on polynomial division in which data is divided by a generator polynomial. The remainder produced indicates whether an error has occurred. When multiple bits are altered (e.g., two bits flipped), the resulting error may still match the checksum for some combinations of errors, causing the CRC to fail in detecting the error. CRC, therefore, cannot uniquely detect all types of multiple-bit errors, especially when the number of altered bits is large.
- **No Error-Correction Mechanism :** The CRC mechanism is purely focused on detection. Once an error is detected (such as a mismatch between the computed checksum and the received checksum), it cannot provide information on where the error occurred, let alone how to fix it. For error correction, more advanced algorithms like Hamming or Reed-Solomon codes are used, as they include information about the location and type of the error, which CRC does not.

- **Polynomial Division Limitation :** The polynomial division used in CRC works by detecting specific patterns of errors, and when multiple bits are flipped, the error pattern can result in a valid checksum, which means CRC is unable to correct those errors. Correcting errors would require additional metadata or redundancy in the transmission, something CRC does not inherently provide.
- **Ambiguity in Multiple-Bit Errors :** CRC's limitation is that, for multiple-bit errors, there are many combinations of flipped bits that might yield the same checksum. Because of this, CRC can't distinguish between these multiple errors and might report no error or a false positive. This ambiguity makes it ineffective at correcting multiple errors, as the original bit pattern can no longer be reconstructed.

7 Learning and Difficulties

7.1 Main Learning Achievements

Throughout this project, we gained several significant insights, including :

- **Mathematical Concepts :** Grasping the principles of polynomial arithmetic over GF(2) and its application in error detection codes provided a solid theoretical base for implementing CRC.
- **Algorithm Design :** Translating mathematical theories into functional Java code required detailed attention to bitwise operations and binary manipulations.
- **Network Communication :** Implementing CRC in a client-server model enhanced our understanding of networking protocols and their error-handling mechanisms.
- **Error Pattern Examination :** Analyzing various error patterns helped us understand the advantages and limitations of different CRC polynomials in real-world scenarios.
- **Efficiency Considerations :** Striking a balance between error detection effectiveness and computational efficiency was a key consideration in the practical application of CRC.

7.2 Challenges Faced

Several challenges arose during the implementation process :

- **Complexity of Bit Manipulation :** Handling binary operations in Java demanded precise management of data types and bit shifts, with a special focus on ensuring correct polynomial division.
- **Syndrome Computation Precision :** Achieving accurate syndrome calculation for error detection required careful bit-level handling and debugging to ensure mathematical integrity.

- **Synchronizing Network Communication :** Maintaining seamless communication between client and server while ensuring data consistency required effective error handling and timeout protocols.
- **Comprehensive Error Testing :** Developing robust test cases to verify error detection and correction in diverse scenarios was time-intensive but essential for thorough validation.
- **Optimizing Performance :** Refining CRC algorithms for better computational efficiency while maintaining their accuracy required multiple iterations and performance profiling.

7.3 Approaches and Solutions

In response to the challenges, we adopted the following strategies :

- **Modular Testing Strategy :** We began by testing individual components of the CRC system before integrating them into the full setup, which helped simplify debugging and validation.
- **Detailed Logging :** Implementing detailed logging at every stage of the CRC process allowed us to quickly pinpoint and resolve issues in the implementation.
- **Comparison with Standard Implementations :** By comparing our results with well-established CRC implementations, we ensured that our solution was mathematically accurate and identified areas for algorithmic improvement.

8 Conclusion

While CRC is highly effective for detecting errors, particularly single-bit errors, it is not equipped for error correction. The inability to uniquely detect all multiple-bit errors and the lack of a correction mechanism limit CRC's application in certain contexts where data integrity is critical and errors must be fixed automatically. Cyclic Redundancy Check (CRC) provides robust error detection capabilities. Using Java, we implemented CRC-8 to CRC-32 and evaluated their strengths in single and burst error detection. Combined with single-bit correction logic via becomes a useful tool in secure communications.