# CRYPTOGRAPHY AND NETWORK SECURITY LAB

**Week - 1 Stream Ciphers**

**1.1) Write a C Program to implement Shift Cipher.**

**AIM:** To write a C Program to implement Shift Cipher.

**DESCRIPTION:**

The Caesar cipher is the simplest and oldest method of cryptography. The Caesar cipher method is based on a mono-alphabetic cipher and is also called a shift cipher or additive cipher. Julius Caesar used the shift cipher (additive cipher) technique to communicate with his officers. For this reason, the shift cipher technique is called the Caesar cipher. The Caesar cipher is a kind of replacement (substitution) cipher, where all letter of plain text is replaced by another letter.A Caesar cipher is a weak method of cryptography. It can be easily hacked. It means the message encrypted by this method can be easily decrypted.

**Plaintext:** It is a simple message written by the user.

**Ciphertext:** It is an encrypted message after applying some technique.

The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down.

The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as.

(Encryption Phase with shift n)     $En(x) = (x+n) \bmod 26$

(Decryption Phase with shift n)     $Dn(x) = (x-n) \bmod 26$

If any case (Dn) value becomes negative (-ve), in this case, we will add 26 in the negative value.

**Where,**

E denotes the encryption

D denotes the decryption

x denotes the letters value

n denotes the key value (shift value)

**PROGRAM:**

```c
#include<stdio.h>
#include<ctype.h>
int main()
{
        char text[500], ch;
        int key;
        printf("Enter a message to decrypt: ");
        scanf("%s", text);
        printf("Enter the key: ");
        scanf("%d", & key);
        for (int i = 0; text[i] != '\0'; ++i)
        {
        ch = text[i];
        if (isalnum(ch))
        {
                if (islower(ch))
                {
                ch = (ch - 'a' - key + 26) % 26 + 'a';
                }
                if (isupper(ch))
                {
                ch = (ch - 'A' - key + 26) % 26 + 'A';
                }
                if (isdigit(ch))
                {
                ch = (ch - '0' - key + 10) % 10 + '0';
                }
```
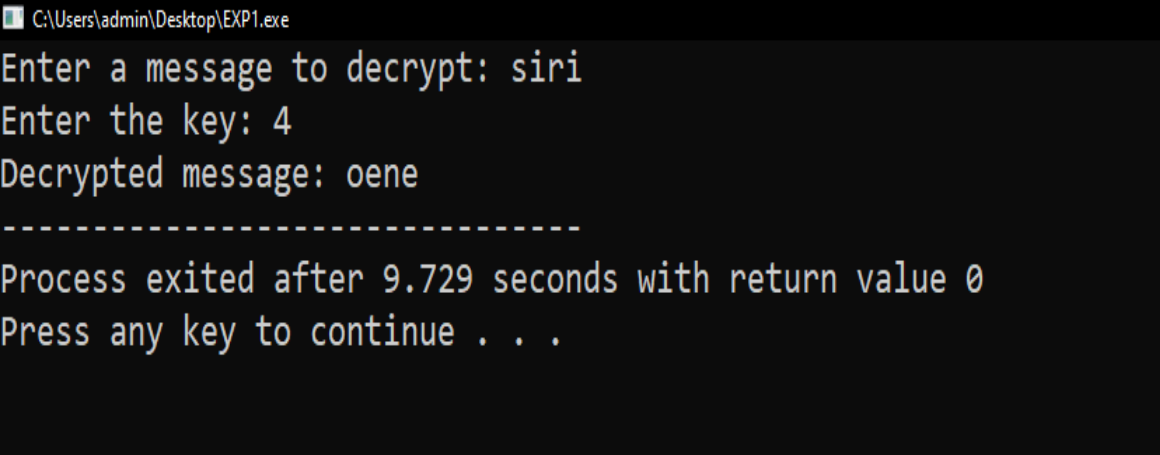
```
        }

        else

        {

                printf("Invalid Message");

        }

        text[i] = ch;

        }

printf("Decrypted message: %s", text);

  return 0;

}
```

**OUTPUT:**

```
C:\Users\admin\Desktop\EXP1.exe
Enter a message to decrypt: siri
Enter the key: 4
Decrypted message: oene
---------------------------------
Process exited after 9.729 seconds with return value 0
Press any key to continue . . .
```

**RESULT:** Thus the implementation of Caesar cipher had been executed successfully

**1.2) Write a C Program to implement Mono-Alphabetic Substitution Cipher.**

**AIM:** To write a C Program to implement Mono-Alphabetic Substitution Cipher.

**DESCRIPTION:**

The substitution cipher is the oldest forms of encryption algorithms according to creates each character of a plaintext message and require a substitution process to restore it with a new character in the ciphertext. This substitution method is deterministic and reversible, enabling the intended message recipients to reverse-substitute ciphertext characters to retrieve the plaintext.

The specific form of substitution cipher is the Monoalphabetic Substitution Cipher, is known as "Simple Substitution Cipher". Monoalphabetic Substitution Ciphers based on an individual key mapping function K, which consistently replaces a specific character α with a character from the mapping K (α).

