COMPUTER NETWORKS LAB
CS208
LAB FILE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBMITTED TO
Prof. Rahul Karaya
CSE Dept.

SUBMITTED BY:
Piyus Kumar
24/CSE/326

Index Page

S. NO.	OBJECTIVE	DATE	SIGN
01			

Program 1: Bit, Byte and Char Stuffing

AIM

To implement Bit,Byte and Character Stuffing in Computer Networks.

THEORY

Introduction:

In Computer Networks, data is transmitted in the form of frames at the Data Link Layer. Each frame has a header, data, and trailer. To identify the start and end of a frame, frame delimiters are used. If the same delimiter appears inside the data, the receiver may misinterpret it as a frame boundary, causing errors. To avoid this, stuffing techniques are used. Stuffing adds extra bits or characters in the data so that delimiters do not appear inside it. At the receiver side, these extra bits or characters are removed using destuffing, restoring the original data.

1.Bit Stuffing:

Bit stuffing is used in bit-oriented protocols such as HDLC. A special flag pattern 01111110 is used to mark frame boundaries. If this pattern appears in the data, framing errors may occur.

To prevent this, whenever five consecutive 1s appear in the data, a 0 is inserted after the fifth 1 at the sender side. At the receiver side, after detecting five 1s, the next 0 is removed during destuffing.

Example:

Data: 1111101111110

After bit stuffing: 1111100 11111010

Bit stuffing is efficient and suitable for high-speed communication but slightly increases frame size.

2. Byte Stuffing

Byte stuffing is used in byte-oriented protocols like PPP. Special bytes such as FLAG (0x7E) and ESC (0x7D) are used as frame delimiters. If these bytes appear in data, confusion may occur.

To avoid this, an ESC byte is added before the control byte, and the data byte is XORed with 0x20. At the receiver, ESC is removed and XOR is applied again to recover the original data.

Example:

Data: 45 7E 23 7D 11
After byte stuffing: 45 7D 5E 23 7D 5D 11

3. Character Stuffing

Character stuffing is used in character-oriented protocols. Special characters such as STX, ETX, and ESC mark frame boundaries. If these appear in data, errors may occur.

To solve this, an ESC character is inserted before any STX, ETX, or ESC present in the data. The receiver removes the ESC during destuffing.

Example:
Data: A<B>C/D
After character stuffing: <A/<B/>C//D>

CODE: Bit Stuffing

```
#include <iostream> #include <string> using
namespace std; string bitStuffing(string s) {
string a = "";     int count = 0;
   for (int i = 0; i < s.length(); i++) {         if (s[i] == '1') {
count++;             a += s[i];         } else {             count = 0;
a += s[i];
       }
         if (count == 5) {             a += '0';             count =
0;
       }
   }
```

```
    return a;
}

string bitDestuffing(string s) {     string a =
"";     int count = 0;
    for (int i = 0; i < s.length(); i++) {         if
(s[i] == '1') {             count++;             a +=
s[i];         } else {
            if (count == 5) {
count    =          0;                    continue;
            }
             count = 0;             a +=
s[i];
        }     }     return a;
}
int      main()      {
string s;
     cout << "Enter string: ";     cin >>
s;
    string stuffed = bitStuffing(s);
    cout << "After Bit Stuffing: " << stuffed << endl;     string
destuffed = bitDestuffing(stuffed);
        cout << "After Bit De-stuffing: " << destuffed << endl;
return 0;
}
```

OUTPUT:

```
PS C:\Users\gautam2401\OneDrive\Documents\CN> cd "c:\Users\gautam2401\OneDrive\Documents\CN\" ; if ($?) { g++ bitstuffing.cp
 -o bitstuffing } ; if ($?) { .\bitstuffing }
Enter string: 10111110111
After Bit Stuffing: 101111100111
After Bit De-stuffing: 10111110111
```

CODE : Byte Stuffing

```
#include <iostream>
using namespace std;
string byteStuffing(string data) {
    string result = "";     char FLAG
= 'F';     char ESC  = 'E';     result
+= FLAG;       for (char ch : data) {
if (ch == FLAG || ch == ESC) {
            result += ESC;  }
result += ch;
    }
    result += FLAG;
return result;} int main()
{     string
data;
    cout << "Enter data: ";
cin   >>   data;                  string   stuffed   =
byteStuffing(data);       cout << "Stuffed data:
" << stuffed << endl;     return 0;
}
```

