CSE 2213 – Data and Telecommunication Lab Report



Lab Report Name:

Implementation of CRC for Error Detection and Single-Bit Error Correction

*Group: B(Even)*

Submitted By

**Md. Mahmudur Rahman Moin (AE-30)**

**Md. Sadman Shihab (SH-36)**

29th Batch

Department of Computer Science & Engineering

University of Dhaka

Submitted On

25 June 2025

Table of Contents

[1. Introduction 3](#_heading=h.ustrejc3oc3l)

[1.1 N](#_heading=h.n1g92nnn65yo)ecessity of CRC [3](#_heading=h.n1g92nnn65yo)

[1.2 H](#_heading=h.ilwc6vj8zhyi)ow CRC is used to detect error and correct single-bit error 4

[2. O](#_heading=h.3v8croa465r)bjectives6

[3. A](#_heading=h.uev69kqee9ix)lgorithms6

3[.1](#_heading=h.n1g92nnn65yo) Client Side6

3[.2](#_heading=h.ilwc6vj8zhyi) Server Side7

4. Implementation 8

4[.1 C](#_heading=h.n1g92nnn65yo)lient Side8

4[.2](#_heading=h.ilwc6vj8zhyi) Server Side10

5. Result Analysis 12

5[.1 C](#_heading=h.n1g92nnn65yo)RC-812

5[.2 C](#_heading=h.ilwc6vj8zhyi)RC-10 12

5[.3 C](#_heading=h.n1g92nnn65yo)RC-1613

5[.4 C](#_heading=h.ilwc6vj8zhyi)RC-32 13

6. Discussion14

7. Learning and Difficulties 16

7[.1](#_heading=h.n1g92nnn65yo) Learning16

7[.2](#_heading=h.ilwc6vj8zhyi) Difficulties 16

8. Conclusion 17

# 

# Introduction

Cyclic Redundancy Check (CRC) is a widely used error-detecting technique in digital networks and storage devices. It helps identify accidental changes to raw data during transmission or storage. CRC is based on polynomial division and appends a sequence of redundant bits, known as the CRC code, to the message. The receiver performs the same CRC calculation and compares the result to detect errors. Blocks of data entering these systems get a short check value attached, based on the remainder of a [polynomial division](https://en.wikipedia.org/wiki/Polynomial_long_division) of their contents. On retrieval, the calculation is repeated and, in the event the check values do not match, corrective action can be taken against data corruption. CRCs can be used for [error correction.](https://en.wikipedia.org/wiki/Error_detection_and_correction)

In CRC, the check (data verification) value is a redundancy (it expands the message without adding information) and the algorithm is based on cyclic codes. CRCs are popular because they are simple to implement in binary hardware, easy to analyze mathematically, and particularly good at detecting common errors caused by noise in transmission channels. Because the check value has a fixed length, the function that generates it is occasionally used as a hash function.

## 1.1 Necessity of CRC

* Reliable Error Detection: CRC can detect common transmission errors such as single-bit errors, burst errors, and double-bit errors with high accuracy.
* Efficient Computation: CRC uses binary division, making it fast and easy to implement in hardware and software with minimal computational overhead.
* Data Integrity Assurance: It ensures that the data received is the same as the data sent, maintaining the integrity of transmitted or stored data.
* Crucial for Noisy Channels: Especially useful in communication systems where noise can corrupt data (e.g., wireless, satellite, and long-distance transmissions).
* Widely Used Standard: Adopted in protocols like Ethernet, USB, HDLC, and storage formats like ZIP files and disk drives, making it a standardized and trusted method.
* Foundation for Error Correction: Though CRC itself only detects errors, it can be paired with retransmission or correction techniques to improve overall reliability.

## 1.2 How CRC is used to detect error and correct single-bit error

### 1. CRC Error Detection – Step-by-Step

* Select a Generator Polynomial (G(x))
  + Choose a binary polynomial (e.g., for CRC-8: 100000111).
* Append Zeros to the Message
  + Add n zeros to the end of the message, where n is the degree of the generator polynomial.
* Divide the Message by the Polynomial
  + Perform binary division (modulo-2) of the extended message using XOR with the generator polynomial.
* Obtain the CRC Remainder
  + The remainder from the division is the CRC code (checksum).
* Transmit Message + CRC
  + Send the original message with the CRC appended to it.
* Receiver Repeats the Division
  + At the receiving end, the full received data (message + CRC) is divided by the same generator polynomial.
* Check the Remainder
  + If the remainder is zero, no error is detected.
  + If the remainder is non-zero, an error is detected.