A monoalphabetic cipher is any cipher in which the letters of the plain text are mapped to cipher text letters based on a single alphabetic key. Examples of monoalphabetic ciphers would include the Caesar-shift cipher, where each letter is shifted based on a numeric key, and the atbash cipher, where each letter is mapped to the letter symmetric to it about the center of the alphabet.

Monoalphabetic cipher is one where each symbol in plain text is mapped to a fixed symbol in cipher text. The relationship between a character in the plain text and the characters in the cipher text is one-to-one. Each alphabetic character of plain text is mapped onto a unique alphabetic character of a cipher text.

A stream cipher is a monoalphabetic cipher if the value of key does not depend on the position of the plain text character in the plain text stream. It includes additive, multiplicative, affine and monoalphabetic substitution cipher. Monoalphabetic Cipher is described as a substitution cipher in which the same fixed mappings from plain text to cipher letters across the entire text are used. Monoalphabetic ciphers are not that strong as compared to polyalphabetic cipher.

Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process. For example, if 'A' is encrypted as 'D', for any number of occurrence in that plaintext, 'A' will always get encrypted to 'D'.

**PROGRAM:**

```c
#include<stdio.h>

char monocipher_encr(char);

char alpha[27][3] = { { 'a', 'f' }, { 'b', 'a' }, { 'c', 'g' }, { 'd', 'u' }, {'e', 'n' }, { 'f', 'i' }, { 'g', 'j' }, { 'h', 'k' }, { 'i', 'l' },

{'j', 'm' }, { 'k', 'o' }, { 'l', 'p' }, { 'm', 'q' }, { 'n', 'r' }, {'o', 's' }, { 'p', 't' }, { 'q', 'v' }, { 'r', 'w' }, { 's', 'x' }, {'t', 'y' }, 'v', 'b' }, { { 'u', 'z' }, { 'w', 'c' }, { 'x', 'd' }, {'y', 'e' }, { 'z', 'h' } };

char str[20];

int main() {
```

```c
    char str[20], str2[20];
    int i;
printf("\n Enter String:");
    gets(str);
    for (i = 0; str[i]; i++) {
        str2[i] = monocipher_encr(str[i]);
    }
    str2[i] = '\0';
printf("\n Before Decryption:%s", str);
printf("\n After Decryption:%s\n", str2);
}
char monocipher_encr(char a) {
    int i;
    for (i = 0; i< 27; i++) {
        if (a == alpha[i][0])
            break;
    }
    return alpha[i][1];
}
```
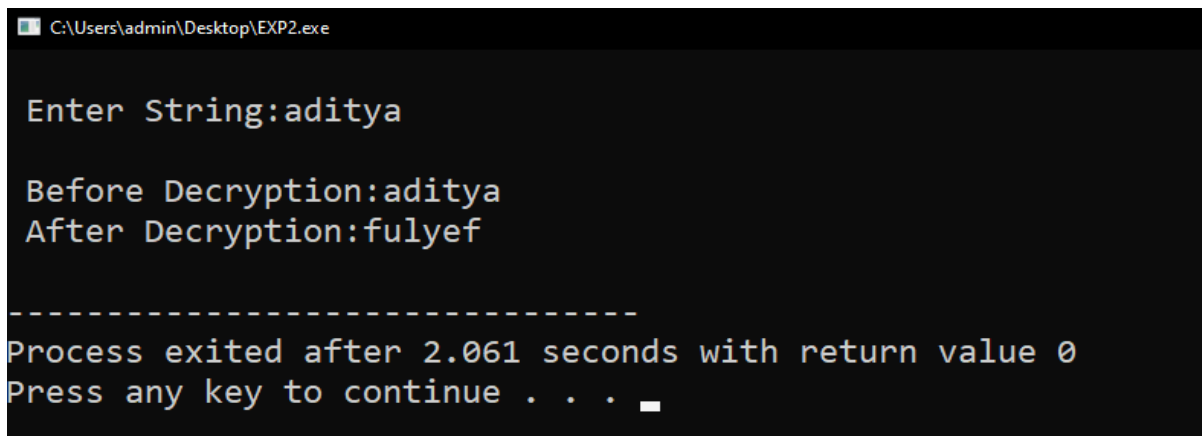
**OUTPUT:**



```
C:\Users\admin\Desktop\EXP2.exe

 Enter String:aditya

 Before Decryption:aditya
 After Decryption:fulyef

---------------------------------
Process exited after 2.061 seconds with return value 0
Press any key to continue . . .
```

1.  **What is cryptography?**

2.  **What exactly are encryption and decryption?**

3.  **What is ciphertext?**

4. **What is the formula for the encryption of shift cipher?**

5. **What is the formula for the decryption shift cipher?**

6. **What is the Mono-alphabetic cipher?**

7. **What is the substitution cipher?**

**Week - 2 Block Ciphers**

**2.1) Write a C Program to implement one-time pad cipher.**

**AIM:** To write a C Program to implement one-time pad cipher.

**DESCRIPTION:**

One-time pad cipher is a type of Vignere cipher which includes the following features −

- It is an unbreakable cipher.
- The key is exactly same as the length of message which is encrypted.
- The key is made up of random symbols.
- As the name suggests, key is used one time only and never used again for any other message to be encrypted.

