



Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology, Pune-37

(An autonomous Institute of Savitribai Phule Pune University)

Department of Computer Engineering

Assignment no. 3

Division and Batch	CS-D and D1
Year and Branch	TY-CS
Subject	Computer Network Lab
GR-no	12220206
Roll no	44
Name	Vedika Sontakke

Problem Statement: Write a program for error detection and correction for 7/8 bits ASCII codes using Hamming Codes or CRC. Demonstrate the packets captured traces using Wireshark Packet Analyzer Tool for peer to peer mode.

Introduction :

CRC stands for "Cyclic Redundancy Check," and it is a technique used in digital communication and data storage to detect errors in data transmission or storage. It's a common method for ensuring data integrity by appending a checksum to the data, which allows the receiver to verify if the data has been corrupted during transmission or storage.

Data Representation: The data to be transmitted or stored is considered as a stream of bits (0s and 1s). This data is treated as a polynomial, where the coefficients of the polynomial are the bits of the data. For example, the binary data 101101 can be represented as the polynomial $x^5 + x^4 + x^2 + x^0$.

Generating Polynomial: A fixed polynomial, called the "generator polynomial" or "divisor," is selected. The degree of the generator polynomial determines the length of the CRC code. The sender and receiver must agree on this polynomial beforehand. Common generator polynomials include CRC-16 (16 bits) and CRC-32 (32 bits).

Applying CRC Calculation: The sender performs polynomial division between the data polynomial and the generator polynomial using a modulo-2 division (binary division without carries or borrows). This division generates a remainder polynomial.

Checksum Generation: The remainder polynomial obtained from the division is the CRC checksum. This checksum is appended to the original data before transmission or storage. The combined data and checksum are often referred to as the "frame."

Receiver Check: Upon receiving the frame, the receiver also calculates the CRC checksum using the same generator polynomial. If the calculated checksum matches the checksum sent by the sender, it indicates that the data was likely transmitted or stored without errors. If there is a mismatch, it suggests that the data might have been corrupted during transmission or storage.

CRC is widely used due to its simplicity and effectiveness in detecting common types of errors, such as single-bit and burst errors. However, it's important to note that CRC is not foolproof and may not detect all types of errors, especially more complex errors. For stronger error detection and correction capabilities, more advanced techniques like Reed-Solomon codes or Hamming codes are used.

In networking protocols, storage systems, and various communication interfaces (such as Ethernet and USB), CRC is often used to ensure the reliability of data transmission and storage by providing a quick and efficient method for error detection.

Code with Socket CRC: {Sender}

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>

#pragma comment(lib, "ws2_32.lib")

int calculateCRC8(unsigned char *data, int length)
{
    int crc = 0;
    int poly = 0x07; // CRC-8 polynomial

    for (int i = 0; i < length; i++)
    {
        crc ^= data[i]; // XOR with the byte

        for (int j = 0; j < 8; j++)
        {
            if ((crc & 1) != 0)
            {
                crc = (crc >> 1) ^ poly;
            }
            else
            {
                crc = crc >> 1;
            }
        }
    }

    return crc & 0xFF;
}
```

```

int main()
{
    WSADATA wsaData;
    SOCKET clientSocket;
    struct sockaddr_in serverAddr;
    int addrSize = sizeof(serverAddr);

    char buffer[100];

    // Initialize Winsock
    WSStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);

    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP address
    serverAddr.sin_port = htons(12345); // Server port

    printf("Enter data to send: ");
    scanf("%s", buffer);

    int dataLength = strlen(buffer);
    int crc = calculateCRC8((unsigned char *)buffer, dataLength);

    // Append the CRC value to the buffer
    buffer[dataLength] = crc;

    // Send data (including CRC) to the server
    sendto(clientSocket, buffer, dataLength + 1, 0, (struct sockaddr
*)&serverAddr, addrSize);

    closesocket(clientSocket);
    WSACleanup();

    return 0;
}

```

Code with Socket CRC: {Receiver}

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock2.h>

#pragma comment(lib, "ws2_32.lib")

int calculateCRC8(unsigned char *data, int length)
{
    int crc = 0;
    int poly = 0x07; // CRC-8 polynomial

    for (int i = 0; i < length; i++)
    {
        crc ^= data[i]; // XOR with the byte

        for (int j = 0; j < 8; j++)
        {
            if ((crc & 1) != 0)
            {
                crc = (crc >> 1) ^ poly;
            }
            else
            {
                crc = crc >> 1;
            }
        }
    }

    return crc & 0xFF;
}

int main()
{
    WSADATA wsaData;
    SOCKET serverSocket, clientSocket;
    struct sockaddr_in serverAddr, clientAddr;
    int addrSize = sizeof(clientAddr);

    char buffer[100];

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create socket
    serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

