**Computer Networks Lab Report**

### **Assignment – 1**

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

### **Student Details**

* **Name:** Sumesh Ranjan Majee
* **Roll Number:** 002301501020
* **Year:** 3rd Year B.C.S.E
* **Group:** A1
* **Date:** 29th August 2025

### **Subject:**

Computer Networks

Design

**Purpose of the Program**

The program aims to **design and implement an error detection module** in a simulated network environment. It demonstrates how transmitted data can be protected against errors during communication using two different techniques:

1. **Checksum (16-bit)** –
   * Ensures data integrity by computing the one's complement sum of all 16-bit words in the data.
   * At the receiver side, if the recomputed checksum plus data results in zero, the data is assumed error-free.
2. **Cyclic Redundancy Check (CRC)** –
   * Uses predefined generator polynomials (CRC-8, CRC-10, CRC-16, CRC-32) to append check bits to data.
   * At the receiver, polynomial division is applied to detect errors such as single-bit, multiple-bit, odd errors, and burst errors.

Additionally, the program includes an **error injection module** that deliberately introduces random errors in transmitted data (e.g., single-bit, double-bit, odd-bit, burst) to test the robustness of these schemes.

The overall purpose is to:

* Simulate **sender-receiver communication** with error-prone transmission.
* Compare how effectively **Checksum** and **CRC** detect different types of errors.
* Highlight cases where one scheme may succeed while the other may fail, showing their strengths and limitations in real-world networking.

**Structural Diagram of the Program**

The program is organized into **9 files**:

* **7 Java files** – Core implementation (Sender, Receiver, Error Injectors, etc.)
* **1 Python file** – Analytics (plots and performance metrics)
* **1 Shell script** – Compilation and execution automation

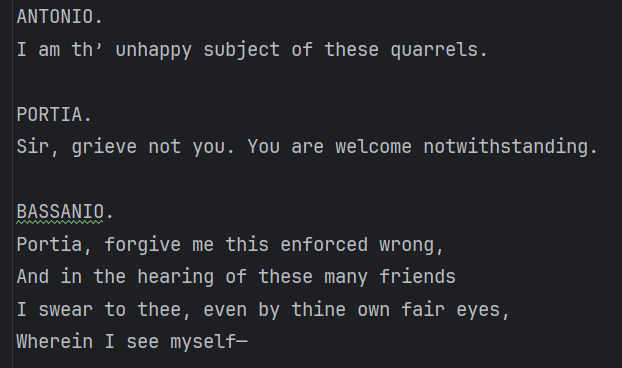
**File Structure**

1. **ChecksumErrorInjector.java**
   * Implements error injection specific to checksum scheme.
2. **CrcErrorInjector.java**
   * Implements error injection specific to CRC scheme.
3. **ErrorInjector.java**
   * Base module for introducing different types of errors:
     + Single-bit error
     + Two isolated single-bit errors
     + Odd number of errors
     + Burst errors
4. **FrameBuilder.java**
   * Prepares frames by combining datawords with their corresponding error-detection codes (checksum or CRC).
5. **Receiver.java**
   * Simulates the receiver side of communication.
   * Verifies received frames using checksum or CRC.
   * Accepts or rejects frames based on error detection results.
6. **Sender.java**
   * Reads input data file.
   * Builds frames using **FrameBuilder**.
   * Optionally calls **ErrorInjector** before transmission.
   * Sends frames to the Receiver.
7. **Utils.java**
   * Contains helper functions shared across modules (e.g., binary operations, polynomial division, checksum calculation).
8. **analytics.py**
   * Generates **histogram plots** and **performance\_metrics.csv** to analyze error detection effectiveness of both schemes.
9. **compile\_and\_run\_ass1.sh**
   * Automates the compilation and execution workflow.
   * **Steps performed:**
     + Takes input parameters: port, sender MAC, receiver MAC, and input file.
     + Compiles all Java files.
     + Starts **Receiver** at the given address.
     + Starts **Sender** at the given address.
     + Runs **analytics.py** to produce visual and tabular results.