Due to this, encrypted message will be vulnerable to attack for a cryptanalyst. The key used for a one-time pad cipher is called **pad**, as it is printed on pads of paper.

The two requirements for the One-Time pad are

- The key should be randomly generated as long as the size of the message.
- The key is to be used to encrypt and decrypt a single message, and then it is discarded.

So encrypting every new message requires a new key of the same length as the new message in one-time pad.

The ciphertext generated by the One-Time pad is random, so it does not have any statistical relation with the plain text.

Encryption

To encrypt a letter, a user needs to write a key underneath the plaintext. The plaintext letter is placed on the top and the key letter on the left. The cross section achieved between two letters is the plain text. It is described in the example below :



Decryption

To decrypt a letter, user takes the key letter on the left and finds cipher text letter in that row. The plain text letter is placed at the top of the column where the user can find the cipher text letter.

**PROGRAM:**

```c
#include<stdio.h>
#include<string.h>
#include<ctype.h>
main()
{
//All the text which ever entered is converted to upper and without spaces
 int i,j,len1,len2,numstr[100],numkey[100],numcipher[100];
 char str[100],key[100],cipher[100];
printf("Enter a string text to encrypt\n");
 gets(str);
 for(i=0,j=0;i<strlen(str);i++)
 {
  if(str[i]!=' ')
  {
   str[j]=toupper(str[i]);
   j++;
  }
 }
 str[j]='\0';
//obtaining numerical plain text ex A-0,B-1,C-2
   for(i=0;i<strlen(str);i++)
   {
numstr[i]=str[i]-'A';
   }
printf("Enter key string of random text\n");
   gets(key);
 for(i=0,j=0;i<strlen(key);i++)
```

```c
{
if(key[i]!=' ')
{
key[j]=toupper(key[i]);
j++;
}
}
key[j]='\0';
//obtaining numerical one time pad(OTP) or key
for(i=0;i<strlen(key);i++)
{
numkey[i]=key[i]-'A';
}


for(i=0;i<strlen(str);i++)
{
numcipher[i]=numstr[i]+numkey[i];
}
//To loop the number within 25 i.e if addition of numstr and numkey is 27 then numcipher should be 1
for(i=0;i<strlen(str);i++)
{
if(numcipher[i]>25)
{
numcipher[i]=numcipher[i]-26;
}
}
printf("One Time Pad Cipher text is\n");
for(i=0;i<strlen(str);i++)
```
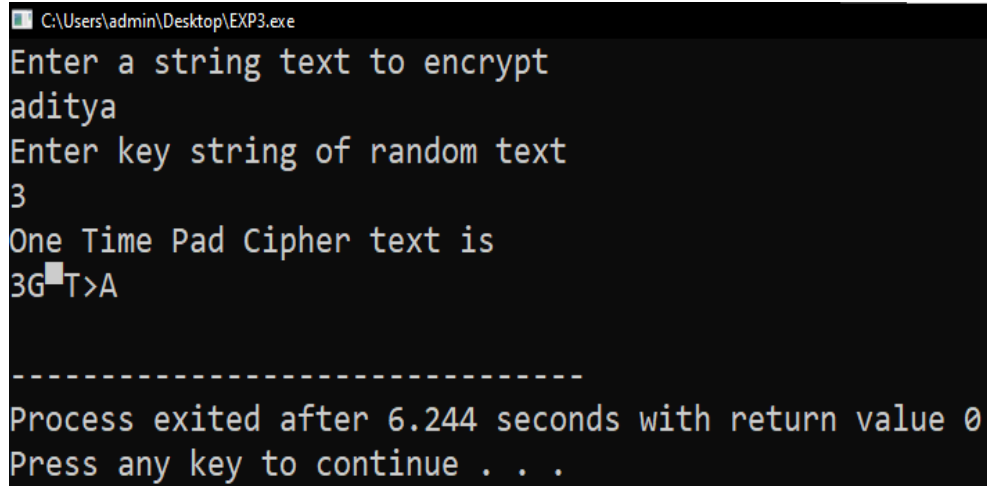
```
    {
printf("%c",(numcipher[i]+'A'));

    }
printf("\n");

}
```

**OUTPUT:**

```
C:\Users\admin\Desktop\EXP3.exe
Enter a string text to encrypt
aditya
Enter key string of random text
3
One Time Pad Cipher text is
3G■T>A

--------------------------------
Process exited after 6.244 seconds with return value 0
Press any key to continue . . .
```

**2.2) Write a C Program to implement vernam cipher.**

**AIM:** To write a C Program to implement vernam cipher.

**DESCRIPTION:**

Vernam Cipher is a method of encrypting alphabetic text. It is one of the Substitution techniques for converting plain text into cipher text. In this mechanism we assign a number to each character of the Plain-Text, like (a = 0, b = 1, c = 2, … z = 25).

Method to take key: In the Vernam cipher algorithm, we take a key to encrypt the plain text whose length should be equal to the length of the plain text.