### 2. CRC-Based Single-Bit Error Correction – Step-by-Step

* Perform CRC Check on Received Message
  + If the CRC remainder is zero, assume the message is error-free.
  + If not zero, proceed to locate the error.
* Flip Each Bit One-by-One
  + Copy the received message and flip the first bit.
  + Recompute the CRC for the modified message.
* Check if CRC Becomes Zero
  + If yes, the bit you flipped was the error — you’ve corrected it.
  + If no, revert the bit and try flipping the next one.
* Repeat for All Bits
  + Continue the process until you find a single-bit flip that makes the CRC remainder zero.
* Restore the Corrected Message
  + Once the error is corrected, use the modified message as the corrected data.

# 2. Objectives

The main objectives of performing this lab are:

* Understand CRC theory and its application in detecting transmission errors.
* Implement CRC encoding and decoding mechanism using Java sockets and polynomial division.
* Support multiple CRC polynomial types (CRC-8, CRC-10, CRC-16, CRC-32).
* Simulate data transmission and both single-bit and burst errors.
* Detect errors using CRC and evaluate CRC’s effectiveness in ensuring data integrity.

# 3. Algorithms

## 3.1 Client Side (Sender)

1. Create a Socket object that takes the IP address and port number as input.
2. Create an object of the DataOutputStream class, which is used to send data to the server side.
3. Open and read content from a file (e.g., input.txt). The text file (input.txt) contains with ASCII character data
4. Read data from file or input.
5. Convert ASCII characters to binary.
6. Choose CRC type and input matching generator polynomial.
7. Append (k-1) zeros to data.
8. Perform modulo-2 division on the appended data using the generator polynomial (XOR-based bitwise division).
9. Obtain the remainder, which is the CRC code.
10. Append remainder to form codeword.
11. Send codeword and generator via socket.

