PERFORMANCE COMPARISON OF ERROR-CORRECTING CODES

Project Report submitted in partial fulfillment of the

requirement for the degree of

Bachelor of Technology

in

Computer Science & Engineering

under the Supervision of

Mr. Amit Kumar Singh

By

Saurabh Kumar, 111231

To

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "**Performance comparison of error-correcting codes**", submitted by **Saurabh Kumar** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Supervisor's Name:

Designation:

Acknowledgement

Many people have contributed to the success of this. Although a single sentence hardly suffices, I would like to thank God for blessing me with His grace. I am profoundly indebted to my guide **Mr. Amit Kumar Singh** for innumerable acts of timely advice, encouragement and I sincerely express my gratitude to him.

I express my immense pleasure and thankfulness to the **Prof. Dr. S.P. Ghrera (HOD)** and all the teachers and staff of the Department of CSE & IT, Jaypee University of Information & Technology, for their cooperation and support.

Date:

Name of the student:

Saurabh Kumar

111231

Table of Content

List of Figures

List of Tables

Problem Statement

There are many reasons for need of Error Correcting Codes such as noise, cross-talk etc., which may lead data to get corrupted during transmission. The upper layers work on some generalized view of network architecture and are not aware of actual hardware data processing. So, upper layers expect error-free transmission between systems. Most of the applications would not function expectedly if they receive erroneous data. Applications such as voice and video may not be that affected and with some errors they may still function well.

Data-link layer uses some error control mechanism to ensure that codeword (data bit streams) are transmitted with certain level of accuracy. But to understand how errors is controlled, it is essential to know what types of errors may occur and what types of correcting techniques to be used.

An error during data transmission has been a serious problem facing over the last few decades. So, our aim is to reduce the error rates by combine two methods (Hamming and Repetition code) of error correcting codes, which improves the performance of the proposed method.

Abstract

Environmental interference and physical defects in the communication medium can cause random bit errors during data transmission. Error coding is a method of detecting and correcting these errors to ensure information is transferred intact from its source to its destination. Error coding is used for fault tolerant computing in computer memory, magnetic and optical data storage media, satellite and deep space communications, network communications, cellular telephone networks, and almost any other form of digital data communication. Error coding uses mathematical formulas to encode data bits at the source into longer bit words for transmission. The "code word" can then be decoded at the destination to retrieve the information. The extra bits in the code word provide redundancy that, according to the coding scheme used, will allow the destination to use the decoding process to determine if the communication medium introduced errors and in some cases correct them so that the data need not be retransmitted. Different error coding schemes are chosen depending on the types of errors expected, the communication medium's expected error rate, and whether or not data retransmission is possible. Faster processors and better communications technology make more complex coding schemes, with better error detecting and correcting capabilities, possible for smaller embedded systems, allowing for more robust communications. However, tradeoffs between bandwidth and coding overhead, coding complexity and allowable coding delay between transmissions, must be considered for each application.

During the transmission of data from transmitter to receiver, there is loss of information in the communication channel due to noise. This loss is measured in terms of bit error rate (BER) and several decoding algorithms and modulation techniques used to minimize it. Some of the error-correcting codes are Hamming

code, Repetition code, BCH code [4], Reed Solomon code and Hybrid code. Hybrid Code[6] are one of the most powerful types of error control codes currently available, which could achieve low BERs at signal to noise ratio (SNR) very close to Shannon limit. Nevertheless, the specific performance of the code highly depends on the particular decoding algorithm used at the receiver.

1. INTRODUCTION

1.1 Overview

In information theory and coding theory with applications in computer science and telecommunication, error detection and correction or error control are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to receiver. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data in many cases.

Forward error correction (FEC) [4] is the method of transmitting error correction information along with the message. At the receiver, this error correction information is used to correct any bit-errors that may have occurred during the transmission. The improved performance comes at the cost of introducing a considerable amount of redundancy in the transmitted code. There are various FEC codes in use today for purpose of error correction. Most codes fall into either of two major categories: block codes [4] and convolutional codes [4].

Block codes work with fixed length blocks of code. Convolutional codes deal with data sequentially (i.e. taken a few bits at a time) with the output depending on both the present input as well as previous inputs.

The error correcting codes is often designed first with the goal of minimizing the gap from Shannon Capacity and attaining the target error probability.

1.2 Noisy Communications

Noise in a communications channel can cause errors in the transmission of binary digits.

- Transmit: 1 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 …
- Receive: 1 1 0 1 1 0 1 0 0 0 1 0 0 0 0 1 0 …

• For some types of information, errors can be detected and corrected but not in others.

Example: Transmit: Come to my house at 17:25 …

Receive: Come to my house at 14:25 …

1.3 Making Digits Redundant

- In binary error correcting codes, only certain binary sequences (called code words) are transmitted.
- This is similar to having a dictionary of allowable words.
- After transmission over a noisy channel, we can check to see if the received binary sequence is in the dictionary of code words and if not, choose the codeword most similar to what was received.

1.4 Binary Error Correcting Codes

- 2k equally likely messages can be represented by k binary digits.
- If these k digits are not coded, an error in one or more of the k binary digits will result in the wrong message being received.
- Error correcting codes is a technique where by more than the minimum number of binary digits are used to represent the messages.
- The aim of the extra digits, called redundant or parity digits, is to detect and hopefully correct any errors that occurred in transmission.

1.5 Types of ECC

- **Binary Codes**

  Encoder and decoder works on a bit basis.

- **Non-binary Codes**

  -Encoder and decoder works on a byte or symbol basis.

  -Bytes usually are 8 bits but can be any number of bits.

  -Galois field arithmetic is used.

-Example is a Reed Solomon Code.

-More generally, we can have codes where the number of symbols is a prime or a power of a prime.

1.6 Types of Binary codes

There are two types of Binary codes:

- Block Codes

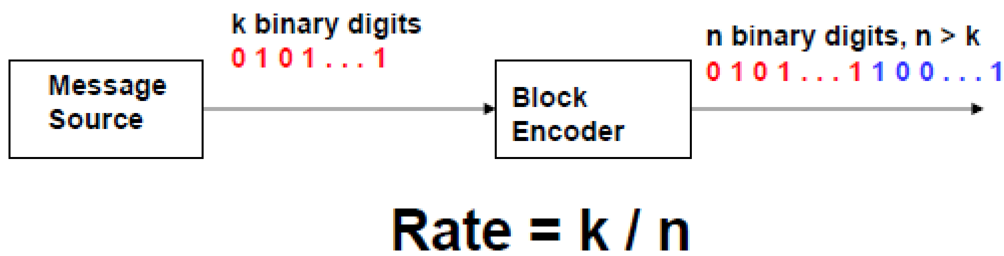


Fig 1:Block codes representation

- Convolutional Codes

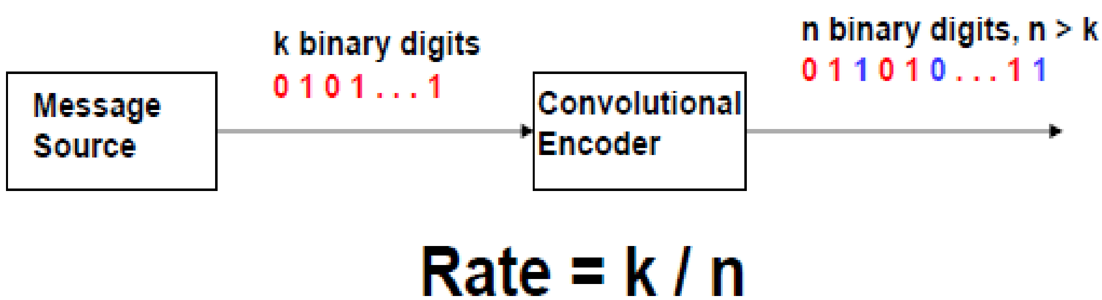


Fig 2:Convolutional codes representation

1.7 Types of Error Correcting Codes

1.7.1 Hamming Code

A Hamming Code [6] can be used to detect and correct one-bit change in an encoded code word. This approach can be useful as a change in a single bit is more than a change in two bits or more bits.

HAMMING BINARY BLOCK CODE WITH k=4 AND n=7

-In general, a block code with k information digits and block length n is called an (n,k) code.

-Thus, this example is called an (7,4) code.

-This is a very special example where we use pictures to explain the code. Other codes are NOT explainable in this way.