**Encryption Algorithm:**

- Assign a number to each character of the plain-text and the key according to alphabetical order.
- Bitwise XOR both the number (Corresponding plain-text character number and Key character number).
- Subtract the number from 26 if the resulting number is greater than or equal to 26, if it isn't then leave it.
- $E(P_i, K_i) = P_i \ (XOR) \ K_i$

**EXAMPLE:**

**Plain-Text:** O A K

**Key:** S O N

**O ==>** 14 = 0 1 1 1 0

**S ==>** 18 = 1 0 0 1 0

**Bitwise XOR Result**: 1 1 1 0 0 = 28

Since the resulting number is greater than 26, subtract 26 from it. Then convert the Cipher-Text character number to the Cipher-Text character.

28 - 26 = 2 ==> C

**CIPHER-TEXT:** C

**Decryption Process**

> The process of decrypting the ciphertext to convert it back into plain text is performed in the same way as the encryption process. Therefore, the formula for decryption of the text under Vernam cipher is as follows,

$$D(C_i, K_i) = C_i \ (XOR) \ K_i$$

**PROGRAM:**

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void encrypt(char *plaintext, char *key,char *ciphertext)
{
        int i;
        for(i=0; i<strlen(plaintext);i++)
        {
                ciphertext[i] = plaintext[i] ^ key[i];
        }
}


void decrypt(char *ciphertext , char *key , char *plaintext)
{
        int i;
        for(i=0; i<strlen(ciphertext);i++)
        {
                plaintext[i] = ciphertext[i] ^ key[i];
        }
}



int main(int argc, char *argv[])
{
        char plaintext[100];
        char key[100];
        char ciphertext[100];
```
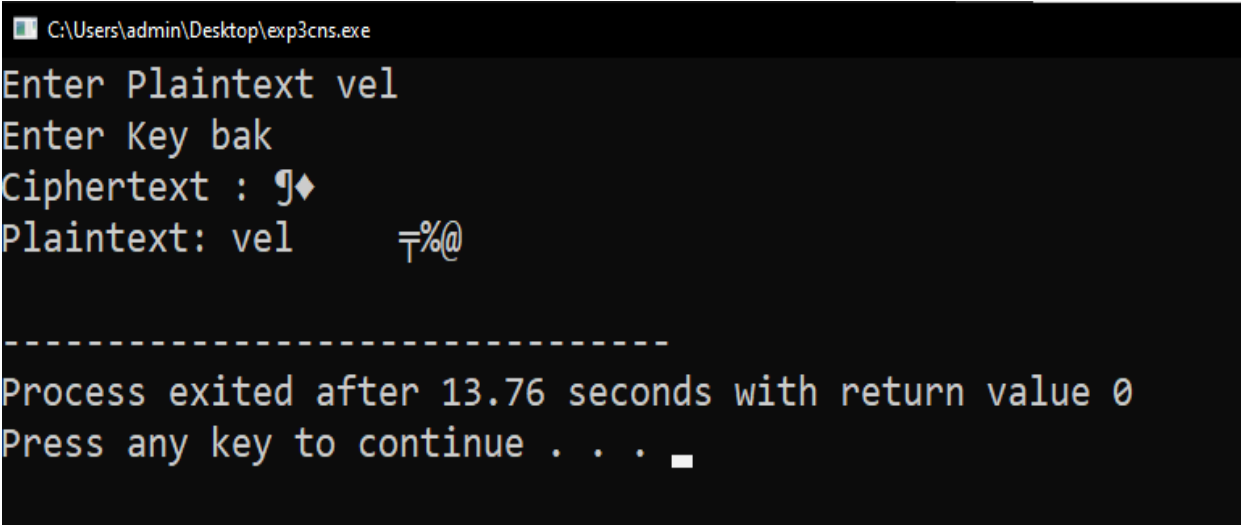
```c
        printf("Enter Plaintext ");
        scanf("%s",plaintext);
        printf("Enter Key ");
        scanf("%s",key);
        encrypt(plaintext,key,ciphertext);
        printf("Ciphertext : %s\n",ciphertext);
        decrypt(ciphertext,key,plaintext);
        printf("Plaintext: %s\n",plaintext);
        return 0;
}
```

**OUTPUT:**

1. **What is the one-time pad cipher?**

2. **What exactly is the Vignere cipher?**

3. **What is ciphertext?**

4. **Why is the one-time pad cipher is unbreakable?**

5. **What is the vernam cipher?**

6. **What is the formula for encrypting the text in the vernam cipher?**

7. **What is the technique used in the vernam cipher and the one-time pad cipher?**

**Week - 3  Symmetric Cryptography**

**3.1) Write a C Program to implement DES Algorithm.**

**AIM :**  To write a C Program to implement DES algorithm.

**DESCRIPTION:**

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only).


Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits.

Since DES is based on the Feistel Cipher, all that is required to specify DES is −

- Round function
- Key schedule
- Any additional processing − Initial and final permutation

**Initial and Final Permutation**

The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES.

**Round Function**

The heart of this cipher is the DES function, *f*. The DES function applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output.

**Expansion Permutation Box**

Since right input is 32-bit and round key is a 48-bit, we first need to expand right input to 48 bits. The graphically depicted permutation logic is generally described as table in DES.

**XOR (Whitener)**

After the expansion permutation, DES does XOR operation on the expanded right section and the round key. The round key is used only in this operation.

**Substitution Boxes**

The S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output.

**PROGRAM:**

