Activity based

Project Report on

Computer Networks

Submitted to Vishwakarma University, Pune

Under the Initiative of

Contemporary Curriculum, Pedagogy, and Practice (C2P2)

By

Merwin Pinto

SRN No : 202100102

Roll No : 01

Diya Oswal

SRN No : 202101718

Roll No : 41

Div : E

Third Year Engineering

Department of Computer Engineering

Faculty of Science and Technology

Academic Year

2023-2024

Project Description:

Parity Bit Checker for Error detection and Correction:

The parity bit checker is a network method designed to detect errors and check the integrity of the data received at the receiver side by the sender side. The parity check method adds a bit to the original data for checking errors at the receiver end.

There are mainly two types of Parity that is **Even Parity and Odd Parity**

This Parity Checker is further divided into 3 parts

1. Message representation into Binary bits and frames

2. Parity bit checking using Even and odd parity concepts

3. comparing performances with other error detection Techniques

PROJECT MODULE 3 : Comparing performances with other error detection Techniques

Performance Analysis:

Parity bit checking is a simple and efficient error detection technique that involves adding an extra bit to the data transmission. The parity bit is calculated based on the number of 1s in the data, and it is set to either 0 or 1 to make the total number of 1s even (even parity) or odd (odd parity). When the data is received, the parity bit is recalculated and compared to the original parity bit. If the two parity bits are the same, then the data is considered correct. If the two parity bits are different, then an error has occurred.

Parity bit checking is a very simple and efficient way to detect single-bit errors. However, it is important to note that it cannot detect all types of errors, such as two-bit errors or errors that occur

in the parity bit itself. For more sophisticated error detection and correction techniques, such as Hamming codes, are used.

Advantages of Parity Bit Checking

Simple and efficient to implement

Low computational overhead

Can detect single-bit errors

Does not require complex hardware or software

Disadvantages of Parity Bit Checking

Cannot detect all types of errors, such as two-bit errors or errors in the parity bit itself

Not as reliable as more sophisticated error detection techniques

Can add overhead to data transmissions

Different Error Detection Techniques

Hamming Distance

Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of substitutions required to change one string into the other. The Hamming distance between two strings, a and b, is denoted as d(a, b).

Applications

The Hamming distance is used in a variety of applications, including:

Error detection and correction: The Hamming distance can be used to detect and correct errors in data transmissions. For example, if the Hamming distance between two codewords is 1, then we can be sure that there is only one error in the data. If the Hamming distance is 2, then we can correct the error by flipping the bit that is different in the two codewords.

Data compression: The Hamming distance can be used to compress data by removing redundant information. For example, we can use the Hamming distance to identify and remove duplicate codewords in a dictionary.

Cryptography: The Hamming distance can be used to measure the similarity of two cryptographic keys. A small Hamming distance between two keys suggests that they may be related, which could be a security vulnerability.

Properties

The Hamming distance has several properties that make it a useful metric for measuring the similarity of strings:

Non-negativity: The Hamming distance between any two strings is always non-negative.

Symmetry: The Hamming distance between two strings is equal to the Hamming distance between the reversed strings.

Identity: The Hamming distance between two identical strings is zero.

Triangle inequality: For any three strings, a, b, and c, the Hamming distance between a and c is less than or equal to the sum of the Hamming distances between a and b and b and c.

Comparison

Feature	Parity Bit Checking	CRC	Hamming Codes
Error Detection	Single-bit errors	Single-bit errors, multi-bit errors, and burst errors	Single-bit errors and some two-bit errors
Error Correction	No	No	Yes
Implementation Complexity	Low	Medium	High
Computational Overhead	Low	Medium	High
Error Protection Level	Low	Medium	High
Common Applications	Simple data transmissions	Data communication protocols, storage devices	Error-critical applications, such as satellite communications

Analysis Parameters

To effectively compare parity bit checking with other error detection techniques, several performance parameters need to be considered:

Error Detection Capability: The ability to detect different types of errors, such as single-bit errors, multi-bit errors, and burst errors.

Error Detection Efficiency: The ratio of errors detected to the number of redundant bits used.

Implementation Complexity: The complexity of the hardware or software required to implement the error detection technique.