- All that we need to know is modulo 2 addition, $\oplus$:

$$0 \oplus 0 = 0,\ 1 \oplus 0 = 1,\ 0 \oplus 1 = 1,\ 1 \oplus 1 = 0.$$

- Message digits: C1 C2 C3 C4

- Code word C1 C2 C3 C4 C5 C6 C7

- Parity Check Equations:

$$C1 \oplus C2 \oplus C3 \oplus C5 = 0$$

$$C1 \oplus C3 \oplus C4 \oplus C6 = 0$$

$$C1 \oplus C2 \oplus C4 \oplus C7 = 0$$

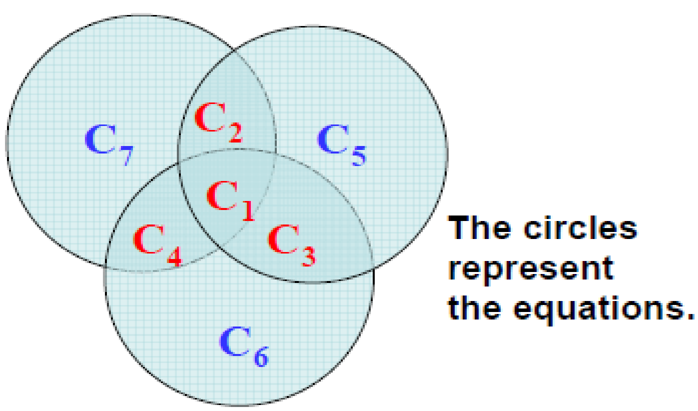


Fig 3: parity check equations

- Parity Check Matrix:

1 1 1 0 1 0 0

1 0 1 1 0 1 0

1 1 0 1 0 0 1

There is an even number of 1's in each circle.

HAMMING (7,4) CODE: ENCODING

- Message: (C1 C2 C3 C4) = (0 1 1 0)

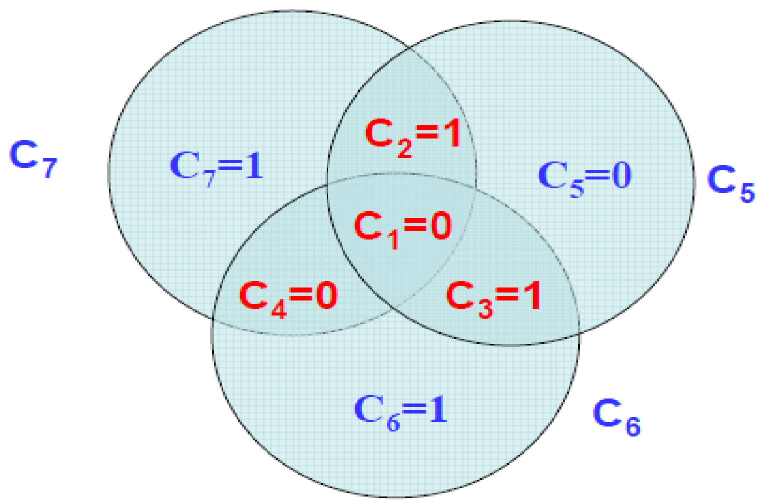


Fig 4:Hamming code encoding

- Resultant code word: 0 1 1 0 0 1 1

HAMMING (7,4) CODE: DECODING

- Transmitted code word: 0 1 1 0 0 1 1
- Example 1: Received block with one error in a message bit.

0 1 0 0 0 1 1

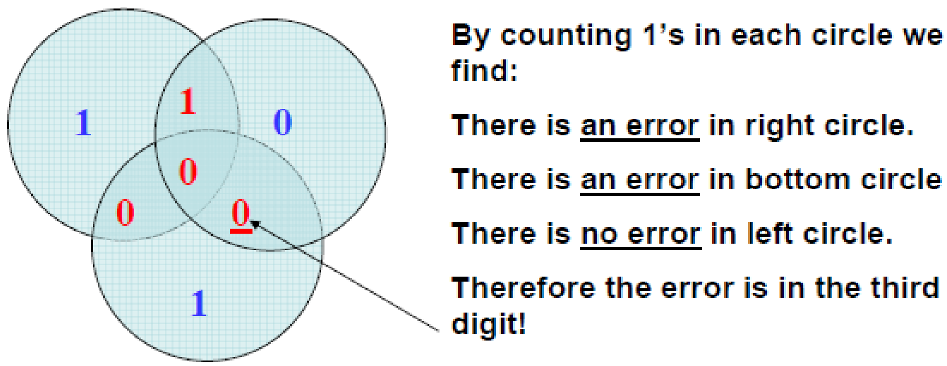


Fig 5: Hamming code decoding

HAMMING (7,4) CODE: DECODING

- Transmitted code word: 0 1 1 0 0 1 1
- Example 2: Received block with two errors:

1 1 1 0 0 0 1

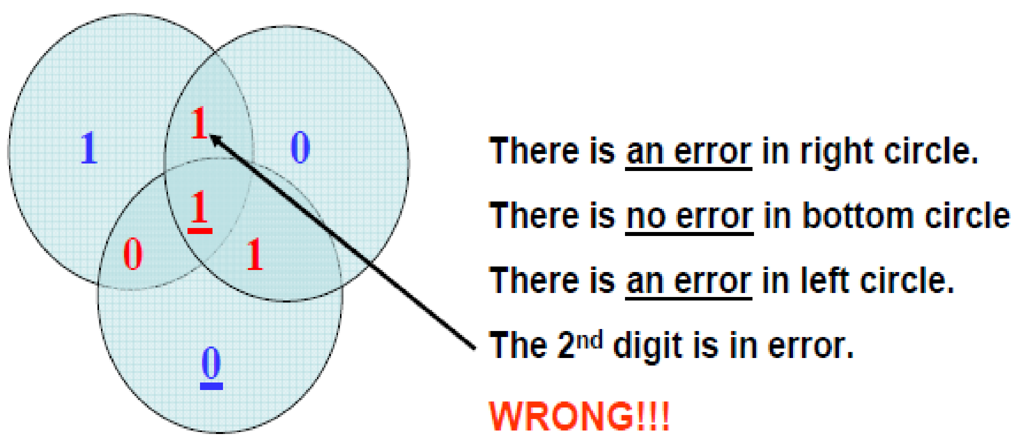


Fig 6: Hamming code decoding of two errors.

1.7.2 BCH Code (Bose, Chaudhuri and Hocquenghem)

How do we modify a Hamming code to correct two errors? In other words, how can we increase its minimum distance from 3 to 5? We will either have to lengthen the code words or eliminate some of them from our code.

Correcting two errors in a long word may not be much better than correcting one error in a short one. So we will try to produce a double error correcting sub code of the Hamming code by removing some code words to make a new code.BCH codes is a generalization of hamming codes for multiple error correction. Binary BCH codes were first discovered by A. Hocquenghem in 1959 and independently by R.C. Bose and D.K. Ray-Chaudhuri in 1960.

1.7.2.1 Introduction

-BCH[4] codes are cyclic codes. Only the codes, not the decoding algorithms, were discovered by these early writers.

-The original applications of BCH codes were restricted to binary codes of length $2^m - 1$for some integer m. These were extended later by Gorenstein and Zeiler(1961) to the nonbinary codes with symbols from Galois Field GF(q).

-The first decoding algorithm for binary BCH codes was devised by Peterson in 1960. Since then peterson's algorithm has been revised by Berlekamp, Massey, Forney and many others.

1.7.2.2 Primitive BCH Codes

-For any integer $m \geq 3$and $t < 2^{m-1}$there exists a primitive BCH code with the following parameters:

- block length: $n = 2^m - 1$
- parity check bits: $n - k \leq m*t$
- minimum distance: $d_{min} \geq 2t+1$

-This code can correct t or fewer random errors over a span of $2^m - 1$

bit positions.

This code is a t-error correcting BCH code.

-For example, for m=5 and t=2

$n=2^5 -1=31$

mt=5×2=10

$n - k \leq m*t=10$

$d_{min} \geq 2(2) + 1 \geq 5$

This is a BCH(31,21) error correcting code.

1.7.2.3 Generator Polynomial of Binary BCH Codes

-Let α be a primitive element in GF(2^m).

-The generator polynomial of the BCH code is defined as the least common multiple g(x) = lcm(m1(x),…,md − 1(x)).

-Note that the degree of g(x) is mt or less.

Hence the number of parity-check bits ;n-k, of the code is at most mt.