```c
#include<stdio.h>
int main()
{
 int i, cnt=0, p8[8]={6,7,8,9,1,2,3,4};
 int p10[10]={6,7,8,9,10,1,2,3,4,5};
 char input[11], k1[10], k2[10], temp[11];
 char LS1[5], LS2[5];
 //k1, k2 are for storing interim keys
 //p8 and p10 are for storing permutation key
 //Read 10 bits from user...
printf("Enter 10 bits input:");
scanf("%s",input);
 input[10]='\0';
 //Applying p10...
 for(i=0; i<10; i++)
 {
cnt = p10[i];
 temp[i] = input[cnt-1];
 }
 temp[i]='\0';
printf("\nYour p10 key is    :");
 for(i=0; i<10; i++)
 { printf("%d,",p10[i]);
 }
printf("\nBits after p10     :");
 puts(temp);
 //Performing LS-1 on first half of temp
```

```c
 for(i=0; i<5; i++)
 {
  if(i==4)
   temp[i]=temp[0];
  else
   temp[i]=temp[i+1];
 }
 //Performing LS-1 on second half of temp
 for(i=5; i<10; i++)
 {
  if(i==9)
   temp[i]=temp[5];
  else
   temp[i]=temp[i+1];
 }
printf("Output after LS-1  :");
 puts(temp);
printf("\nYour p8 key is     :");
 for(i=0; i<8; i++){
printf("%d,",p8[i]);
}
 //Applying p8...
 for(i=0; i<8; i++)
 {
cnt = p8[i];
 k1[i] = temp[cnt-1];
 }
printf("\nYour key k1 is     :");
```
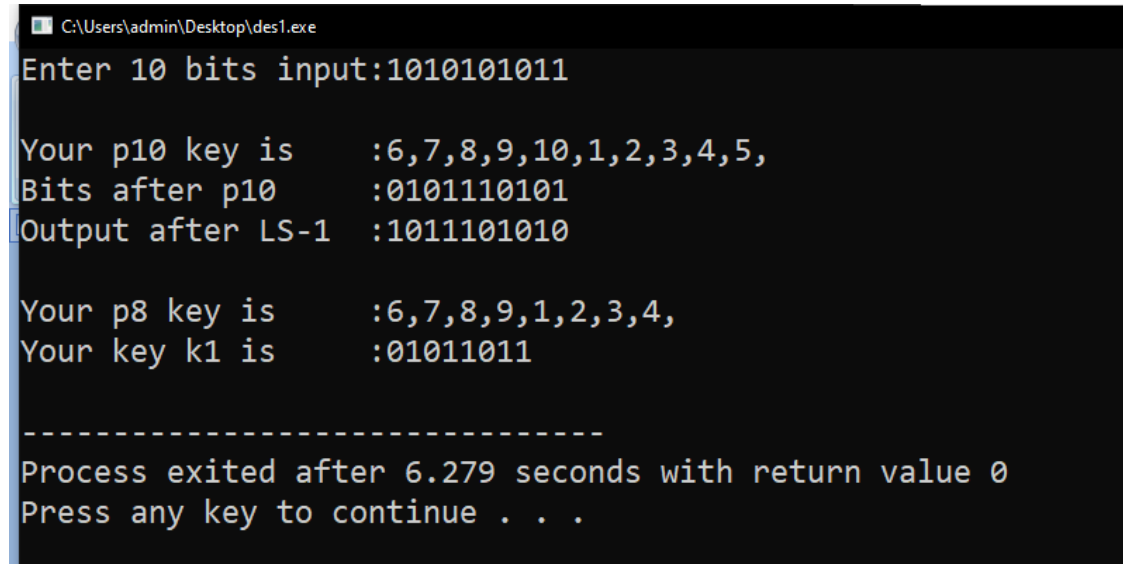
puts(k1);

//This program can be extended to generate k2 as per DES algorithm.

}

**OUTPUT:**

```
C:\Users\admin\Desktop\des1.exe

Enter 10 bits input:1010101011

Your p10 key is     :6,7,8,9,10,1,2,3,4,5,
Bits after p10      :0101110101
Output after LS-1    :1011101010