Computational Overhead: The additional processing time required to perform error detection.

Error Correction Capability: The ability to not only detect errors but also correct them without retransmission.

Cyclic Redundancy Check (CRC)

Cyclic Redundancy Check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to digital data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial 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 (see bitfilters).

How it Works

A CRC-enabled device calculates a short, fixed-length binary sequence, known as the check value or CRC, for each block of data to be sent or stored and appends it to the data, forming a codeword. When a codeword is received or read, the device either compares its check value with one freshly calculated from the data block, or equivalently, performs a CRC on the whole codeword and compares the resulting check value with an expected residue constant.

CRC Calculation

The CRC calculation is based on polynomial division. The data block is divided by a predetermined generator polynomial, and the remainder is the CRC value. The generator polynomial is a carefully chosen binary polynomial that has certain properties that make it suitable for CRC calculations.

CRC Properties

CRCs have several properties that make them effective error detection codes:

They are easy to calculate and implement in hardware or software.
They can detect a wide range of errors, including single-bit errors, multi-bit errors, and burst errors.
They are relatively immune to noise

.

CRC Applications

CRCs are used in a wide variety of applications, including:

Data communication protocols: CRCs are used in many data communication protocols, such as Ethernet, PPP, and USB, to detect errors in data transmissions.
Storage devices: CRCs are used in storage devices, such as hard disks and solid-state drives, to detect errors in stored data.
Error correction codes: CRCs can be used in conjunction with other error detection and correction codes to provide more sophisticated error protection.

Implementation :

Code :