-Note that the generator polynomial of the binary BCH code is originally found to be the least common multiple of the minimum polynomials $\varphi_1, \varphi_2, L, \varphi_{2t.}$

i.e. $g(x)=\text{LCM}\ \{\varphi_1(x), \varphi_2(x), \varphi_3(x), L, \varphi_{2t-1}(x), \varphi_{2t}(x)\}$

However, generally, every even power of α in GF(2^m) has the

same minimal polynomial as some preceding odd power of α in

GF(2^{2m})**.**

As a consequence, the generator polynomial of the t-error correcting

binary BCH code can be reduced to

$$g(x)=\text{LCM}\{\varphi_1(x), \varphi_3(x), L, \varphi_{2t-1}(x)\}.$$

Example:- m=4,t=3

Let $\propto$ be a primitive element in GF(2^4) which is constructed

based on the primitive polynomial $p(x)= 1+x+x^4$

$$g(x) = \text{LCM}\{\varphi_1(x), \varphi_3(x), \varphi_5(x)\}$$

$$= \varphi_1(x)\varphi_3(x)\varphi_5(x)$$

$$= 1+x+x^2+x^4+x^5+x^8+x^{10}$$

This code is a (15, 5) BCH cyclic code.

1.7.2.4 Properties of Binary BCH codes

- Consider a t-error correcting BCH code of length $n = 2^m - 1$ with generator polynomial g(x).
- g(x) has a $\propto, \propto^2, \propto^3, L, \propto^{2t}$

$g(\propto^{i})=0$ for $1\leq i \leq 2t$

- Since a code polynomial c(x) is a multiple of g(x), c(x) also has $\propto,\propto^{2},\propto^{3},\mathbf{L},\propto^{2t}$ as roots ,i.e. $c(\propto^{i})=0$ for $1\leq i \leq 2t$.
- A polynomial of degree less than 2^{m} - 1is a code polynomial if and, only if it has $\propto,\propto^{2},\propto^{3},\mathbf{L},\propto^{2t}$ as roots.

1.7.3 REED-SOLOMON CODE

1.7.3.1 History

In coding theory, Reed-Solomon(RS)[6] codes are non-binary cyclic error-correcting codes invented by Irving S.Reed and Gustavo Solomon in 1960.They described a systematic way of building codes that could detect and correct multiple random symbol errors.

1.7.3.2 Introduction

Reed-Solomon codes [6] are examples of error correcting codes, in which redundant information is added to data so that it can be recovered reliably despite errors in transmission or storage and retrieval. The error correction system used on CD's and DVD's is based on a Reed-Solomon code. These codes are also used on satellite links and other communications systems.

By adding t check symbols to the data, an RS code can detect any combination of up to **t** erroneous symbols, or correct up to $\lfloor t/2 \rfloor$ symbols. RS codes are suitable as multiple-burst bit-error correcting codes, since a sequence of b + 1 consecutive bit errors can affect at most two symbols of size b.

The choice of t is up to the designer of the code, and may be selected within wide limits.

The RS decoder corrects the entire symbol, whether the error was caused by one bit being corrupted or by all of the bits being corrupted.

Thus, if a symbol is wrong, it might as well be wrong in all of its bit positions. This gives RS codes tremendous burst-noise advantages over binary codes. Burst-noise is relatively common in wireless communication due to fading.

The code minimum distance for RS code is given by

$$d_{min} = n - k + 1$$

where k is now the number of data symbols being encoded, and n is

the length of the codeword.

The code can correct up to t symbol errors, where t is given by

$$t = \frac{n-k}{2}$$

This equation shows that a codeword needs 2t parity symbols to

correct t errors.

1.7.3.3 Advantages

- RS codes can be used for long block lengths with less decoding time than other codes because RS codes work with symbol-based arithmetic.
- It provides better throughput.

1.7.3.4 Applications

- Used in the Voyager spacecraft
- They are currently used in the compact disc player
- Specific applications for digital audio, data transfer over mobile radio, satellite communications, spread spectrum systems.

1.7.4 REPETITION CODE

Introduction

- In coding theory, the repetition code [8] is one of the most basic error-correcting codes. In order to transmit a message over a noisy channel that may corrupt the transmission in a few places, the idea of the repetition code is to just repeat the message several times.

- The repetition code [8] is generally a very naive method of encoding data across a channel, and it is not preferred for Additive White Gaussian Noise Channels (AWGN), due to its worse-than-the-present error performance.
- The chief attraction of the repetition code[8] is the ease of implementation.

Repetition Coder

The encoder is a simple device that repeats, r times, a particular bit to the waveform modulator when the bit is received from the source stream.

For example, if we have a (3,1) repetition code[8], then encoding the signal m= 101001 yields a code $c = 111000111000000111$

Repetition Decoder

Repetition decoding is usually done using Majority logic detection. To determine the value of a particular bit, we look at the received copies of the bit in the stream and choose the value that occurs more frequently.

For example, suppose we have a (3, 1) repetition code [8] and we decoded the signal c=11000111. The decoded message is m= 101, as we have most occurrence of 1's (two to one), 0's (two to one), and 1's (three to zero) in the first, second, and third code sequences, respectively.

How much does this Repetition code [8] improve Reliability?

Repeating each bit three times allows us to correct one error in group of three bits, but not more errors.

Suppose each bit has probability P of being received correctly, independently of each bit.

The probability that a group of three repeated bits will be decoded Correctly is:

Pr(0 errors) + Pr(1 error) = P^3 + $3P^2(1 – P)$

Here is a plot of this vs P:

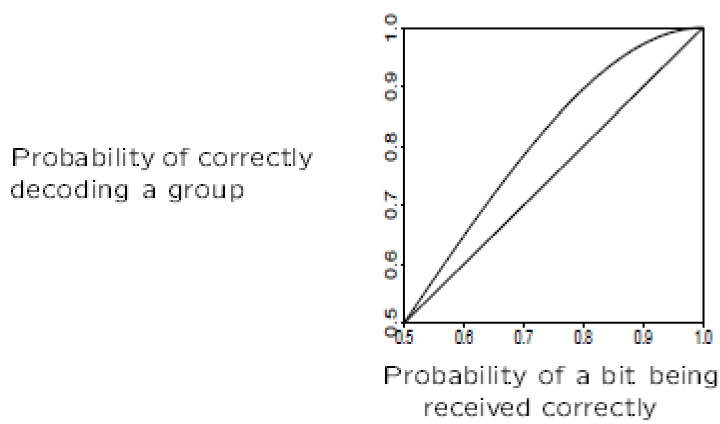


Fig 7: Performance of Repetition code

Applications

- Due to the simplicity of the channel encoding and decoding for repetition codes, they find applications in fading channels and non-AWGN environments. Repetition codes [8] can be viewed as a method of space-time diversity as well.
- Some UARTs, such as the ones used in the FlexRay protocol, use a majority filter to ignore brief noise spikes. This spike-rejection filter can be seen as a kind of a repetition decoder.

1.7.5 Comparison of various Codes

The comparison [2] of various codes is as shown in figure:

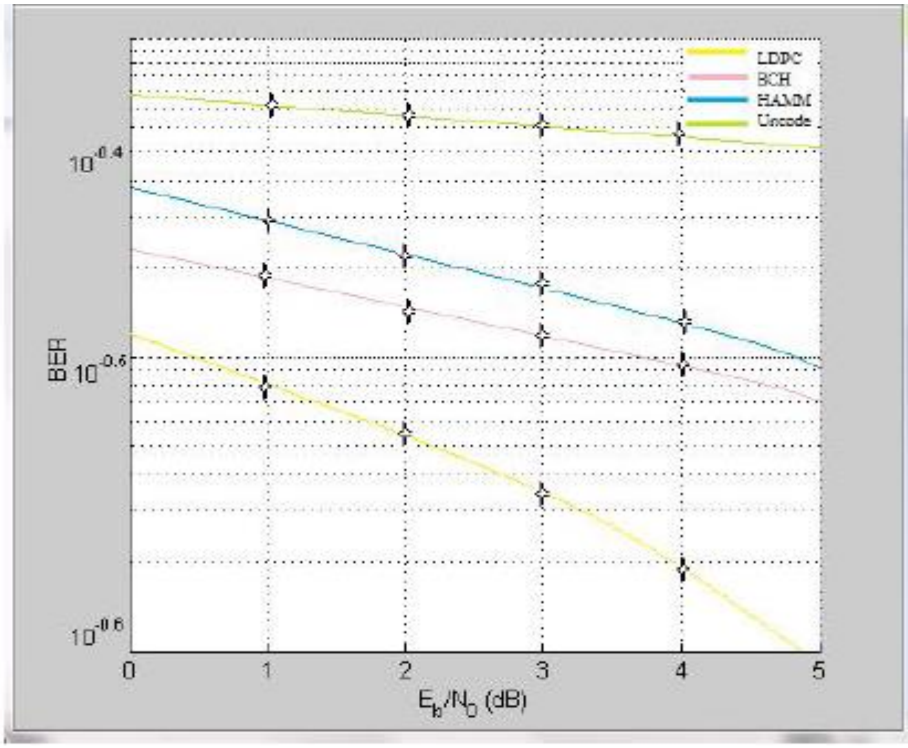


Fig 8: Comparison of various codes

Table 1: Comparing different Error-Correcting Codes on BER

Performance Metric	Hamming (7,4)	Repetition (3,1)	Hamming (15,11)	Hybrid code
BER	0.000	0.800	0.3637	0.111