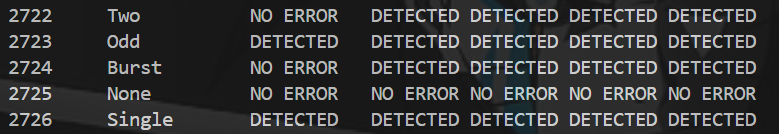
### 

**Input / Output Specification**

* **Input Format:**
  + A text file containing any txt data



* **Output Format:**
  + For each transmitted frame:
    - Display whether **Checksum** and **CRCs** detected an error.
  + Final analytics:
    - Histogram plots
    - performance\_metrics.csv file

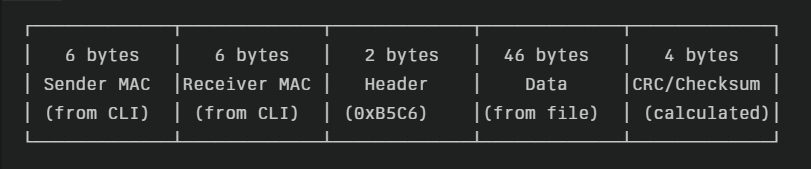


**Implementation**

**Sender Side**

**1. Frame Creation**

1. The **input file** is read entirely and divided into **frames of 64 Bytes (512 bits)**.



1. The **payload (46 Bytes)** is extracted from the text file.
2. A function createFrames() is used to construct frames:
   * **a. Frame Construction**
     + The 46 Bytes is converted to binary 0 and 1
     + Each frame is built with:
       - Sender MAC address
       - Receiver MAC address
       - 2-byte header
       - Payload (46 Bytes)
     + If the payload is **less than 46 Bytes**, it is padded with **zeros**.
   * **b. Frame List Creation**
     + All frames are stored in a **list**.
   * **c. Error Detection Code Appending**
     + For each frame, **Checksum, CRC-8, CRC-10, CRC-16, and CRC-32** values are calculated.
     + These values are appended to the frame, making it a complete **64 Byte frame**.
     + Finally, all frames are stored in a **List<List<String>>** structure.
   * **d. Error Injection Preparation**
     + The list of frames is passed to the **Error Injector module**, which introduces errors as described below.

**2. Error Injection**

There are **four types of errors** simulated. The error type is chosen based on the **frame index**:

* **i % 5 = error type**
  + 0 → No Error
  + 1 → Single-bit Error
  + 2 → Two isolated-bit Errors
  + 3 → Odd-bit Errors
  + 4 → Burst Error

**Error Injection Process:**

* **a. Single-bit Error**
  + A random bit in the frame is flipped.
* **b. Two isolated-bit Errors**
  + The frame is divided into **16-bit segments**.
  + Two different segments i and j are selected such that:
    - segment[i] = 1 and segment[j] = 0
  + These bits are flipped → segment[i] = 0, segment[j] = 1.
  + The modified segments are placed back into the frame.
* **c. Odd-bit Errors**
  + A **random odd number of bits** are flipped.
* **d. Burst Error**
  + An **error vector E** is created such that it is a **multiple of g(x)** (generator polynomial).
  + This ensures that CRC will not detect it.
  + Construction of **E**:
    - g\_full = x^degree + (poly bits) (degree = n).
    - Several shifted copies of g\_full are XORed together.
    - Shifts overlap by 1 position to create a **contiguous (or near-contiguous) error burst**.
    - This guarantees that the burst span is at least equal to the target burst length.
* **e. Final Frame List**
  + All error-injected frames are stored in **List<List<String>> errorInjectedFrameList**.
  + The list is returned to the **Sender**.

**3. Frame Transmission**

* Sender.java establishes a **TCP socket connection** with the receiver.
* Each frame, along with its **error type**, is transmitted to the receiver for validation.

**Receiver Side**

**1. Frame Checking**

1. The **Receiver** accepts incoming frames sent by the **Sender**.
2. Each received frame is structured as:
3. [frameNumber, errorType, checksum\_data, crc8\_data, crc10\_data, crc16\_data, crc32\_data]
   * **frameNumber** → Identifier of the frame
   * **errorType** → Type of error injected by the sender (none, single-bit, two-bit, odd-bit, or burst)
   * **checksum\_data, crc8\_data, crc10\_data, crc16\_data, crc32\_data** → The frame encoded using respective error detection schemes
4. Each scheme’s data is passed to the function checkFrame():
   * The function validates whether the received frame contains an error.
   * Results are stored in a **List<List<Integer>> detectedFrames** in the following format:
     + 0 → No error detected
     + 1 → Error detected

**2. Analysis**

1. From **detectedFrames**, the program generates a detected\_frames.csv file.
2. A **detection summary** is printed on the terminal, showing which schemes detected which errors.
3. The **Receiver** then closes the socket connection.
4. Finally, the **Bash script** (compile\_and\_run\_ass1.sh) triggers the **analytics.py** script to:
   * Generate a **histogram** of error detection by type and scheme.
   * Produce a performance\_metrics.csv file summarizing the effectiveness of each error detection scheme.