OUTPUT:

```
PS C:\Users\gautam2401\OneDrive\Documents\CN> cd "c:\Users\gautam2401\OneDrive\Documents\CN\" ; if ($?) { g++ bytestuff.cpp
o bytestuff } ; if ($?) { .\bytestuff }
Enter data: FILEHELL
Stuffed data: FEFILEEHEELLF
```

CODE : Character Stuffing

```
#include <iostream>
#include <string>
using namespace std;
string characterStuffing(string data) {
```

```
    char flag = 'F';    char esc = 'E';    string final = "";    final += flag;       for (int i
= 0; i < data.length(); i++) {        if (data[i] == flag || data[i] == esc) {            final
+= esc;            }          final += data[i];
   }     final += flag;       return
final;
}

int main() {    string data;    cout << "Enter data: ";    cin >> data;     string result =
characterStuffing(data);    cout << "After stuffing: " << result << endl;    return 0;
}
```

OUTPUT:

```
PS C:\Users\gautam2401\OneDrive\Documents\CN> cd "c:\Users\gautam2401\OneDrive\Documents\CN\" ; if ($?) { g++ characterstuffi
ng.cpp -o characterstuffing } ; if ($?) { .\characterstuffing }
Enter data: HELLOFM
After stuffing: FHEELLOEFMF
```

Program 2: Study and Implement CRC

AIM

To study and implement cycle redundancy check.

THEORY

Introduction:

Cyclic Redundancy Check (CRC) is an error detection technique used in computer networks at the Data Link Layer to ensure reliable data transmission. Errors may occur during transmission due to noise or interference, and CRC helps in detecting such errors efficiently. In CRC, the data is treated as a binary polynomial and divided by a predefined generator polynomial using modulo-2 (XOR) division. The remainder obtained from this division is called the CRC bits. These CRC bits are appended to the original data to form the transmitted frame.

At the receiver side, the received frame is divided by the same generator polynomial. If the remainder is zero, the data is considered error-free; otherwise, an error is detected. CRC is widely used in protocols like Ethernet and HDLC because it can detect single-bit errors, multiple-bit errors, and burst errors with high accuracy.

CODE:

```
#include <iostream>
#include <string>
using namespace std;

string xorDiv(string a, string b) {
string result = "";      for (int i = 1; i
< b.length(); i++) {         result +=
(a[i] == b[i]) ? '0' : '1';
    }      return
result;
}  string crc(string data, string gen) {
int genLen = gen.length();     string
temp = data.substr(0, genLen);     int
i
= genLen;

    while (i <= data.length()) {
if (temp[0] == '1')              temp =
xorDiv(temp, gen);         else
            temp = xorDiv(temp, string(genLen, '0'));
                        if   (i   <
data.length())
temp         +=         data[i];
i++;
    }
    return temp;
}

int main() {     string data, gen;     cout << "Enter data: ";
cin >> data;     cout << "Enter generator: ";     cin >>
gen;     string paddedData = data + string(gen.length() -
1, '0');     string remainder = crc(paddedData, gen);
string transmitted = data + remainder;     cout << "CRC
bits: " << remainder << endl;     cout << "Transmitted
frame: " << transmitted << endl;     string check =
crc(transmitted, gen);     if (check.find('1') ==
string::npos)
        cout << "No Error Detected" << endl;     else
cout << "Error Detected" << endl;

    return 0;
}
```

OUTPUT

```
PS C:\Users\gautam2401\OneDrive\Documents\CN> cd "c:\Users\gautam2401\OneDrive\Documents\CN\" ; if ($?) { g++ crc.cpp -o crc
} ; if ($?) { .\crc }
Enter data: 1011111
Enter generator: 1110
CRC bits: 010
Transmitted frame: 1011111010
No Error Detected
```

Program 3: Stop and Wait

AIM

Study and implement stop and wait.

THEORY

Introduction:

Stop-and-Wait is a simple **flow control and error control protocol** used at the **Data Link Layer**. In this protocol, the sender transmits **one frame at a time** and then **waits for an acknowledgment (ACK)** from the receiver before sending the next frame.

After sending a frame, the sender starts a **timer**. If the acknowledgment is received before the timer expires, the sender sends the next frame. If the ACK is not received within the timeout period, the sender assumes the frame or ACK is lost and **retransmits the same frame**.