2. LITERATURE REVIEW

Different reported techniques are discussed below:

- Blais et al. [3] proposed a method for improving code rate of BCH code over other codes by introducing some explicit families of good algebraic codes.
- Wallace et al. [4] proposed different decoding algorithms and shows that Berlekamp decoding algorithm is complex and not too attractive.
- Mitchell et al. [6] compared BER performance of different error correcting codes for various code rates.
- Fujihasji et al. [7] proposed a method to modified hamming codes that have the capability of all error detecting and one error correcting on an ideal optical channel.
- Singh and Bahel et al. [2] presented a study of various block code namely Hamming code and Bose-Chaudhuri-Hocquenghem(BCH) code and estimated the performance of such codes on the basis of Eb/No value.
- Neal et al. [8] proposed a method i.e. Repetition method to improve the communication between both sides.
- Berger and Todorov et al. [9] presented a method to improve the communication process using block error-correcting codes.

Table 2 shows the existing methods using ECCs.

Table 2: Existing Methods using ECCs

S.No.	Author's Name, Year	ECC Used	Results
1.	Eric Blais & Venkat Guruswami, 2010	BCH code	BCH code have better rate than binary codes constructed from RS codes.
2.	Hank Wallace, 2001	BCH code	The Berlekamp decoding algorithm is complex and not too attractive for high-speed communication system.
3.	Gregory Mitchell, 2009	Hamming code BCH code Reed Solomon	FEC codes allows communications to achieve the same level of transmission reliability, quantified by the BER, at lower output levels.
4.	Priti Shankar, 2000	Hamming code	To overcome the limitation of not being able to distinguish between single and double errors.
5.	Chugo Fujihashi, 2008	Hamming Code	Modified Hamming code offer a capability of all error detecting and one error correcting.
6.	Jaspreet Singh and Dr. Shalini Bahel, 2014	Hamming Code BCH Code	Performance of BCH code is better than Hamming Code. For smaller value of

			Eb/No the BCH is having better results and Hamming give better result for higher value.

3. DESIGN

3.1 Calculating the Hamming Code

The key to the Hamming Code [3] is the use of extra parity bits to allow the identification of a single error. Create the code word as follows:

- Mark all bit positions that are powers of two as parity bits. (Positions 1, 2, 4, 8, 16, 32, 64, etc.)
- All other bit positions are for the data to be encoded. (Positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
- Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.
  Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1,3,5,7,9,11,13,15,...)
  Position 2: check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc. (2,3,6,7,10,11,14,15,...)
  Position 4: check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc. (4,5,6,7,12,13,14,15,20,21,22,23,...)
  Position 8: check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, (8-15,24-31,40-47,...)
  Position 16: check 16 bits, skip 16 bits, check 16 bits, skip 16 bits, etc. (16-31,48-63,80-95,...)
  Position 32: check 32 bits, skip 32 bits, check 32 bits, skip 32 bits, etc. (32-63,96-127,160-191,...)

- Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

Example:

Data stored = 00111001

Check bits:

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Fig 9: Check bits calculation

Putting it together:

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1

Fig 10: code word representation

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

	C8	C4	C2	C1
	0	1	1	1
$\oplus$	0	0	0	1
	0	1	1	0

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

Fig 11: error correction in hamming code

3.2 BCH Decoding Algorithm

- Consider a BCH code [4] with $n = 2^m - 1$ and generator polynomial g(x).

- Suppose a code polynomial c(x) = $c_0 + c_1 x + L + c_{n-1}x^{n-1}$ is transmitted.

  Let r(x) = $r_0 + r_1 x + L + r_{n-1}x_{n-1}$ be the received polynomial.

- Then r(x) = c(x) + e(x), where e(x) is the error polynomial.
- To check whether r(x) is a code polynomial or not, we simply test

  whether $r(\propto) = r(\propto^2) = L = r(\alpha^{2t}) = 0$.

If yes, then r(x) is a code polynomial otherwise r(x) is not a code polynomial and the presence of errors is detected.

Procedure

The BCH codes decoding has 3 steps:

1. **Syndrome Computation**

The syndrome consists of $2t$ components in GF (2^m)

$$\overline{S} = (S_1 \; S_2 \cdots S_{2t})$$

and $S_i = r(\alpha^i)$ for $1 \le i \le 2t$

Computation:

Let $\emptyset_i(x)$ be the minimum polynomial of α^i

Dividing r(x) by $\emptyset_i(x)$ we obtain:

r(x) = a(x) $\emptyset_i(x)$ + b(x)

Then $S_i = b(\propto^i)$

$S_i = b(\propto^i)$ can be obtained by linear feedback shift-register with connection based on $\emptyset_i(x)$.

2. **Syndrome and Error Pattern**

Since r(x) = c(x) +e(x)

then $s_i = r(\propto^i) = c(\propto^i) + e(\propto^i) = e(\propto^i)$ for $1 \le i \le t$.

This gives a relationship between the syndrome and the error pattern.

Suppose e(x) has V errors $(V \le t)$ at the locations specified by $x^{j_1}, x^{j_2}, \ldots, x^{j_v}$.

i.e. $e(x) = x^{j_1} + x^{j_2} + \cdots + x^{j_v}$ where, $0 \le j_1 < j_2 < \cdots < j_v \le n-1$.

From above two equations, we have the following relation between syndrome components and error location:

$S_1 = e(\propto) = \propto^{j_1} + \propto^{j_2} + \ldots\ldots + \propto^{j_v}$

$S_2 = e(\propto^2) = (\propto^{j_1})^2 + (\propto^{j_2})^2 + \ldots\ldots + (\propto^{j_v})^2$

.

.

.

$S_{2t} = e(\propto^{2t}) = (\propto^{j_1})^{2t} + (\propto^{j_2})^{2t} + \ldots\ldots + (\propto^{j_v})^{2t}$

If we can solve the $2t$ equations, we can determine $\propto^{j_1}, \propto^{j_2}, \mathbf{L}, \propto^{j_v}$

The unknown parameter $\propto^{j_u} = \mathrm{Z_u}$ for u=1, 2, L, V are called the “error laocation number”.

When $\propto^{j_u}$,$1 \leq \mathrm{u} < \mathrm{V}$ are found, the powers j_u, u=1,2,L,V give us the error location in e(x).

3. Error Location Polynomial

(Error-Locator Polynomial)

Suppose that $v \leq t$ errors actually occur

Define error-locator polynomial L (z) as

L (z) = (1+Z_1z)(1+Z_2z)…….(1+Z_vz)

$= \prod_{i=1}^{v}(1 + Z_i\mathrm{z})$

$= \sigma_0 + \sigma_1\mathrm{z} + \sigma_2 z^2 + \ldots\ldots + \sigma_v z^v$

where, σ_0=1.

v

L(z) has Z_1^{-1},Z_2^{-1},……..,Z_v^{-1} as roots.

Note that Z= $\propto^{j_u}$

If we can determine L(z) from the syndrome $S = (S_1, S_2, \mathbf{L}, S_{2t})$,

then the roots of L(z) give us the error-location numbers.

3.3 Repetition Code Algorithm

We are trying to send a series of bits through a channel that sometimes randomly changes a bit from 0 to 1 or vice versa.

One way to improve reliability despite this “noise” is to send each bit three times.

E.g., if we want to send the bit sequence 01101, we actually send 000111111000111.

The receiver looks at the bits in groups of three, and decodes each group to the bit that occurs most often in the group.