```
memset(&serverAddr, 0, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_addr.s_addr = INADDR_ANY;
serverAddr.sin_port = htons(12345);

// Bind the socket
bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr));

printf("Server listening...\n");

recvfrom(serverSocket, buffer, sizeof(buffer), 0, (struct sockaddr
*)&clientAddr, &addrSize);

int dataLength = strlen(buffer);
int receivedCRC = buffer[dataLength]; // Extract received CRC

buffer[dataLength] = '\0'; // Remove CRC from the buffer

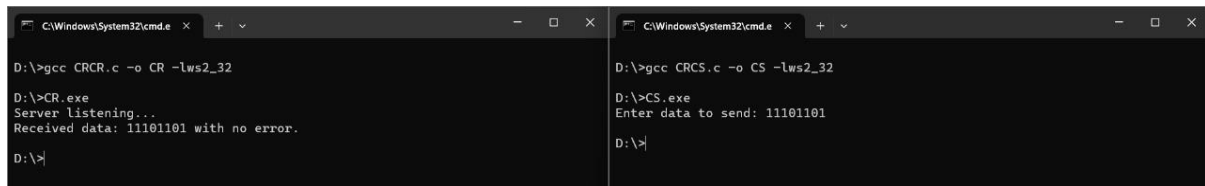
int calculatedCRC = calculateCRC8((unsigned char *)buffer, dataLength);

if (receivedCRC == calculatedCRC)
{
    printf("Received data: %s\n", buffer);
}
else
{
    printf("Error detected\n");
}

closesocket(serverSocket);
WSACleanup();

return 0;
}
```

Output with Error CRC:



```
C:\Windows\System32\cmd.e x + v
D:\>gcc CRCR.c -o CR -lws2_32
D:\>CR.exe
Server listening...
Received data: 11101101 with no error.
D:\>

C:\Windows\System32\cmd.e x + v
D:\>gcc CRCs.c -o CS -lws2_32
D:\>CS.exe
Enter data to send: 11101101
D:\>
```

Hamming Code:-

Hamming code is an error-correcting code that is used to detect and correct errors in data transmission or storage. It's a more advanced technique compared to CRC and provides not only error detection but also correction capabilities. Hamming code is particularly effective at correcting single-bit errors.

Data Representation: Similar to CRC, the data to be transmitted or stored is considered as a stream of bits (0s and 1s). The bits are positioned in such a way that each bit has a specific position that is a power of 2 (1, 2, 4, 8, 16, etc.). These positions are reserved for parity bits.

Parity Bits: Parity bits are additional bits added to the data to create a structured pattern. Each parity bit is responsible for checking a specific set of data bits. The positions of the parity bits are determined by their binary representation. For example, the first parity bit checks every bit that has a 1 in the least significant (rightmost) bit of its position (bit 1, 3, 5, 7, etc.). The second parity bit checks every bit that has a 1 in the second least significant bit of its position (bit 2, 3, 6, 7, 10, 11, etc.), and so on.

Error Correction: When the data is received, the receiver calculates the parity bits based on the received data. If any of the parity bits do not match the calculated parity, an error is detected. The position of the incorrect parity bit provides information about the position of the erroneous bit.

Correction Mechanism: Hamming code can also correct single-bit errors. If an error is detected in a parity bit, the receiver can determine which data bit

caused the error and correct it. This is done by flipping the erroneous bit, effectively correcting the error.

Hamming code is effective for correcting single-bit errors and detecting double-bit errors within a certain range. However, it is less efficient at detecting and correcting multiple errors or burst errors compared to more advanced error-correcting codes like Reed-Solomon codes.

Code with Socket Hamming Code: {Sender}

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>

void calculateParity(int data[4], int parity[3])
{
    parity[0] = data[0] ^ data[1] ^ data[3];
    parity[1] = data[0] ^ data[2] ^ data[3];
    parity[2] = data[1] ^ data[2] ^ data[3];
}

int main()
{
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        perror("WSAStartup failed");
        return 1;
    }

    SOCKET sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == INVALID_SOCKET)
    {
        perror("Socket creation failed");
        WSACleanup();
        return 1;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(12345);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```



```

    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr))
== SOCKET_ERROR)
    {
        perror("Connection failed");
        closesocket(sockfd);
        WSACleanup();
        return 1;
    }

    printf("Enter a 4-bit message: ");
    char m[5];
    scanf("%4s", m);

    int d0 = m[3] - '0';
    int d1 = m[2] - '0';
    int d2 = m[1] - '0';
    int d3 = m[0] - '0';

    int data[4] = {d3, d2, d1, d0};
    int parity[3];
    calculateParity(data, parity);

    int hammingMsg[7];
    hammingMsg[0] = d3;
    hammingMsg[1] = d2;
    hammingMsg[2] = d1;
    hammingMsg[3] = parity[2];
    hammingMsg[4] = d0;
    hammingMsg[5] = parity[1];
    hammingMsg[6] = parity[0];