Your p8 key is      :6,7,8,9,1,2,3,4,
Your key k1 is      :01011011

---------------------------------
Process exited after 6.279 seconds with return value 0
Press any key to continue . . .
```

**3.2) Write a C Program to implement AES algorithm.**

**AIM :** To write a C Program to implement AES algorithm.

**DESCRIPTION:**

The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found at least six time faster than triple DES.

A replacement for DES was needed as its key size was too small. With increasing computing power, it was considered vulnerable against exhaustive key search attack. Triple DES was designed to overcome this drawback but it was found slow.

The features of AES are as follows −

- Symmetric key symmetric block cipher
- 128-bit data, 128/192/256-bit keys
- Stronger and faster than Triple-DES
- Provide full specification and design details
- Software implementable in C and Java

**Encryption Process:** Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes. The first round process is depicted below –

**Byte Substitution (SubBytes)**

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

**Shiftrows**

Each of the four rows of the matrix is shifted to the left. Any entries that 'fall off' are re-inserted on the right side of row.

**MixColumns**

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes.

**Addroundkey**

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

**Decryption Process:** The process of decryption of an AES ciphertext is similar to the encryption process in the reverse order. Each round consists of the four processes conducted in the reverse order −

- Add round key
- Mix columns
- Shift rows
- Byte substitution

**PROGRAM :**

<u>**aes.h:**</u>

#include <stdint.h>

#include <stddef.h>

/* Define constants and sbox */

#define Nb 4

#define Nk(keysize) ((int)(keysize / 32))

#define Nr(keysize) ((int)(Nk(keysize) + 6))

/* State and key types */

typedef uint8_t** State;

typedef uint8_t* Key;

/* My additional methods */

void encrypt(char* plain, char* key);

void decrypt(char* cipher, char* key);

State* toState(uint8_t* input);

uint8_t** fromState(State* state);

void freeState(State* state);

void stringToBytes(char* str, uint8_t* bytes);

/* AES main methods */

uint8_t** Cipher(uint8_t* input, uint8_t* keySchedule, size_tkeySize);

uint8_t** InvCipher(uint8_t* input, uint8_t* w, size_tkeySize);

/* AES sub-methods */

void _SubBytes(State* state, const uint8_t* box);

void SubBytes(State* state);

void InvSubBytes(State* state);

void _ShiftRows(State* state, int multiplier);

void ShiftRows(State* state);

void InvShiftRows(State* state);

void MixColumns(State* state);

void InvMixColumns(State* state);

void AddRoundKey(State* state, uint8_t* roundKey);

void KeyExpansion(uint8_t* key, uint8_t* keySchedule, size_tkeySize);

/* AES sub-sub-methods and round constant array */

uint8_t* SubWord(uint8_t* a);

uint8_t* RotWord(uint8_t* a);

uint8_t* Rcon(int a);

/* AES helper methods */

uint8_t* xorWords(uint8_t* a, uint8_t* b);

uint8_t* copyWord(uint8_t* start);

uint8_t* getWord(uint8_t* w, int i);

uint8_t galoisMultiply(uint8_t a, uint8_t b);

**const.c :**

#include <stdint.h>

const uint8_t sbox[] = {0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};

const uint8_t isbox[] = {0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};

**aes.c :**

```c
#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <stddef.h>

#include "aes.h"

#include "const.c"

int main(){

   /* Examples of encryption */

printf("ENCRYPTION:\n");

   encrypt("3243f6a8885a308d313198a2e0370734","2b7e151628aed2a6abf7158809cf4f3c");

   encrypt("00112233445566778899aabbccddeeff","000102030405060708090a0b0c0d0e0f");
encrypt("00112233445566778899aabbccddeeff","000102030405060708090a0b0c0d0e0f1011121314151617");
encrypt("00112233445566778899aabbccddeeff","000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f");

printf("DECRYPTION:\n");

   decrypt("3925841d02dc09fbdc118597196a0b32", "2b7e151628aed2a6abf7158809cf4f3c");

   decrypt("69c4e0d86a7b0430d8cdb78070b4c55a", "000102030405060708090a0b0c0d0e0f");

   decrypt("dda97ca4864cdfe06eaf70a0ec0d7191", "000102030405060708090a0b0c0d0e0f1011121314151617");

   decrypt("8ea2b7ca516745bfeafc49904b496089", "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f");
```

```c
    return 0;

}

void AES_main(char* text, char* keyStr, int encrypting){

    /* Takes a 128-bit hexadecimal string plaintext and 128-, 192- or 256- bit hexadecimal string key and
applies AES encryption or decryption. */

    uint8_t *keySchedule, **output;

    int i;

    /* Convert input string to state */

    uint8_t* input = malloc(sizeof(uint8_t) * 16);

stringToBytes(text, input);

    /* Convert key string to bytes */

size_tkeyBytes = (sizeof(uint8_t)*strlen(keyStr))/2;

    Key key = malloc(keyBytes);

stringToBytes(keyStr, key);

    /* Convert number of bytes to bits */

size_tkeySize = keyBytes * 8;

    /* Create array for key schedule */

keySchedule = calloc(4 * Nb * (Nr(keySize) + 1), sizeof(uint8_t));

    /* Expand key */

KeyExpansion(key, keySchedule, keySize);

    /* Run cipher */

    if(encrypting){

        output = Cipher(input, keySchedule, keySize);

    } else{

        output = InvCipher(input, keySchedule, keySize);

    }

    /* Display result */

    for(i = 0; i< 16; i++){

printf("%02x", (*output)[i]);
```

```c
    }
printf("\n");
    /* Free memory */
    free(input);

    free(key);

    free(keySchedule);

    free(*output);

    free(output);

}

void encrypt(char* plaintext, char* keyStr){

AES_main(plaintext, keyStr, 1);

}

void decrypt(char* ciphertext, char* keyStr){

AES_main(ciphertext, keyStr, 0);