An example in which the correct message is decoded despite four transmission errors:

$$\begin{array}{cccccc} & & \text{messsage transmitted} & & \\ 01101 & \rightarrow & 000111111000111 & & \\ & \rightarrow & 010111011100101 & \rightarrow & 01101 \\ & & \text{noisy message received} & & \end{array}$$

Limitation of Repetition Code

The receiver looks at the bits in groups of three, and decodes each group to the bit that occurs most often in the group. So, there will be no more than a single error in a particular group of three and If there, then it will decode codeword incorrectly.

3.4 Reed Solomon Code Implementation

An RS [6] (n, k, n-k+1) code has:

- k digit messages
- n digit codeword
- n – k + 1 distance between code words (at least)
- (n - k)/2 errors before it cannot be decoded
- 2s = n - k

Encoding Process:

- m is the message encoded as a polynomial
- m\` = m* pow(x,2s)
- b = m\`(mod g)

  where m\`= q*g + b for some q
- c = m\`- b

-Code words are multiples of g, and are systematic.

-Verifying a codeword is valid is a matter of checking for divisibility by g.

Decoding Process:

1. Calculate Syndromes

   Calculate the first 2s syndromes.

- Syndromes are defined for all l:

  $S_l = \sum_{i=1}^{s} Y_i X_i^l$
- For the first 2s, it reduces to:

  $S_l = E(\propto^l) = \sum_{i=1}^{s} Y_i \propto^{l_{j_i}}$ $1 \leq l \leq 2s$
  - $S_l = R(\propto^l) = E(\propto^l)$ for the first 2s powers of $\propto$.
- Encode the syndromes in a generator polynomial:

  $s(z) = \sum_{i=1}^{\infty} s_i z^i$
- This can be computed by finding each s_i from received codeword for the first 2s values.

2. Berlekamp-Massey Algorithm - calculates the Error Locator Polynomials and Error Evaluator Polynomials.

- Input: Syndrome polynomial from the above step.
- Output: Error Locator Polynomial σ (z) and Error Evaluator Polynomial ω (z).
- Defined as:

$$\sigma(z) = \prod_{i=1}^{s}(1 - X_i z)$$

$$\omega(z) = \sigma(z) + \sum_{i=1}^{s} zX_iY_i \prod_{\substack{j=1 \\ j\neq 1}}^{s}(1 - X_j z)$$

- Notice that the error locations are the inverse roots of σ(z). (Roots are 1/X 1, 1/X 2… 1/X s).
- Observe the following relation:

$$\frac{\omega(z)}{\sigma(z)} = 1 + \sum_{i=1}^{s} \frac{zX_iY_i}{1 - X_i z}$$

=……intermediate steps omitted

= 1 + s(z)

- Key equation thus states:

$$(1 + s(z))\sigma(z) \overset{(\mathrm{mod}\ z^{2s+1})}{=} \omega(z)$$

- σ(z) and ω(z) have degree at most s.
- Key Equation represents a set of 2s equations and 2s unknowns.
- B-M iterates 2s times.
- At each iteration, it produces a pair of polynomials:

$$(\propto_l\ (z), \omega_l\ (z))$$

- where the polynomials satisfy that iteration's key equation:

(1 + s (z)) σ(l) (z) (mod z l+1) = ω(l) (z)

3. Chien's Procedure

- Recall the definition of σ (z):

$\sigma (z) = \prod_{i=1}^{s}(1 - X_i z)$

- Now that we have σ (z), finding the array of Xi values is simply a matter of solving for the roots.
- The Easy Way: since we're working over a small field, just test every value
  1. Let α be a generator
  2. Initialize {Xi} to the empty set
  3. For l = 1, 2 . . . .

     If σ (α l) = 0: add α −l to {Xi}

4. Forney's Formula

   Using the Error Evaluator Polynomial ω (z) and the error locations {Xi},

   the error magnitudes {Yi} can be computed.

3.5 Hybrid Coding Implementation

In this type of coding we apply two error-correcting codes.

Here we used Hamming code [6] as an "outer code" and Repetition code[8] as an "inner code".

The inner code gets the error rate down and the Hamming code [6] is then applied to correct the rest of the errors. We denote this encoding scheme with Hamm/Inn or Hamm/Rept.

→ Hamming encoder →Repetition encoder → Repetition decoder →Hamming dc

In this we apply a similar error-correcting scheme by using Hamming code [6] with proper parameters as an outer code and Repetition code [8] as an inner code.

- As we have already explained it is more applicable to use repetition code as inner code and Hamming code as outer code in hybrid error correcting scheme. In this case the signature error probability is given by:

$$P_{sig,hybrid} = \sum_{i=t+1}^{n} \binom{n}{i} P_{rep}^{i}(1 - P_{rep})^{n-i}$$

4. IMPLEMENTATION

4.1 Comparison Parameters:

- Error detection capability
- Bit-error rate(BER)

4.2 Tools and Technologies:

4.2.1 Java 1.6 Version:

4.2.1.1 Characteristics:

JAVA is a programming language [1], developed by Sun Microsystems and first released in 1995 (release 1.0). Since that time, it gained a large popularity mainly due to two characteristics:

- A JAVA programme is hardware and operating system independent. If well written (!), the same JAVA programme, compiled once, will run identically on a SUN/Solaris workstation, a PC/windows computer or a Macintosh computer. Not mentioning other UNIX flavors, including Linux, and every Web browser, with some restrictions described below. This universal executability is made possible because a JAVA programme is run through a JAVA Virtual Machine.
- It is an object oriented language. This feature is mainly of interest for software developers.

4.2.1.2 JAVA Virtual Machine (JVM):

A JAVA programme is build by a JAVA compiler which generates its own binary code. This binary code is independant from any hardware and operating system. To be executed, it needs a *virtual machine*, which is a programme analyzing this code and executing the instructions it contains.

Of course, this Java Virtual Machine (JVM)[1] is hardware and operating system dependant. Two types of Virtual Machines exist: those included in every Web Browser, and those running as an independent programme, like the Java RunTime Environment (JRE) from Sun Microsystems. These programmes need to be downloaded for your particular platform.