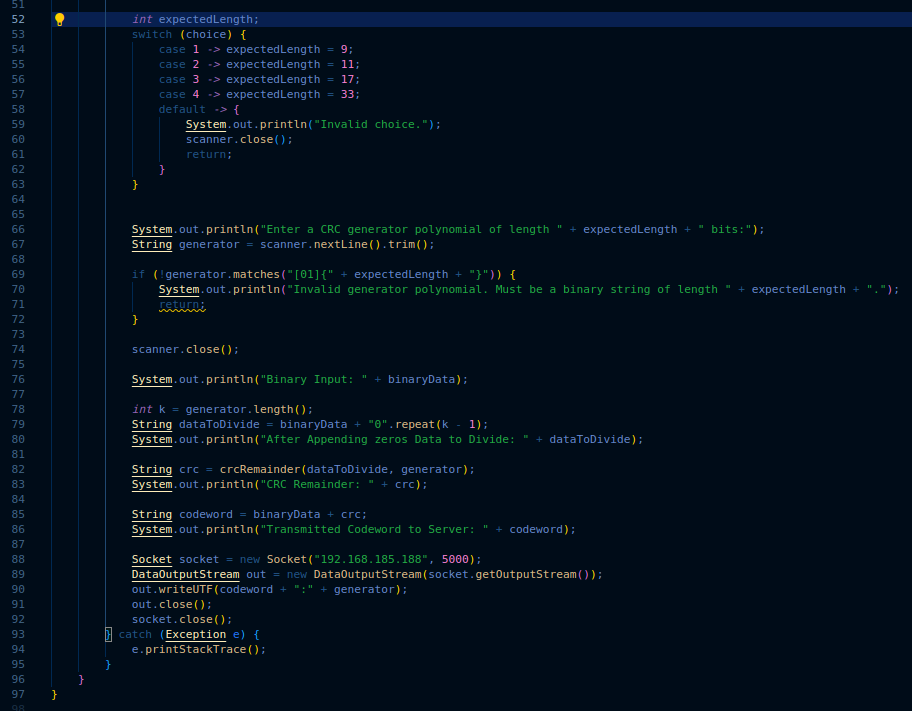
## 3.2 Server-side

1. Create a ServerSocket object, namely handshaking socket, which takes a port number as input.
2. Create a plain Socket object that accepts client requests
3. Create an object of the DataInputStream class, which is used to read data
4. Accept connection and receive codeword + generator from the input stream
5. Randomly flip one bit (simulating noise).
6. Recalculate CRC on received codeword.
7. If CRC is valid → No error.
8. Else, try flipping each bit to find correction.
9. If exactly one fix found → Corrected.
10. If zero or multiple fixes → Uncorrectable.