}
/* AES main methods*/

void KeyExpansion(uint8_t* key, uint8_t* w, size_tkeySize){

    /*Takes a 128-, 192- or 256-bit key and applies the key expansion algorithm to produce a key
schedule.*/

    int i, j;

    uint8_t *wi, *wk, *temp, *rconval;

    /* Copy the key into the first Nk words of the schedule */

    for(i = 0; i<Nk(keySize); i++){

        for(j = 0; j < Nb; j++){

            w[4*i+j] = key[4*i+j];

        }

    }
i = Nk(keySize);

    /* Generate Nb * (Nr + 1) additional words for the schedule */
```

```c
    while(i< Nb * (Nr(keySize) + 1)){

        /* Copy the previous word */

        temp = copyWord(getWord(w, i-1));

        if(i % Nk(keySize) == 0){

            /* If i is divisble by Nk, rotate and substitute the word and then xor with Rcon[i/Nk] */
rconval = Rcon(i/Nk(keySize));
xorWords(SubWord(RotWord(temp)), rconval);
            free(rconval);
        } else if(Nk(keySize) > 6 &&i % Nk(keySize) == 4){
            /* If Nk> 6 and i mod Nk is 4 then just substitute */
memcpy(temp, SubWord(temp), 4);
        }
        /* Get pointers for the current word and the (i-Nk)th word */
wi = getWord(w, i);
wk = getWord(w, i - Nk(keySize));
        /* wi = temp xor wk */
memcpy(wi, xorWords(temp, wk), 4);
        free(temp);
i++;
    }
}
uint8_t** Cipher(uint8_t* input, uint8_t* w, size_tkeySize){
    /*AES Cipher method - Takes a 128 bit array of bytes and the key schedule and applies the cipher algorithm, returning a pointer to an array of output. */
    int i;
    uint8_t** output;
    State* state = toState(input);
    /* Cipher method */
AddRoundKey(state, getWord(w, 0));
```

```c
    for(i = 1; i< Nr(keySize); i++){
SubBytes(state);

ShiftRows(state);

MixColumns(state);

AddRoundKey(state, getWord(w, i*Nb));

    }
SubBytes(state);

ShiftRows(state);

AddRoundKey(state, getWord(w, Nr(keySize)*Nb));

    output = fromState(state);
freeState(state);

    return output;

}
uint8_t** InvCipher(uint8_t* input, uint8_t* w, size_tkeySize){
    /*AES InvCipher method - Takes 128 bits of cipher text and the key schedule and applies the inverse
cipher, returning a pointer to an array of plaintext bytes. */

    int i;

    uint8_t** output;

    State* state = toState(input);

    /* Inverse cipher method */
AddRoundKey(state, getWord(w, Nr(keySize) * Nb));

    for(i = Nr(keySize) - 1; i>= 1; i--){
InvShiftRows(state);

InvSubBytes(state);

AddRoundKey(state, getWord(w, i*Nb));

InvMixColumns(state);

    }
InvShiftRows(state);

InvSubBytes(state);
```

```c
    AddRoundKey(state, getWord(w, 0));

    output = fromState(state);

freeState(state);

    return output;

}

/*State to/from and helper methods*/

State* toState(uint8_t* input){

    /*Takes an array of bytes and returns a pointer to a State.*/

    int i, j;

    /* Malloc state pointer and state. The state pointer is returned because it is more useful than the state itself */

    State* stateptr = malloc(sizeof(State));

    *stateptr = malloc(4 * sizeof(uint8_t*));

    State state = *stateptr;

    for(i = 0; i< 4; i++){

        state[i] = malloc(Nb * sizeof(uint8_t));

    }

    for(i = 0; i< 4; i++){          /* Fill state */

        for(j = 0; j < Nb; j++){

            /* Set value in state array to current byte j and i are swapped because the input is transposed */

            state[j][i] = *input;

            input++;                /* Increment pointer */

        }

    }

    return stateptr;

}

uint8_t** fromState(State* state){

    /*Takes a State and returns a pointer to an array of bytes.*/

    int i, j;
```

```c
    uint8_t** outputptr = malloc(sizeof(uint8_t*));      /* Malloc outputptr and output */

    *outputptr = malloc(sizeof(uint8_t) * 16);

    uint8_t* output = *outputptr;

    for(i = 0; i< 4; i++){          /* Fill output */

       for(j = 0; j < Nb; j++){

          *output = (*state)[j][i];                              /* Increment the pointer */

          output++;            /* Increment the pointer */

       }

    }

    return outputptr;

}
void freeState(State* state){

    /*Free the memory used by each row, the state itself and the pointer to the state. */

    int i;

    for(i = 0; i< 4; i++){

       free((*state)[i]);

    }

    free(*state);

    free(state);

}
void stringToBytes(char* str, uint8_t* bytes){

    /*Converts a hexadecimal string of bytes into an array of uint8_t.*/

    int i;

    for(i = 0; i<strlen(str) - 1; i += 2){

       char* pair = malloc(2 * sizeof(char));       /* Allocate space for pair of nibbles */

memcpy(pair, &str[i], 2);                /* Copy current and next character to pair */