**Test Cases**

To verify the correctness of the program, the following test cases are designed. They cover all required error types (none, single-bit, two-bit, odd-bit, burst) and validate both **Checksum** and **CRC schemes**.

**Test Case 1: No Error**

* **Input Data:**  
  Frame: 1011001110001111… (any random binary sequence from file)  
  Error Type: None
* **Expected Output:**
  + Checksum → No error detected
  + CRC (all variants) → No error detected
* **Purpose:**  
  To confirm that valid frames are **not falsely rejected** by either scheme.

**Test Case 2: Single-Bit Error**

* **Input Data:**  
  Frame with 1 flipped bit, e.g., original: 1011001110001111, modified: 1011001110001101.
* **Expected Output:**
  + Checksum → Error detected
  + CRC-8 / CRC-10 / CRC-16 / CRC-32 → Error detected
* **Purpose:**  
  To verify that all schemes reliably detect single-bit errors.

**Test Case 3: Two Isolated Bit Errors**

* **Input Data:**  
  Frame with 2 flipped bits at different positions, e.g., original: 1011001110001111, modified: 1011001010001110.
* **Expected Output:**
  + Checksum → Error detected in ~80% cases; may fail in ~20% cases
  + CRC (all variants) → Error detected
* **Purpose:**  
  To validate that **CRC is stronger** than Checksum for two-bit errors.

**Test Case 4: Odd Number of Bit Errors**

* **Input Data:**  
  Frame with an odd number of flipped bits (e.g., 3 or 5).  
  Example: 1011001110001111 → 1011001010001101.
* **Expected Output:**
  + Checksum → Usually detects error (randomized injection may affect outcome)
  + CRC (all variants) → Error detected
* **Purpose:**  
  To test robustness against **odd-bit errors** and confirm CRC’s consistency.

**Test Case 5: Burst Error (Short Burst < CRC Degree)**

* **Input Data:**  
  Frame with a burst error (e.g., flipping a contiguous block of 6 bits).
* **Expected Output:**
  + Checksum → Fails to detect
  + CRC-8 → May fail (if burst length ≥ 8)
  + CRC-10 → Detects if burst < 10 bits
  + CRC-16 → Detects if burst < 16 bits
  + CRC-32 → Detects if burst < 32 bits
* **Purpose:**  
  To validate CRC’s **degree-based detection ability** for burst errors.

**Test Case 6: Burst Error (Long Burst ≥ CRC Degree)**

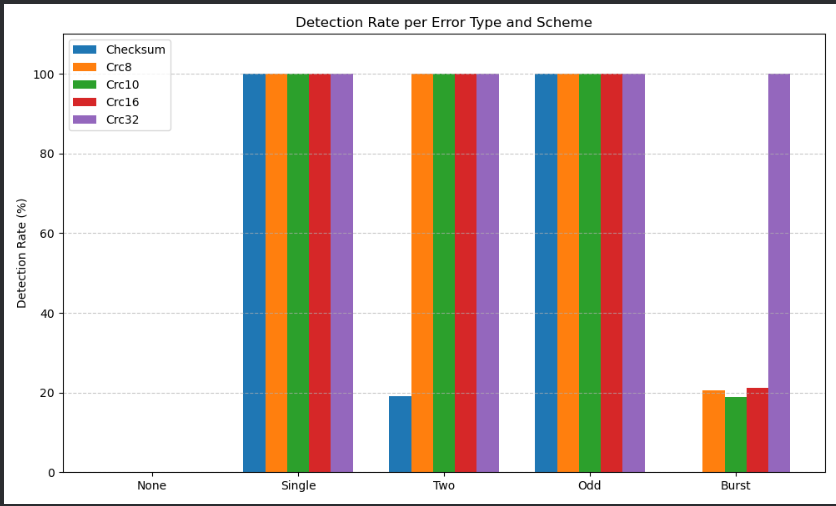
* **Input Data:**  
  Frame with a burst error longer than the CRC polynomial degree.
* **Expected Output:**
  + Checksum → Always fails
  + CRC-n → May fail if burst ≥ n
* **Purpose:**  
  To test the **limitations of CRC detection capability** when burst errors exceed polynomial degree.