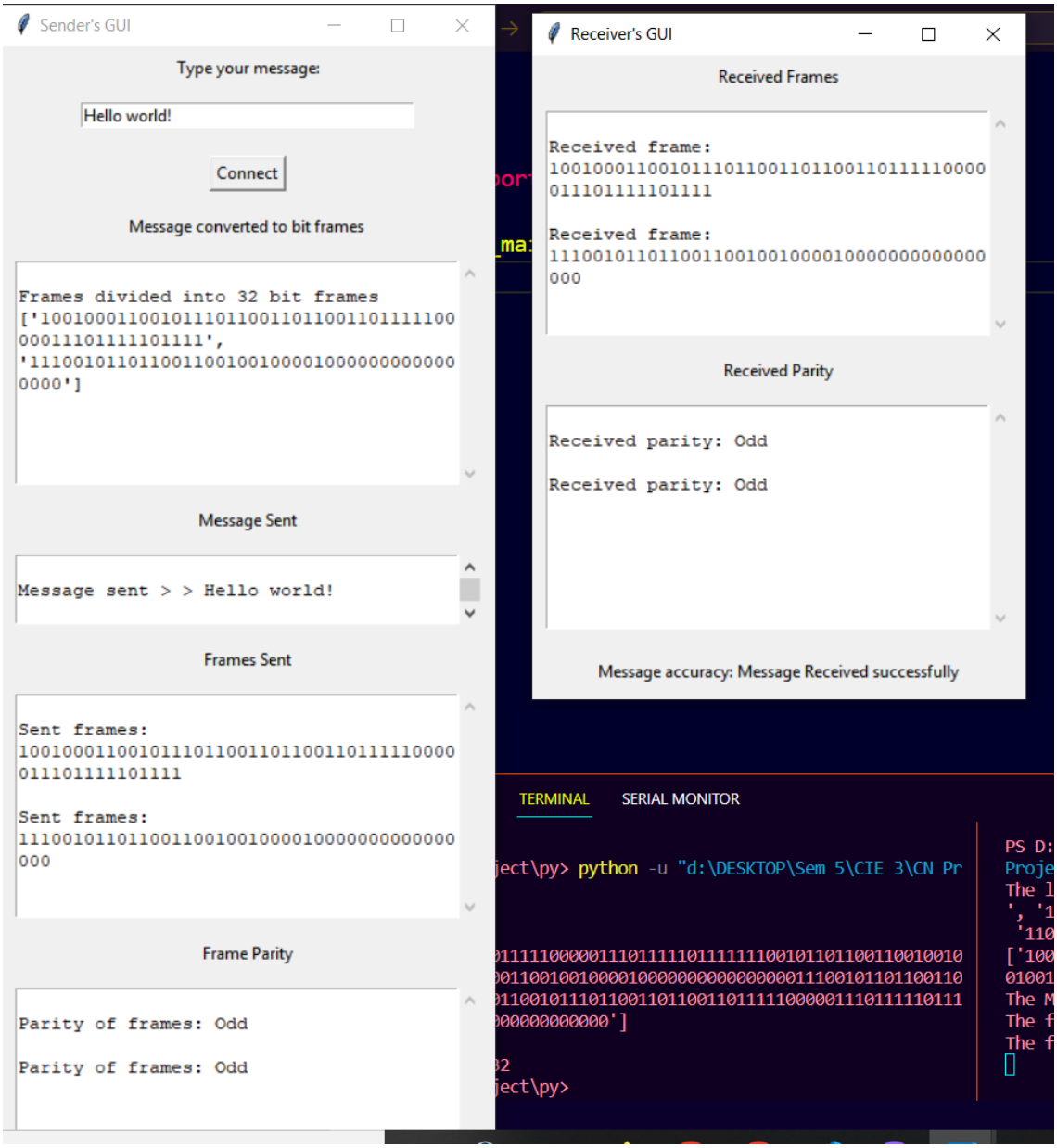
```
import socket
import tkinter as tk
from tkinter import scrolledtext
from code_file import Frame_creation, check_parity,
simulate_frame_corruption

class MultiGUI:
    def __init__(self, master):
        self.master = master
        master.title("Sender's GUI")
        self.prompt_label = tk.Label(master, text="Type your message:")
        self.prompt_label.pack(pady=5)
        self.input_entry = tk.Entry(master, width=40)
        self.input_entry.pack(padx=10, pady=10)
        self.connect_button = tk.Button(master, text="Connect",
command=self.connect_to_server)
        self.connect_button.pack(pady=10)
        self.prompt_label1 = tk.Label(master, text=f"Message converted
to bit frames ")
        self.prompt_label1.pack(pady=5)
        self.text_area1 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area1.pack(padx=10, pady=10)
        self.prompt_label2 = tk.Label(master, text="Message Sent")
        self.prompt_label2.pack(pady=5)
        self.text_area2 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=2)
        self.text_area2.pack(padx=10, pady=10)
        self.prompt_label3 = tk.Label(master, text="Frames Sent")
        self.prompt_label3.pack(pady=5)
```

```
        self.text_area3 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area3.pack(padx=10, pady=10)
        self.prompt_label4 = tk.Label(master, text="Frame Parity")
        self.prompt_label4.pack(pady=5)
        self.text_area4 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area4.pack(padx=10, pady=10)
        self.prompt_label5 = tk.Label(master, text="Received Frames")
        self.prompt_label5.pack(pady=5)
        self.text_area5 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area5.pack(padx=10, pady=10)
        self.prompt_label6 = tk.Label(master, text="Received Parity")
        self.prompt_label6.pack(pady=5)
        self.text_area6 = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area6.pack(padx=10, pady=10)

        self.text_area_msg = scrolledtext.ScrolledText(master,
wrap=tk.WORD, width=40, height=10)
        self.text_area_msg.pack(padx=10, pady=10)
    def read_file(self, file_name):
        try:
            with open(file_name, 'r') as file:
                content = file.read()
            return content
        except FileNotFoundError:
            return None
    def connect_to_server(self):
        string = self.input_entry.get()
        frame_size = 32
        char_bit = 4
        error_rate = 1
        skip_probability = 0.5
        client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        port_no = ('localhost', 8080)
        client_socket.connect(port_no)
        frames = Frame_creation(string, frame_size, char_bit)
        self.text_area1.insert(tk.END, f"\nFrames divided into
{frame_size} bit frames \n{frames}\n")
        parity_type_holder1 = []
        self.text_area2.insert(tk.END, f"\nMessage sent > > {string}\n")
```

```
        for fr in frames:
            frame_type = check_parity(fr)
            parity_type_holder1.append(frame_type)
        for frame, parity in zip(frames, parity_type_holder1):
            client_socket.send(frame.encode())
            self.text_area3.insert(tk.END, f"\nSent frames: {frame}\n")
            self.text_area4.insert(tk.END, f"\nParity of frames:
{parity}\n")
        with open("p1.txt", 'w') as file:
            for i in parity_type_holder1:
                file.write(i + '\n')
        content1 = self.read_file("p1.txt")
        C_received_frames = simulate_frame_corruption(frames,
error_rate, skip_probability)
        parity_type_holder2 = []
        for fr2 in C_received_frames:
            client_socket.send(frame.encode())
            frame_type2 = check_parity(fr2)
            parity_type_holder2.append(frame_type)
        for frame, parity in zip(C_received_frames,
parity_type_holder2):
            client_socket.send(frame.encode())
            self.text_area5.insert(tk.END, f"\nReceived frames:
{frame}\n")
            self.text_area6.insert(tk.END, f"\nParity of frames:
{parity}\n")
        with open("p2.txt", 'w') as file:
            for i in parity_type_holder2:
                file.write(i + '\n')


        client_socket.close()
        self.create_receiver_window(C_received_frames,
parity_type_holder2)
    def create_receiver_window(self, received_frames, parity_types):
            receiver_window = tk.Toplevel(self.master)
            receiver_window.title("Receiver's GUI")
            prompt_label_received_frames1 = tk.Label(receiver_window,
text="Received Frames")
            prompt_label_received_frames1.pack(pady=5)
            text_area_received_frames1 =
scrolledtext.ScrolledText(receiver_window, wrap=tk.WORD, width=40,
height=10)
            text_area_received_frames1.pack(padx=10, pady=10)
```

```
            prompt_label_received_parity2 = tk.Label(receiver_window,
text="Received Parity")
            prompt_label_received_parity2.pack(pady=5)
            text_area_received_parity2 =
scrolledtext.ScrolledText(receiver_window, wrap=tk.WORD, width=40,
height=10)
            text_area_received_parity2.pack(padx=10, pady=10)
            for frame, parity in zip(received_frames, parity_types):
                text_area_received_frames1.insert(tk.END, f"\nReceived
frame: {frame}\n")
                text_area_received_parity2.insert(tk.END, f"\nReceived
parity: {parity}\n")
            content1 = self.read_file("p1.txt")
            content2 = self.read_file("p2.txt")
            if content1 == content2:
                result_message = "Message Received successfully"
            else:
                result_message = "Message Corrupted, please retransmit"
            # Display the result in the receiver's GUI
            result_label = tk.Label(receiver_window, text=f"Message
accuracy: {result_message}")
            result_label.pack(pady=10)
            self.text_area_msg.insert(tk.END, f"\nMessage accuracy:
{result_message}\n")
if __name__ == "__main__":
    root = tk.Tk()
    app = MultiGUI(root)
    root.mainloop()
```