# 4. Implementation

## 4.1 Client Side





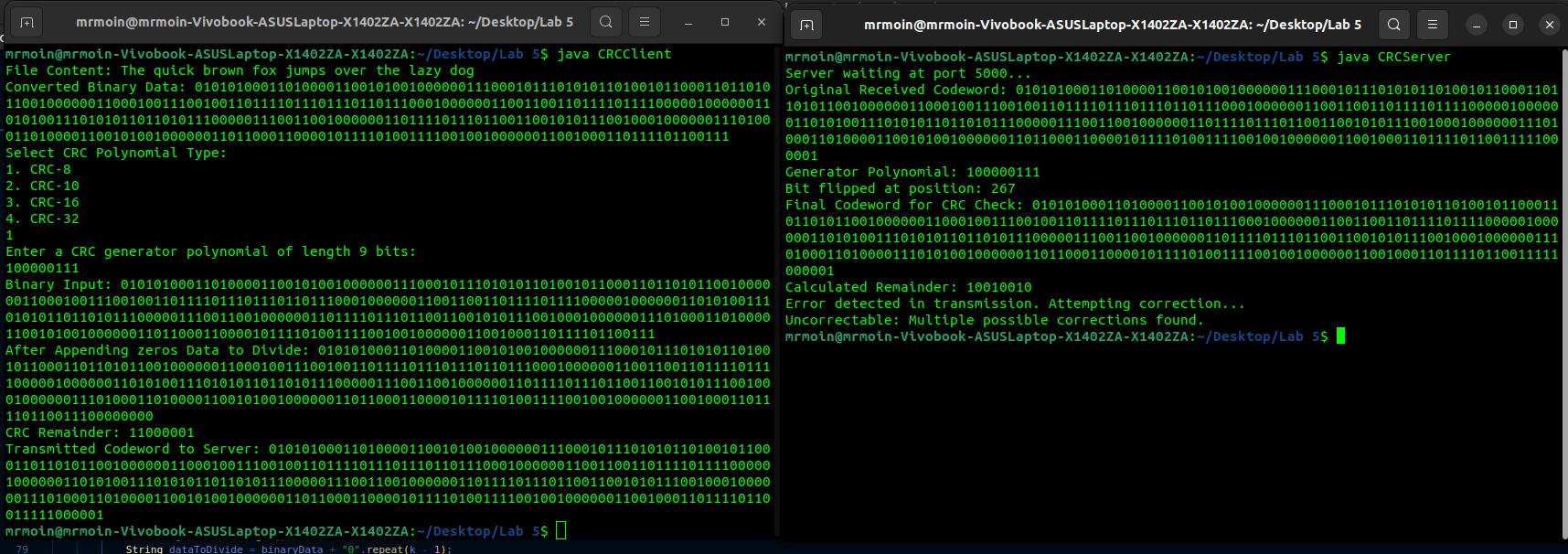
## 4.2 Server side



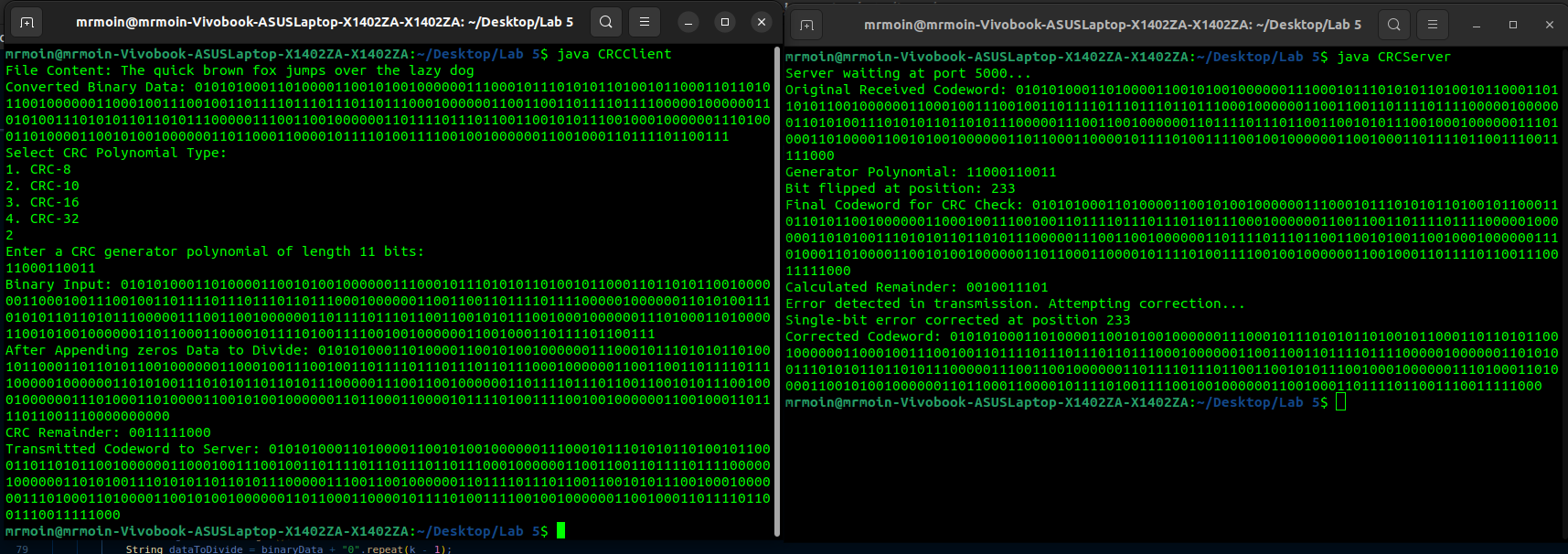
# 

# 5. Result Analysis

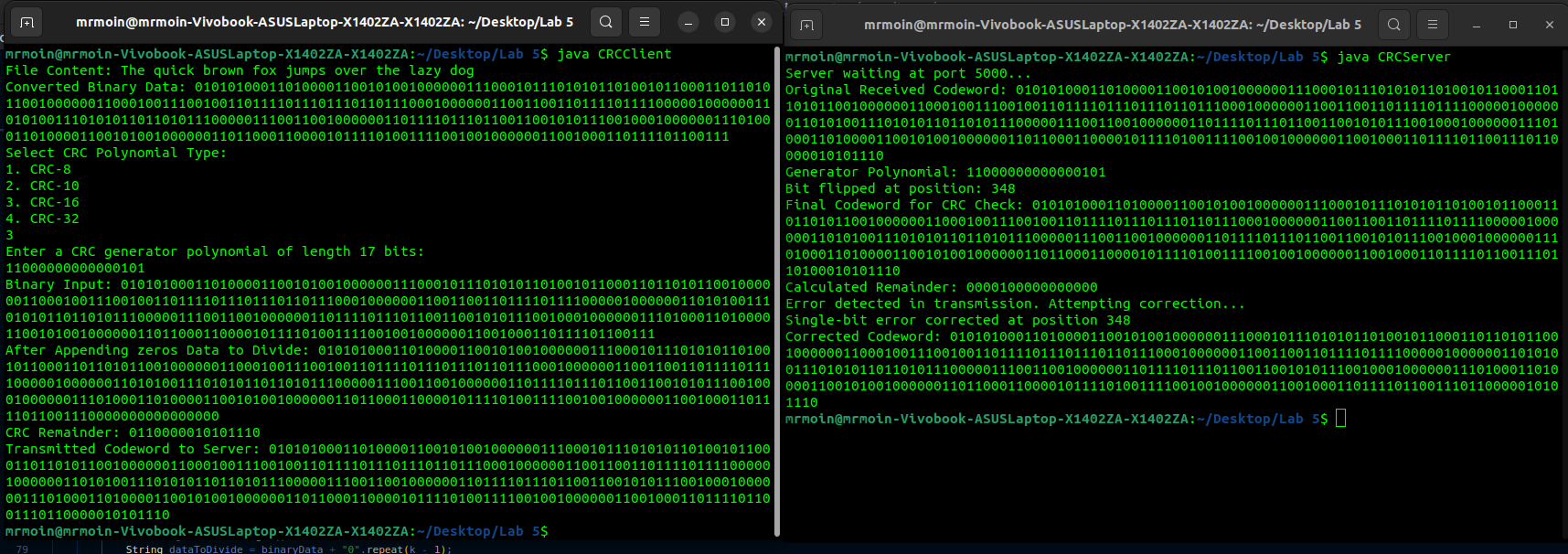
## 5.1 CRC-8

 Client Server

## 5.2 CRC-10

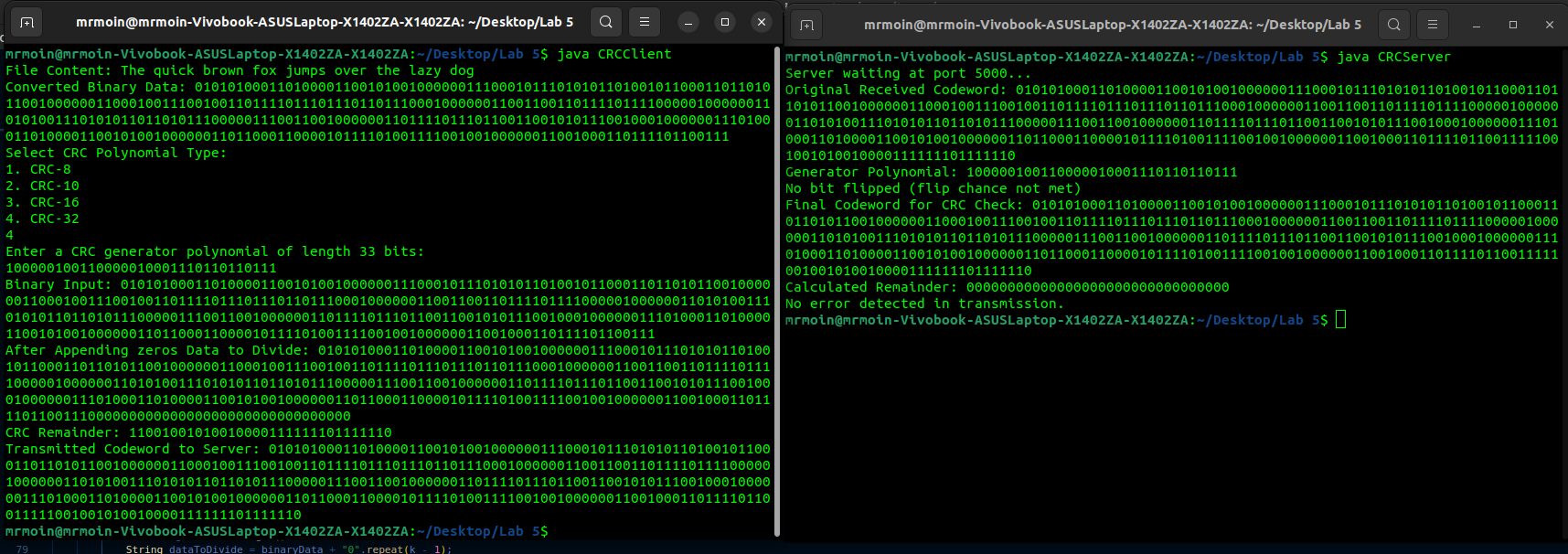
 Client Server

## 5.3 CRC-16

 Client Server

## 

## 5.4 CRC-10

 Client Server

6. Discussion

**Comparative Analysis: CRC Polynomial Generators**

| **CRC Type** | **Polynomial (Binary)** | **Degree (n)** | **Common Use Cases** | **Detection Capability** | **Correction Capability** |
| --- | --- | --- | --- | --- | --- |
| CRC-8 | 100000111 | 8 | ATM cells, SMBus, automotive | Detects 1-bit and short burst errors up to 8 bits | Can correct 1-bit errors (brute force) |
| CRC-10 | 11000110011 | 10 | Telecommunication (e.g., ITU-T) | Better burst detection than CRC-8 | Can correct 1-bit errors with effort |
| CRC-16 | 11000000000000101 | 16 | USB, Modbus, X.25, disk sectors | Detects all 1-bit, many burst errors up to 16 bits | Can correct 1-bit only |
| CRC-32 | 100000100110000010001110110110111 | 32 | Ethernet, ZIP files, PNG | Strong burst error detection up to 32 bits | Too large for brute-force correction |