To avoid duplicate frames, **sequence numbers (0 and 1)** are used. The receiver sends an ACK containing the expected sequence number. Stop-and-Wait is easy to implement but is **inefficient for high-speed networks** because the sender remains idle while waiting for acknowledgments.

CODE:

```
#include <iostream>
#include <cstdlib>
#include <ctime> using
namespace std;

int main() {
srand(time(0));
    int frame = 0;
    const int TIMEOUT = 5;

    for (int i = 0; i < 5; ) {
    cout << "\nSender: Sending Frame " << frame << endl;
    int ack_time = rand() % 10;
    cout << "ACK arrival time: " << ack_time << endl;
    if (ack_time > TIMEOUT) {
            cout << "Sender: Timeout occurred!" << endl;
            cout << "Sender: Resending Frame " << frame << endl;
        } else {
            cout << "Receiver: ACK received for Frame " << frame << endl;
.           frame = 1 - frame;
            i++;
        }
    }

    cout << "\nTransmission completed successfully!" << endl;
    return 0;
}
```

OUTPUT

```
PS C:\Users\gautam2401\OneDrive\Documents\CN> cd "c:\Users\gautam2401\OneDrive\Documents\CN\" ; if ($?) { g++ stopnwait.cpp -
o stopnwait } ; if ($?) { .\stopnwait }

Sender: Sending Frame 0
ACK arrival time: 9
Sender: Timeout occurred!
Sender: Resending Frame 0

Sender: Sending Frame 0
ACK arrival time: 4
Receiver: ACK received for Frame 0

Sender: Sending Frame 1
ACK arrival time: 4
Receiver: ACK received for Frame 1

Transmission completed successfully!
PS C:\Users\gautam2401\OneDrive\Documents\CN>
```

Program 4: Go-Back-N ARQ

AIM

To study and implement the Go-Back-N Automatic Repeat Request (ARQ) protocol.

THEORY

Introduction:

Go-Back-N ARQ is an **error control and flow control protocol** used at the **Data Link Layer**. It is an improvement over the Stop-and-Wait protocol, allowing the sender to transmit **multiple frames** without waiting for an acknowledgment for each one.

The protocol uses a **sliding window** of size *N* at the sender, representing the maximum number of unacknowledged frames that can be sent. The receiver only sends a **cumulative acknowledgment (ACK)** for the next frame it expects to receive.

If a frame is lost or corrupted, the receiver discards that frame and all subsequent out-of-sequence frames, sending an ACK for the last correctly received frame. The sender, upon timing out, **retransmits the lost frame and all subsequent frames** that were already sent (the "Go-Back-N" mechanism). This makes it more efficient than Stop-and-Wait, especially in high-bandwidth-delay product networks, but less efficient than Selective Repeat ARQ as it retransmits more data than necessary.

CODE:

```
#include <iostream>
using namespace std;

int main() {
    int totalFrames, windowSize;
    cout << "Enter total number of frames: ";
    cin >> totalFrames;
    cout << "Enter window size: ";
    cin >> windowSize;

    int currentFrame = 1;
    while (currentFrame <= totalFrames) {
        for (int i = 0; i < windowSize && (currentFrame + i) <= totalFrames; ++i) {
            cout << "Sending Frame " << (currentFrame + i) << endl;
        }

        int lastAck;
```

```
        cout << "Enter the last frame successfully received: ";
        cin >> lastAck;

        if (lastAck >= currentFrame) {
            currentFrame = lastAck + 1;
        } else {
            cout << "Timeout or Error! Retransmitting window starting from frame " <<
currentFrame << "..." << endl;
        }
    }

    cout << "All frames sent and acknowledged successfully." << endl;
    return 0;
}
```

OUTPUT

Output Clear

```
Enter total number of frames: 10
Enter window size: 4
Sending Frame 1
Sending Frame 2
Sending Frame 3
Sending Frame 4
Enter the last frame successfully received: 4
Sending Frame 5
Sending Frame 6
Sending Frame 7
Sending Frame 8
Enter the last frame successfully received: 6
Sending Frame 7
Sending Frame 8
Sending Frame 9
Sending Frame 10
Enter the last frame successfully received: 10
All frames sent and acknowledged successfully.


=== Code Execution Successful ===
```