Output:


Sender's GUI
Type your message:
Hello world!
Connect
Message converted to bit frames
Frames divided into 32 bit frames
['10010001100101110110011011001101111100
00011101111101111',
'11100101101100110010010000100000000000000
0000']
Message Sent
Message sent > > Hello world!
Frames Sent
Sent frames:
10010001100101110110011011001101111100000
011101111101111
Sent frames:
11100101101100110010010000100000000000000
000
Frame Parity
Parity of frames: Odd
Parity of frames: Odd
Receiver's GUI
Received Frames
Received frame:
10010001100101110110011011001101111100000
011101111101111
Received frame:
11100101101100110010010000100000000000000
000
Received Parity
Received parity: Odd
Received parity: Odd
Message accuracy: Message Received successfully
TERMINAL
SERIAL MONITOR
ject\py> python -u "d:\DESKTOP\Sem 5\CIE 3\CN Pr
01111100000111011111011111110010110110011001001
00110010010000100000000000000001110010110110011
01100101110110011011001101111100000111011111011
000000000000']
32
ject\py>

Conclusion :

The final analysis of various detection methods provides us a comprehensive overview of Parity checker , Hamming distance and Cyclic Redundancy Check (CRC) as error detection techniques.helps us understand the concepts of Parity checker, Hamming distance, its applications, properties, and comparison with other error detection techniques. Additionally, helps us understand and explore CRC, its working principle, properties, and applications.