## Why and How These Are Used

### 1. Single-bit Error Detection

* All standard CRC polynomials guarantee detection of single-bit errors as long as:
  + The generator polynomial has more than one non-zero term.
* Implementation:
  + Easy modulo-2 division using shift registers or XOR in code.
  + A non-zero CRC remainder upon checking indicates an error.

### 2. Burst Error Detection

* A burst error of length k affects k consecutive bits.
* A CRC of degree n can:
  + Detect all burst errors of length ≤ n.
  + Detect most burst errors > n, but not all.
* How It Works:
  + CRC polynomials are chosen to ensure minimal collision (error not detected).
  + Longer polynomials (like CRC-32) catch longer bursts.

### 3. Single-bit Error Correction

* Brute-force bit flipping:
  + Flip each bit one-by-one and recheck the CRC.
  + If a flipped version results in a zero CRC, you've found and corrected the bit.
* Only feasible for short messages and small CRCs:
  + CRC-8 and CRC-10 can be practically used this way.
  + CRC-16 and CRC-32 are computationally expensive for correction.

## **Why CRC Cannot Correct Multiple-bit (Burst) Errors**

1. No Location Information
   * CRC tells something is wrong, not where it’s wrong.
2. Ambiguity
   * Many different erroneous messages can produce same CRC.
   * Not enough redundancy to uniquely identify error patterns.
3. No Inherent Correction Logic
   * CRC is a detection code, not an error-correcting code (ECC) like Hamming or Reed-Solomon.
4. Brute-force infeasibility
   * For a k-bit message, you’d need to try 2^k combinations for all possible multiple-bit errors.

**Implementation Insights**

| **Aspect** | **Detection (Single & Burst)** | **Correction (Single-bit only)** |
| --- | --- | --- |
| Complexity | Low (XOR + shift) | Moderate (O(n) bit-flips) |
| Speed | Very fast in hardware or software | Slower due to iterative correction check |
| Feasibility for Burst | Yes (up to n bits reliably) | No (no bit location info) |
| Code Size/Overhead | Minimal | Acceptable only for short messages |

# 7. Learning and Difficulties

## 7.1 Learning

* CRC math and polynomial division in binary.
* Java socket communication.
* Error simulation and bit-level manipulation.
* Single-bit error correction via brute-force validation.

## 7.2 Difficulties

* Ensuring exact generator length for each CRC type.
* Managing ambiguous correction scenarios.
* Distinguishing between detection vs correction limits of CRC.

8. Conclusion

This experiment demonstrates how CRC can effectively detect transmission errors using polynomial division based on binary polynomials. Through practical implementation, we observed that CRC is a powerful tool for detecting transmission errors, especially when integrated into a client-server communication model. Although CRC is not inherently a correction technique, we implemented a brute-force bit-flipping strategy that enabled us to correct single-bit errors by re-validating the CRC — effectively simulating Hamming distance 1 correction.

Among the CRC types tested, CRC-32 demonstrated the highest accuracy in both detecting and correcting errors due to its longer generator polynomial, which offers greater redundancy and stronger error detection capabilities. CRC-16 also showed reliable performance, whereas CRC-8 and CRC-10, although efficient and lightweight, struggled with ambiguity when correcting single-bit errors in longer messages.

Ultimately, this lab reinforced the importance of selecting the appropriate CRC polynomial for a given communication context, balancing computational cost with error-detection reliability. It highlighted CRC's robustness in detecting even burst errors, while clarifying its limitations in reliably correcting them without additional error-correcting codes. The hands-on implementation and testing provided valuable insight into real-world error detection systems and the critical role CRC plays in maintaining data integrity.