    printf("\nMessage with Hamming code: ");
    for (int i = 0; i < 7; i++)
    {
        printf("%d ", hammingMsg[i]);
    }
    printf("\n");

    send(sockfd, (char *)hammingMsg, sizeof(hammingMsg), 0);

    closesocket(sockfd);
    WSACleanup();

    return 0;
}

```

Code with Socket Hamming Code: {Receiver}

```
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>

void calculateParity(int data[4], int parity[3])
{
    parity[0] = data[0] ^ data[1] ^ data[3];
    parity[1] = data[0] ^ data[2] ^ data[3];
    parity[2] = data[1] ^ data[2] ^ data[3];
}

int main()
{
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        perror("WSAStartup failed");
        return 1;
    }

    SOCKET server_socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket_fd == INVALID_SOCKET)
    {
        perror("Socket creation failed");
        WSACleanup();
        return 1;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(12345);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(server_socket_fd, (struct sockaddr *)&server_addr,
    sizeof(server_addr)) == SOCKET_ERROR)
    {
        perror("Binding failed");
        closesocket(server_socket_fd);
        WSACleanup();
        return 1;
    }

    if (listen(server_socket_fd, 1) == SOCKET_ERROR)
    {
        perror("Listening failed");
    }
}
```

```

        closesocket(server_socket_fd);
        WSACleanup();
        return 1;
    }

    struct sockaddr_in client_addr;
    int client_len = sizeof(client_addr);
    SOCKET socket_fd = accept(server_socket_fd, (struct sockaddr
*)&client_addr, &client_len);
    if (socket_fd == INVALID_SOCKET)
    {
        perror("Accepting connection failed");
        closesocket(server_socket_fd);
        WSACleanup();
        return 1;
    }

    int receivedMsg[7];
    recv(socket_fd, (char *)receivedMsg, sizeof(receivedMsg), 0);

    int d3 = receivedMsg[0];
    int d2 = receivedMsg[1];
    int d1 = receivedMsg[2];
    int p2 = receivedMsg[3];
    int d0 = receivedMsg[4];
    int p1 = receivedMsg[5];
    int p0 = receivedMsg[6];

    int receivedData[4] = {d3, d2, d1, d0};
    int receivedParity[3] = {p0, p1, p2};

    int calculatedParity[3];
    calculateParity(receivedData, calculatedParity);

    int errorBit = 0;
    for (int i = 0; i < 3; i++)
    {
        if (receivedParity[i] != calculatedParity[i])
        {
            errorBit += (1 << i);
        }
    }

    if (errorBit > 0)
    {
        printf("Error detected at bit position: %d\n", errorBit);
        receivedData[errorBit - 1] ^= 1;
    }

```

```

else
{
    printf("No Error Detected.");
    printf("\nReceived message: %d %d %d %d\n", receivedData[0],
receivedData[1], receivedData[2], receivedData[3]);
}

closesocket(socket_fd);
closesocket(server_socket_fd);
WSACleanup();

return 0;
}

```

Output Hamming Code:

```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

D:\>gcc HammingS.c -o HS -lws2_32

D:\>HS.exe
Enter a 4-bit message: 1001

Message with Hamming code: 1 0 0 1 1 0 0

D:\>

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

D:\>gcc HammingR.c -o HR -lws2_32

D:\>HR.exe
No Error Detected.
Received message: 1 0 0 1

D:\>

```

Wireshark:

Wireshark is a widely used open-source packet analyzer tool that allows users to capture, analyze, and inspect the network traffic flowing over a computer network. It is commonly used by network administrators, security professionals, and developers to diagnose network issues, troubleshoot problems, and analyze the behavior of network protocols.

Packet Capture: Wireshark captures packets of data as they travel across a network interface. It can capture data from various sources, such as Ethernet, Wi-Fi, and loopback interfaces. This is often done by putting the network interface into "promiscuous mode," allowing it to capture all network traffic, not just the traffic intended for the specific machine.

Live Capture and Offline Analysis: Wireshark can capture packets in real-time from the network or analyze previously captured packet trace files (usually saved with a ".pcap" extension). This allows users to examine network traffic over a specified time period.

Display and Filtering: Wireshark provides a graphical user interface that displays captured packets in a comprehensive and organized manner. Users can apply various filters to display specific types of traffic, such as packets from a particular source or destination, specific protocols, or packets with specific characteristics.

Packet Analysis: Once packets are captured and displayed, Wireshark provides extensive tools to analyze them. Users can expand packet headers to view details of different network protocols, including Ethernet, IP, TCP, UDP, HTTP, DNS, and more. This helps in understanding the communication patterns and potential issues within the network.

Color Coding and Marking: Wireshark color codes packets based on various criteria, making it easier to identify different types of packets and potential problems. For example, packets with errors or anomalies might be highlighted in a different color.

Packet Decoding: Wireshark decodes packet data, allowing users to see the information contained within different layers of the protocol stack. This is particularly useful for understanding the content and structure of application-layer data.

Statistics and Graphs: Wireshark provides statistical tools and graphical representations of network traffic, helping users understand trends and patterns in the data. This includes information like packet counts, data rates, and more.

Exporting Data: Users can export captured data or analysis results in various formats for further investigation or reporting. This can be useful for sharing information with colleagues or external parties.

Plugin Support: Wireshark supports a wide range of plugins that can extend its functionality. These plugins can provide additional dissectors for specific protocols, enhanced analysis tools, and more.

Wireshark is an invaluable tool for diagnosing network issues, debugging protocol implementations, monitoring network security, and understanding the behavior of various network applications. However, it's important to note that Wireshark should be used responsibly and ethically, as capturing and analyzing network traffic can potentially involve sensitive information.

Wireshark Output:

Capturing from Adapter for loopback traffic capture

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter: <Ctrl-F>

No.	Time	Source	Destination	Protocol	Length	Info
8	20.298028	127.0.0.1	127.0.0.1	TCP	53	57399 → 57379 [PSH, ACK] Seq=1 Ack=1 Win=8410 Len=0
9	20.298048	127.0.0.1	127.0.0.1	TCP	44	57379 → 57399 [ACK] Seq=1 Ack=10 Win=8419 Len=0
10	20.298058	127.0.0.1	127.0.0.1	TCP	47	57379 → 57399 [PSH, ACK] Seq=1 Ack=10 Win=8419 Len=3
11	20.298073	127.0.0.1	127.0.0.1	TCP	44	57399 → 57379 [ACK] Seq=10 Ack=4 Win=8410 Len=0
12	35.626601	127.0.0.1	127.0.0.1	TCP	56	61940 → 12345 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
13	35.626648	127.0.0.1	127.0.0.1	TCP	56	12345 → 61940 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
14	35.626674	127.0.0.1	127.0.0.1	TCP	44	61940 → 12345 [ACK] Seq=1 Ack=1 Win=327424 Len=0
15	39.662634	127.0.0.1	127.0.0.1	TCP	72	61940 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=28
16	39.662664	127.0.0.1	127.0.0.1	TCP	44	12345 → 61940 [ACK] Seq=1 Ack=29 Win=2161152 Len=0
17	39.662681	127.0.0.1	127.0.0.1	TCP	44	61940 → 12345 [FIN, ACK] Seq=29 Ack=1 Win=327424 Len=0
18	39.662687	127.0.0.1	127.0.0.1	TCP	44	12345 → 61940 [ACK] Seq=1 Ack=30 Win=2161152 Len=0
19	39.667405	127.0.0.1	127.0.0.1	TCP	44	12345 → 61940 [FIN, ACK] Seq=1 Ack=30 Win=2161152 Len=0
20	39.667423	127.0.0.1	127.0.0.1	TCP	44	61940 → 12345 [ACK] Seq=30 Ack=2 Win=327424 Len=0
21	45.305374	127.0.0.1	127.0.0.1	TCP	53	57399 → 57379 [PSH, ACK] Seq=10 Ack=4 Win=8410 Len=9
22	45.305432	127.0.0.1	127.0.0.1	TCP	44	57379 → 57399 [ACK] Seq=4 Ack=19 Win=8419 Len=0
23	45.306699	127.0.0.1	127.0.0.1	TCP	47	57379 → 57399 [PSH, ACK] Seq=4 Ack=19 Win=8419 Len=3
24	45.306721	127.0.0.1	127.0.0.1	TCP	44	57399 → 57379 [ACK] Seq=19 Ack=7 Win=8410 Len=0

> Frame 1: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface \Device\NPF_{...} {...}

> Null/Loopback

> Internet Protocol Version 4, Src: 192.168.1.11, Dst: 224.0.0.251

> User Datagram Protocol, Src Port: 5353, Dst Port: 5353

> Multicast Domain Name System (query)

0000 02 00 00 00 45 00 00 50 aa 3b 00 00 01 11 00 00E..P.....

0010 c0 a8 01 0b e0 00 00 fb 14 e9 14 e9 00 3c 72 ecCr.....

0020 00 00 00 00 01 00 00 00 00 00 00 00 10 56 01 72m.....

0030 61 64 55 70 6c 61 6e 63 68 59 77 61 72 06 5f 64aduplanc hbar...s

0040 6f 73 76 63 04 5f 74 63 70 05 6c 6f 63 61 6c 06osvc- tc p-local

0050 00 ff 00 01

Adapter for loopback traffic capture: <live capture in progress>

Packets: 24 · Displayed: 24 (100.0%)

Profiles: Default