4.3 Diagrams

4.3.1 Use Case Diagram

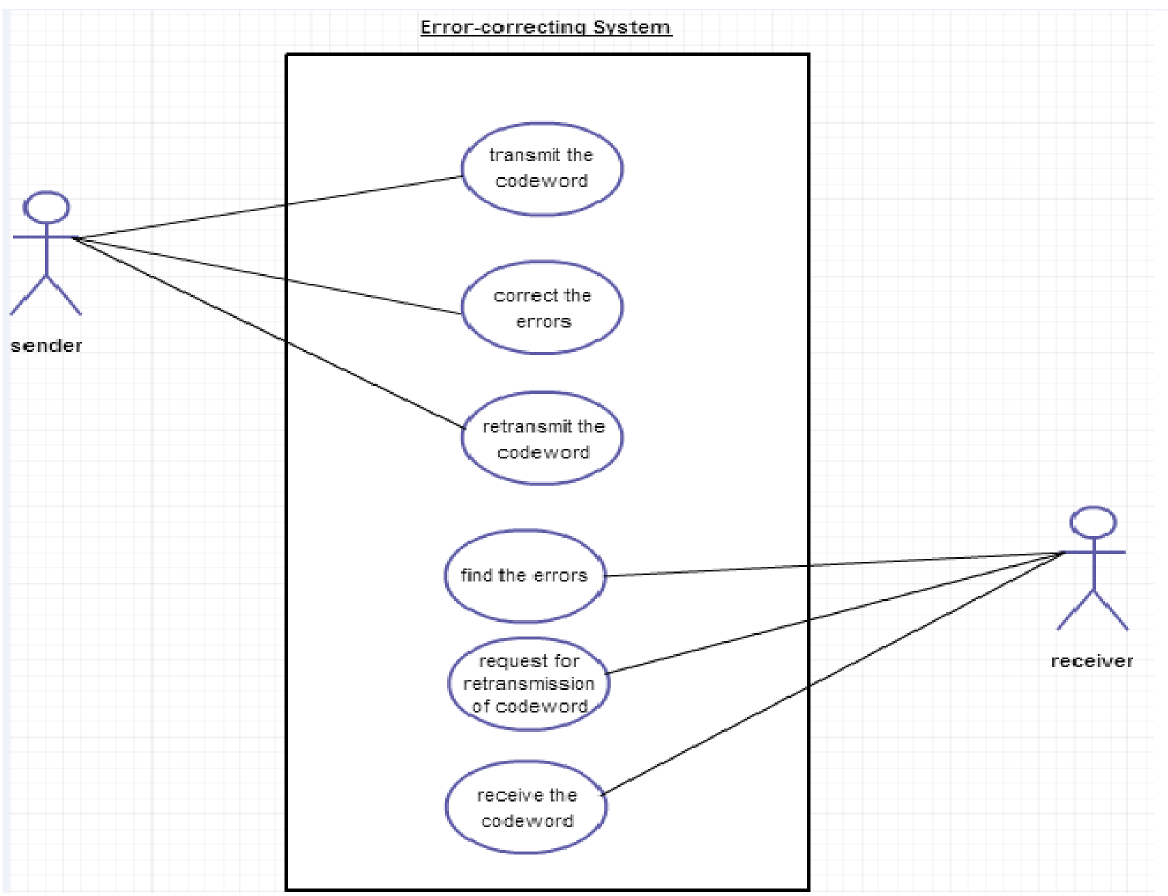


Fig 12: Use Case Diagram

4.3.2 Activity Diagram

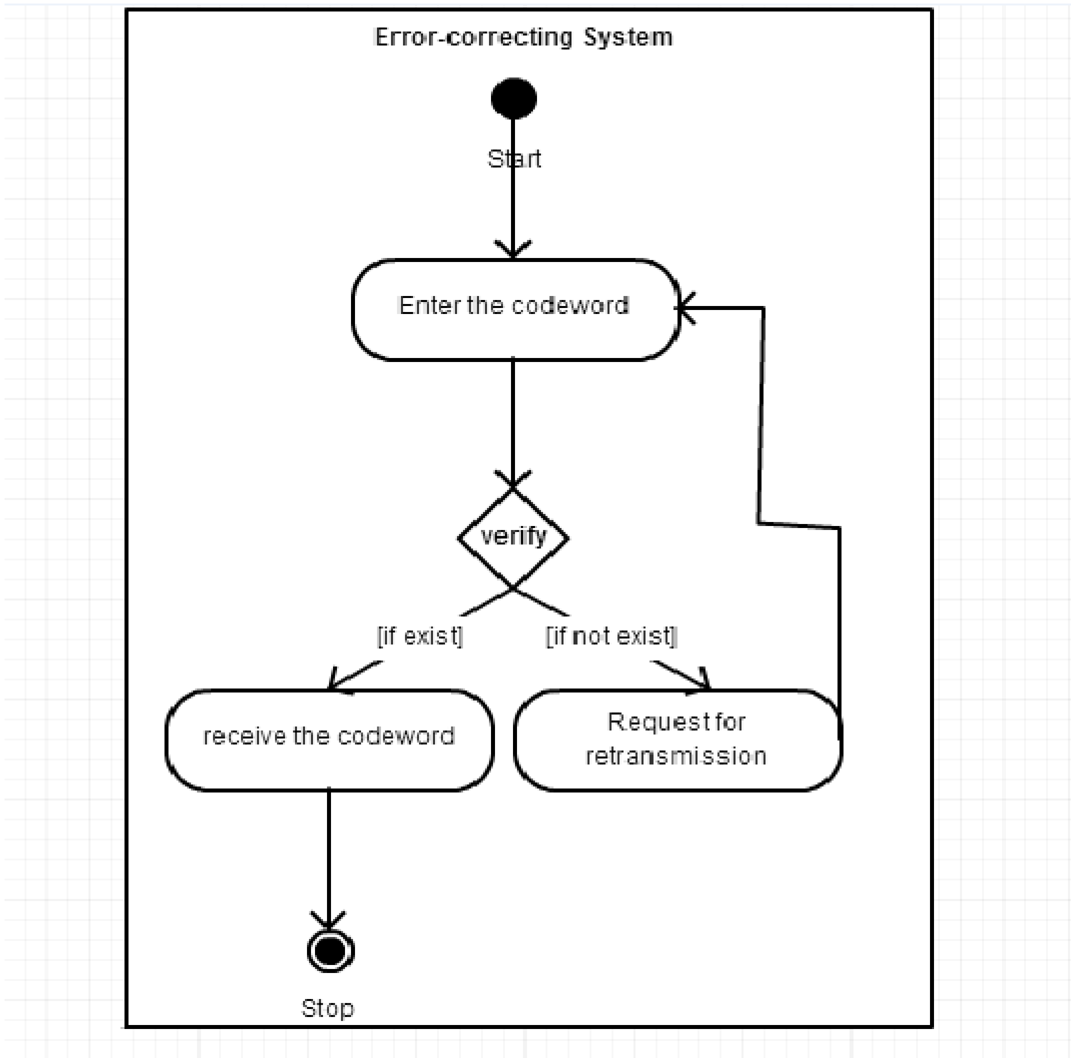


Fig 13: Activity Diagram

4.4 Code

4.4.1 BCH Code

```
package error.codes;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Random;
public class BCHNEW {
    int m = 5, n = 31, k = 21, t = 2, d = 5;
    int length = 31;
    int p[] = new int[6];
    int alpha_to[] = new int[32];
    int index_of[] = new int[32];
    int g[] = new int[11];
    int recd[] = new int[31];
    int data[] = new int[21];
    int bb[] = new int[11];
    int numerr, decerror = 0;
    int errpos[] = new int[32];
    int seed;
    void read_p() {
            p[0] = p[2] = p[5] = 1;
            p[1] = p[3] = p[4] = 0;
    }
    void generate_gf() {
            int i, mask=1;
            alpha_to[m] = 0;
            for (i = 0; i < m; i++) {
                    alpha_to[i] = mask;
                    index_of[alpha_to[i]] = i;
                    if (p[i] != 0)
```

```
            alpha_to[m] ^= mask;
            mask <<= 1;
        }
        index_of[alpha_to[m]] = m;
        mask >>= 1;
        for (i = m + 1; i < n; i++) {
            if (alpha_to[i - 1] >= mask)
        alpha_to[i] = alpha_to[m] ^ ((alpha_to[i - 1] ^ mask) << 1);
            else alpha_to[i] = alpha_to[i - 1] << 1;
            index_of[alpha_to[i]] = i;
        }
        index_of[0] = -1;
    }
    void gen_poly() {
        int ii, jj, ll, kaux;
        int test, aux, nocycles, root, noterms, rdncy;
        int cycle[][] = new int[15][6];
        int size[] = new int[15];
        int min[] = new int[11];
        int zeros[] = new int[11];
        cycle[0][0] = 0;
        size[0] = 1;
        cycle[1][0] = 1;
        size[1] = 1;
        jj = 1;
        do {
            ii = 0;
        do {
            ii++;
            cycle[jj][ii] = (cycle[jj][ii - 1] * 2) % n;
            size[jj]++;
```

```
                aux = (cycle[jj][ii] * 2) % n;
        } while (aux != cycle[jj][0]);
        ll = 0;
        do {
                ll++;
                test = 0;
        for (ii = 1; ((ii <= jj) && (test == 0)); ii++)
for (kaux = 0; ((kaux < size[ii]) && (test == 0)); kaux++)
                if (ll == cycle[ii][kaux])
                                        test = 1;
        } while ((test != 0) && (ll < (n - 1)));// test
        if (test == 0) {// (!test)
                jj++; /* next cycle set index */
                cycle[jj][0] = ll;
                size[jj] = 1;
        }
} while (ll < (n - 1));
nocycles = jj; /* number of cycle sets modulo n */
kaux = 0;
rdncy = 0;
for (ii = 1; ii <= nocycles; ii++) {
        min[kaux] = 0;
        for (jj = 0; jj < size[ii]; jj++)
                for (root = 1; root < d; root++)
                        if (root == cycle[ii][jj])
                                min[kaux] = ii;
        if (min[kaux] != 0) {
                rdncy += size[min[kaux]];
                kaux++;
        }}
noterms = kaux;
```

```
                kaux = 1;
                for (ii = 0; ii < noterms; ii++)
                        for (jj = 0; jj < size[min[ii]]; jj++) {
                                zeros[kaux] = cycle[min[ii]][jj];
                                kaux++;
                        }
System.out.printf("This is a (%d, %d, %d) binary BCH code\n", length,k, d);
                g[0] = alpha_to[zeros[1]];
                g[1] = 1; /* g(x) = (X + zeros[1]) initially */
                for (ii = 2; ii <= rdncy; ii++) {
                        g[ii] = 1;
                        for (jj = ii - 1; jj > 0; jj--)
                                if (g[jj] != 0)
        g[jj] = g[jj - 1] ^alpha_to[(index_of[g[jj]] + zeros[ii]) % n];
                                else
                                        g[jj] = g[jj - 1];
                        g[0] = alpha_to[(index_of[g[0]] + zeros[ii]) % n];
                }
                System.out.printf("g(x) = ");
                for (ii = 0; ii <= rdncy; ii++) {
                        System.out.printf("%d", g[ii]);
                        if ((ii != 0) && ((ii % 70) == 0))
                                System.out.printf("\n");
                }
                System.out.printf("\n");
        }
        void encode_bch() {
                int i, j;
                int feedback;
                for (i = 0; i < length - k; i++)
                        bb[i] = 0;
```

```
        for (i = k - 1; i >= 0; i--) {
            feedback = data[i] ^ bb[length - k - 1];
            if (feedback != 0) {
                for (j = length - k - 1; j > 0; j--)
                    if (g[j] != 0)
                        bb[j] = bb[j - 1] ^ feedback;
                    else
                        bb[j] = bb[j - 1];
                bb[0] = g[0] & feedback;// g[0] &&feedback
            } else {
                for (j = length - k - 1; j > 0; j--)
                    bb[j] = bb[j - 1];
                bb[0] = 0;
            }
        }}
    void decode_bch() {
        int i, j, q;
        int elp[] = new int[3], s[] = new int[5], s3;
        int count = 0, syn_error = 0;
        int loc[] = new int[3], err[] = new int[3], reg[] = new int[3];
        int aux;
        System.out.printf("s[] = (");
        for (i = 1; i <= 4; i++) {
            s[i] = 0;
            for (j = 0; j < length; j++)
                if (recd[j] != 0)
                    s[i] ^= alpha_to[(i * j) % n];
            if (s[i] != 0)
            syn_error = 1; /* set flag if non-zero syndrome */

            s[i] = index_of[s[i]];
```