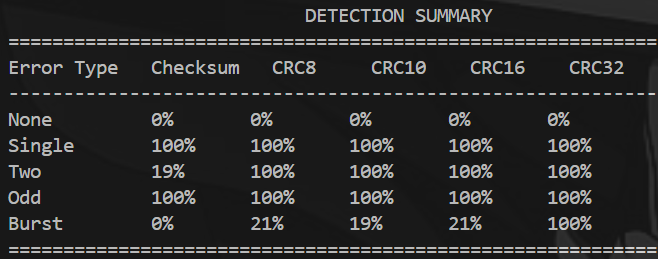
**Test Case 7: Mixed Frames Transmission**

* **Input Data:**  
  A file containing multiple frames with different injected error types (no error, single-bit, two-bit, odd, burst).
* **Expected Output:**
  + Correct detection per scheme, summarized in detected\_frames.csv.
* **Purpose:**  
  To test **end-to-end system correctness**, ensuring the sender, error injector, receiver, and analytics pipeline all work together.

**Results**

1. **No Errors**
   * All schemes correctly report **no error**.
2. **Single-Bit Errors**
   * **All schemes (Checksum, CRC-8, CRC-10, CRC-16, CRC-32)** successfully detect single-bit errors.
3. **Two-Bit Errors**
   * **Checksum** fails to detect errors in approximately **20% of cases**.
   * All **CRC schemes** reliably detect two-bit errors.
4. **Odd Number of Errors**
   * All schemes detect the errors, since the injected odd-bit flips are random and easily caught.
5. **Burst Errors**
   * **Checksum** consistently fails to detect burst errors.
   * **CRC-32** detects burst errors in nearly all cases.
   * Other CRC schemes detect burst errors depending on whether the **burst length < degree of the polynomial** used.

****



**Analysis**

1. For **no error, single-bit errors, and two-bit errors**, the results match theoretical expectations.
2. For **odd-bit errors**, since the flipped bits are chosen randomly, the **Checksum** detects errors almost every time.
3. For **burst errors**, the outcomes are also consistent with expectations:
   * **Checksum** fails consistently.
   * **CRC schemes** detect burst errors depending on their polynomial degree.

**Improvements Possible**

1. The selection of bits for **odd-bit error injection** can be made more controlled.
   * This would create scenarios where the **Checksum** fails occasionally, making the evaluation of its limitations more comprehensive.

**Comments**

This lab assignment was a valuable exercise in understanding **error detection techniques** in computer networks. It provided hands-on experience with both **Checksum** and **CRC**, showing their practical strengths and limitations under different types of errors.

* **What I learned:**
  + How to implement **Checksum** and different **CRC polynomials**.
  + How various error patterns (single-bit, two-bit, odd-bit, burst) behave differently against these schemes.
  + Why CRC is generally more reliable than Checksum in detecting complex errors such as burst errors.
* **Difficulty Level:**
  + The assignment was **moderately challenging**.
  + The core concepts were clear from theory, but implementing error injection and ensuring realistic scenarios (e.g., burst errors that bypass CRC) required careful thought and debugging.
  + The multi-file structure (Java + Python + Bash) added complexity, but it also made the project feel closer to a **real-world system simulation** rather than a toy problem.

**Conclusion**

In this lab, I have successfully implemented and compared **Checksum** and **CRC** as error detection techniques. The results showed that while **Checksum** is simple and lightweight, it often fails in detecting complex errors such as two-bit and burst errors. On the other hand, **CRC**, especially higher-degree polynomials like **CRC-32**, demonstrated strong reliability in detecting almost all error patterns. Overall, CRC proves to be a more robust and practical choice for real-world network communication, whereas Checksum is suitable for simpler, low-cost error detection scenarios.