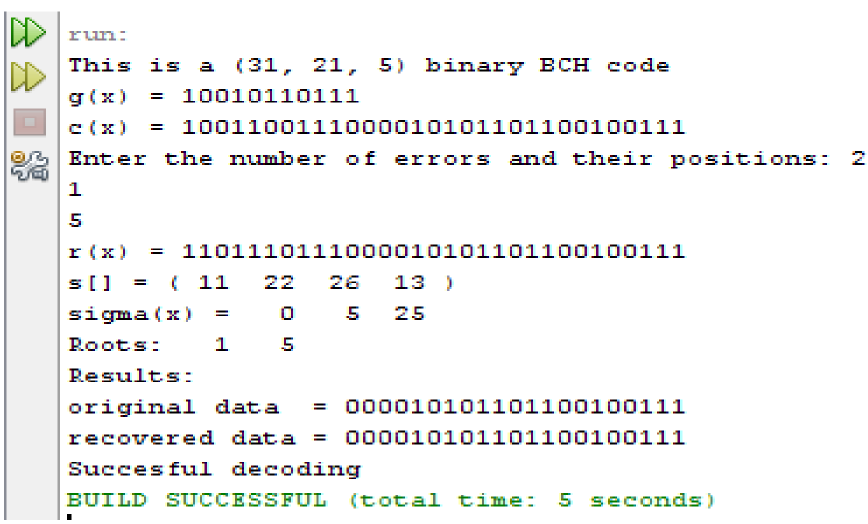
```
System.out.printf("%3d ", s[i]);
}
System.out.printf(")\n");
if (syn_error != 0) { /* If there are errors, try to correct them */
	if (s[1] != -1) {
		s3 = (s[1] * 3) % n;
		if (s[3] == s3) /* Was it a single error ? */
		{
			System.out.printf("One error at%d\n", s[1]);
			recd[s[1]] ^= 1; /* Yes: Correct it */
		} else {
			if (s[3] != -1)
		aux = alpha_to[s3] ^ alpha_to[s[3]];
			else
				aux = alpha_to[s3];
			elp[0] = 0;
		elp[1] = (s[2] - index_of[aux] + n) % n;
		elp[2] = (s[1] - index_of[aux] + n) % n;
			System.out.printf("sigma(x) = ");
			for (i = 0; i <= 2; i++)
		System.out.printf("%3d ", elp[i]);
			System.out.printf("\n");
			System.out.printf("Roots: ");
			for (i = 1; i <= 2; i++)
				reg[i] = elp[i];
			count = 0;
		for (i = 1; i <= n; i++) { /* Chien search */
				q = 1;
				for (j = 1; j <= 2; j++)
					if (reg[j] != -1) {
				reg[j] = (reg[j] + j) % n;
```

```
                    q ^= alpha_to[reg[j]];
                            }
                    if (q == 0) { /* store error location number*/
                    loc[count] = i % n;
                    count++;
                    System.out.printf("%3d ", (i % n));
                }}
                System.out.printf("\n");
                    if (count == 2)
                for (i = 0; i < 2; i++)
                    recd[loc[i]] ^= 1;
                    else
            System.out.printf("incomplete decoding\n");
                }
            } else if (s[2] != -1) /* Error detection */
            System.out.printf("incomplete decoding\n");
        }}
public void run() {
        int i;
            read_p(); /* read generator polynomial g(x) */
generate_gf(); /* generate the Galois Field GF(2**m) */
gen_poly(); /* Compute the generator polynomial of BCH code */
        seed = 1;
        Random  random = new Random(seed);
        for (i = 0; i < k; i++)
            data[i] = (random.nextInt() & 67108864) >> 26;
        encode_bch(); /* encode data */
        for (i = 0; i < length - k; i++)
        recd[i] = bb[i]; /* first (length-k) bits are redundancy */
        for (i = 0; i < k; i++)
        recd[i + length - k] = data[i]; /* last k bits are data */
```

```
System.out.printf("c(x) = ");
for (i = 0; i < length; i++) {
    System.out.printf("%1d", recd[i]);
    if ((i != 0) && ((i % 70) == 0))
        System.out.printf("\n");
}
System.out.printf("\n");
System.out.printf("Enter the number of errors and their positions: ");
BufferedReader wt = new BufferedReader(new
InputStreamReader(System.in));
String numerrStr = null;
try {
    numerrStr = wt.readLine();
} catch (IOException e) {
            e.printStackTrace();
}
numerr = Integer.valueOf(numerrStr);
for (i = 0; i < numerr; i++) {
    String errposStr = null;
    try {
        errposStr = wt.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    errpos[i] = Integer.valueOf(errposStr);
    recd[errpos[i]] ^= 1;
}
System.out.printf("r(x) = ");
for (i = 0; i < length; i++)
System.out.printf("%1d", recd[i]);
decode_bch();
```

```
            System.out.printf("Results:\n");
            System.out.printf("original data  = ");
            for (i = 0; i < k; i++)
                  System.out.printf("%1d", data[i]);
            System.out.printf("\nrecovered data = ");
            for (i = length - k; i < length; i++)
                  System.out.printf("%1d", recd[i]);
            System.out.printf("\n");
            for (i = length - k; i < length; i++)
                  if (data[i - length + k] != recd[i])
                        decerror++;
            if (decerror != 0)
              System.out.printf("%d message decoding errors\n", decerror);
            else     System.out.printf("Succesful decoding\n");
      }
         public static void main(String[] args) {
            BCHNEW bch_32_21_5 = new BCHNEW();
            bch_32_21_5.run();
      }   }
```

OUTPUT: (Snapshot)



```
run:
This is a (31, 21, 5) binary BCH code
g(x) = 10010110111
c(x) = 1001100111000010101101100100111
Enter the number of errors and their positions: 2
1
5
r(x) = 1101110111000010101101100100111
s[] = ( 11  22  26  13 )
sigma(x) =   0   5  25
Roots:   1   5
Results:
original data  = 000010101101100100111
recovered data = 000010101101100100111
Succesful decoding
BUILD SUCCESSFUL (total time: 5 seconds)
```

Fig 14: Output of the code

5. FUTURE OUTLOOK

5.1 Description

The next course of action includes the performance improvement of Error Correcting Codes and implementation of an application using Hybrid code. The implementation will be preceded by the design and modeling of method. The implementation will be followed by the testing phase of scheme.

CONCLUSION

In this report, we mainly work on detecting and correcting errors in a codeword during data transmission by using different error correcting techniques. Different Error correcting codes work on different parameters. Some of the codes are Hamming code, BCH code, Repetition Code and Hybrid Code.

Hamming code correct an error detected at the receiver side. The main advantage of this code is Encoding and Decoding are easy to implement. But, the problem with this code is that they can detect and correct only a single error. So, Hamming code is limited to few applications only.

So we will try to produce a double error correcting sub-code of the Hamming code by removing some code words to make a new one.BCH code is a generalization of hamming codes for multiple error correction. Another advantage of BCH code is the ease with which they can be decoded, namely, via an algebraic method known as syndrome decoding.

Repetition Code can correct multiple errors, but the problem with this code is its Bit Error Rate (BER) is too high.

So we will try to implement a different kind of code i.e. The Hybrid Code. It is a combination of two error correcting codes. Here we used hamming code as an "outer code" and repetition code as an "inner code". It's main advantages are it can correct multiple errors and produces very low BER.

REFERENCES

[1] Herbert Schildt “Complete Reference JAVA, 5thedition, TataMcGraw Hill”, chapter 5

[2] Jaspreet singh and Dr. Shalini Bahel “Comparitive Study of different Transmission Techniques of different Block Codes”, IJETT, vol-9 No.12, Mar-2014

[3] Eric Blais & Venkat Guruswami “Introduction to Coding Theory”, Notes 6, Feb-2010

[4] Hank Wallace, “error detection and correction using BCH code” Atlantic Quality Design, Inc. , 2001

[5] Shu Lin and Daniel J.Costello Jr. “Error control Coding” 2nd edition, Englewood Cliffs, New Jersey: Prentice Hall, 1983

[6] Gregory Mitchell “Investigation of Hamming, Reed-Solomon, and BCH Forward Error Correcting Codes”, ARL-TR-4901,July 2009

[7] Chugo Fujihashi “Error Detecting Modified Hamming Codes for ideal optical channel”, Tokyo Polytech. Univ. vol. 31, No., 2008

[8] Radford M.Neal “Performance Improvement of Repetition code”, Notes for CSC 310, 2004.

[9] Thierry Berger and Todor Todorov “Improving the watermarking process with usage of block error-correcting codes”, Serdica J.Computing ,163-180, 2008.

Appendix

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package test;

import java.util.Scanner;
import java.io.*;
import java.util.Random;
/**
 *
 * @author Saurabh
 */
public class HybridCode {
    public static void main(String args[]){
    int i=0,j=0,dd=0;

  int total_bits;
  double bits=0;
  int a[]=new int[11];
  int b[]=new int[15];
  int c[]=new int[15];
  int d[]=new int[45];
  int p[]=new int[4];
  int bb[]=new int[45];
  int cc[]=new int[15];
  Scanner sr=new Scanner(System.in);

  String Hex="ABC";
  int location =0;
```

```
for(i=0;i<3;i++) //input data in array a
{
   if(Hex.charAt(i)=='A')
   {
      a[location++]=0;
      a[location++]=1;
      a[location++]=0;
   }
   else if(Hex.charAt(i)=='B')
   {
      a[location++]=1;
      a[location++]=0;
      a[location++]=1;
      a[location++]=1;
   }
   else if(Hex.charAt(i)=='C')
   {
      a[location++]=1;
      a[location++]=1;
      a[location++]=0;
      a[location++]=0;
   }
}
total_bits=a.length;
System.out.println("Data codeword length is:");
System.out.println(total_bits);
System.out.println("Data code is:");

for(i=0;i<11;i++)  //display data through array a
{
 System.out.print(a[i]);
```

```
 System.out.print("\t");
}

for(j=0;j<15;j++)  // Hamming code for even parity input in array b
{
 b[j]=0;
}

if((a[0]+a[1]+a[3]+a[4]+a[6]+a[8]+a[10])%2==0)
  b[0]=0;
  else
  b[0]=1;

if((a[0]+a[2]+a[3]+a[5]+a[6]+a[9]+a[10])%2==0)
  b[1]=0;
  else
  b[1]=1;

  if((a[1]+a[2]+a[3]+a[7]+a[8]+a[9]+a[10])%2==0)
  b[3]=0;
  else
  b[3]=1;

  if((a[4]+a[5]+a[6]+a[7]+a[8]+a[9]+a[10])%2==0)
  b[7]=0;
  else
  b[7]=1;

  System.out.println("");
  for(j=0,i=0;j<15;)
  {
```

```
 if(j==0||j==1||j==3||j==7)
 j++;
 else
  {
    b[j]=a[i];
     j++;
     i++;
  }
}
System.out.println(""); //print the Hamming code for even parity
System.out.println("Hamming code for even parity is:");

for(j=0;j<15;j++)
{
 System.out.print(b[j]);
 System.out.print("\t");
}
System.out.println("");
System.out.println("");

for(i=0;i<15;i++)
{
 c[i]=b[i];
}

Random r= new Random();
int rand = r.nextInt(15);

 if(c[rand]==1)
     c[rand]=0;
 else
```

```
    c[rand]=1;



  System.out.println("the hamming code with error is:");
  for(j=0;j<15;j++)
  System.out.print(c[j]);
  System.out.println("");

  int e=0;
  int size=45;
  for(i=0;e<15;)
  {
   if(c[e]==1)
   {
      bb[i++]=1;
      bb[i++]=1;
      bb[i++]=1;

      e++;
   }
   else
   {
      bb[i++]=0;
      bb[i++]=0;

      bb[i++]=0;
      e++;
   }

}
  System.out.print("\n");
```

```
System.out.print("Received Data code:");
for(i=0;i<size;i++){
    System.out.print(bb[i]);
}
System.out.print("\n");
System.out.print("\n");

for(i=0;i<size;i++){
    d[i]=bb[i];
}

System.out.println("Enter an error in data code:"); //Input Hamming code by
user and compare
for(i=0;i<45;i++)
{
 bb[i]=sr.nextInt();
}

System.out.print("\n");
System.out.println("Encrypted Data:");

  for(j=0;j<size;j++){
   //System.out.print("");
   System.out.print(" " + bb[j]);
   }

  System.out.println("");
int same=0;
for(i=0;i<=d.length-1;i++){
    if(d[i]!=bb[i]){
        same++;
```

```
    }
  }
  System.out.println("Number of Errors in a code word:");
  System.out.println(same);

  int count=0,index=0,countz=0,counto=0;
 for(i=0;i<size;i++){
   if(bb[i]==0)
       countz++;
     else
       counto++;

   if(i%3==2)
   {
    if(countz>counto)
    {
      cc[index++]=0;
    }
    else
    {
      cc[index++]=1;
    }
    countz=0;
    counto=0;
   }
 }
 System.out.println("\nFinal data after decrytption n correction is by Repetition Code:");
  for(i=0;i<15;i++)
    System.out.print(cc[i]);
```

```
p[0]=cc[0];
p[1]=cc[1];
p[2]=cc[3];
p[3]=cc[7];

if((((c[2]+c[4]+c[6]+c[8]+c[10]+c[12]+c[14])%2)==0 &&
p[0]==0||(((c[2]+c[4]+c[6]+c[8]+c[10]+c[12]+c[14])%2)!=0 && p[0] ==1)))
p[0]=0;
else
p[0]=1;


if((((c[2]+c[5]+c[6]+c[9]+c[10]+c[13]+c[14])%2)==0 &&
p[1]==0||(((c[2]+c[5]+c[6]+c[9]+c[10]+c[13]+c[14])%2)!=0 && p[1] ==1)))
p[1]=0;
else
p[1]=1;


if((((c[4]+c[5]+c[6])%2)==0 && p[2]==0||(((c[4]+c[5]+c[6])%2)!=0 &&
p[2] ==1)))
p[2]=0;
else
p[2]=1;


if((((c[8]+c[9]+c[10])%2)==0 && p[3]==0||(((c[8]+c[9]+c[10])%2)!=0 &&
p[3] ==1)))
p[3]=0;
else
p[3]=1;
```

```
    for(i=3;i>=0;i--)   //find out the place for wrong bit.
    {
     dd=dd+(p[i]*(int)Math.pow(2,i));
    }

    System.out.println("");
    if(dd==0)
    System.out.println("The data code is correctly received");
    else
    System.out.println("The "+ dd +"  bit is wrongly received");

    if(c[dd-1]==0)  //correct Hamming code.
     c[dd-1]=1;
      else
     c[dd-1]=0;
       System.out.println("");

      System.out.println("The Correct Hamming code is:");
      for(i=0;i<15;i++)
      {
          System.out.print(c[i]);
          System.out.println("\t");
      }
      System.out.println("The Bit Error Rate:");
       bits=(double) same/total_bits;
        System.out.printf("%f",bits);
  }
}
```