 /* Use strtol to convert string to long, which is implicitly converted to a uint8_t. This is stored in index i/2
as there are half as many bytes as  hex characters */

       bytes[i/2] = strtol(pair, NULL, 16);
```

```c
        free(pair);

    }

}

/*AES sub-methods*/

void _SubBytes(State* state, const uint8_t* box){

    /*GeneralisedSubBytes method which takes the S-box to use as an argument.*/

    int i, j;

    for(i = 0; i< 4; i++){

        for(j = 0; j < Nb; j++){

            uint8_t new = box[(*state)[i][j]];    /* Get the new value from the S-box */


            (*state)[i][j] = new;

        }

    }

}

void SubBytes(State* state){

    _SubBytes(state, sbox);

}

void InvSubBytes(State* state){

    _SubBytes(state, isbox);

}

void _ShiftRows(State* state, int multiplier){

    /*GeneralisedShiftRows method which takes a multiplier which affects the shift direction.*/

    int i, j;

    for(i = 0; i< 4; i++){

        /* The row number is the number of shifts to do */

        uint8_t temp[4];

        for(j = 0; j < Nb; j++){
```

```c
            /* The multiplier determines whether to do a left or right shift */

            temp[((j + Nb) + (multiplier * i)) % Nb] = (*state)[i][j];

        }
        /* Copy temp array to state array */
memcpy((*state)[i], temp, 4);

    }
}
void ShiftRows(State* state){
    _ShiftRows(state, -1);
}
void InvShiftRows(State* state){
    _ShiftRows(state, 1);
}
uint8_t galoisMultiply(uint8_t a, uint8_t b){
    uint8_t p = 0;
    int i;
    int carry;
    for(i = 0; i< 8; i++){
        if((b & 1) == 1){
            p ^= a;
        }
        b >>= 1;
        carry = a & 0x80;
        a <<= 1;
        if(carry == 0x80){
            a ^= 0x1b;
        }
    }
```

```c
    return p;

}

void MixColumns(State* state){

    /*Applies the MixColumns method to the state.*/

    int c, r;

    for(c = 0; c < Nb; c++){

        uint8_t temp[4];

        temp[0] = galoisMultiply((*state)[0][c], 2) ^ galoisMultiply((*state)[1][c], 3) ^ (*state)[2][c] ^ (*state)[3][c];

        temp[1] = (*state)[0][c] ^ galoisMultiply((*state)[1][c], 2) ^ galoisMultiply((*state)[2][c], 3) ^ (*state)[3][c];

        temp[2] = (*state)[0][c] ^ (*state)[1][c] ^ galoisMultiply((*state)[2][c], 2) ^ galoisMultiply((*state)[3][c], 3);

        temp[3] = galoisMultiply((*state)[0][c], 3) ^ (*state)[1][c] ^ (*state)[2][c] ^ galoisMultiply((*state)[3][c], 2);

        /* Copy temp array to state */

        for(r = 0; r < 4; r++){

            (*state)[r][c] = temp[r];

        }

    }

}

void InvMixColumns(State* state){

    /*Applies InvMixColumns to the state. See Section 5.3.3 of the standard for explanation. */

    int c, r;

    for(c = 0; c < Nb; c++){

        uint8_t temp[4];

        temp[0] = galoisMultiply((*state)[0][c], 14) ^ galoisMultiply((*state)[1][c], 11) ^ galoisMultiply((*state)[2][c], 13) ^ galoisMultiply((*state)[3][c], 9);

        temp[1] = galoisMultiply((*state)[0][c], 9) ^ galoisMultiply((*state)[1][c], 14) ^ galoisMultiply((*state)[2][c], 11) ^ galoisMultiply((*state)[3][c], 13);
```

```c
        temp[2]    =    galoisMultiply((*state)[0][c],    13)    ^    galoisMultiply((*state)[1][c],    9)        ^
galoisMultiply((*state)[2][c], 14) ^ galoisMultiply((*state)[3][c], 11);

        temp[3]    =    galoisMultiply((*state)[0][c],    11)    ^    galoisMultiply((*state)[1][c],    13)    ^
galoisMultiply((*state)[2][c], 9) ^ galoisMultiply((*state)[3][c], 14);

    /* Copy temp array to state */

    for(r = 0; r < 4; r++){

        (*state)[r][c] = temp[r];

    }

  }

}

void AddRoundKey(State* state, uint8_t* roundKey){

  /*Takes a pointer to the start of a round key and XORs it with the columns of the state. */

  int c, r;

  for(c = 0; c < Nb; c++){

    for(r = 0; r < 4; r++){

        /* XOR each column with the round key */

        (*state)[r][c] ^= *roundKey;

roundKey++;

    }

  }

}

/* AES sub-sub-methods*/

uint8_t* SubWord(uint8_t* a){

  /*Substitute bytes in a word using the sbox.*/

  int i;

  uint8_t* init = a;

  for(i = 0; i< 4; i++){

    *a = sbox[*a];

    a++;
```

```c
    }
    return init;
}
uint8_t* RotWord(uint8_t* a){
    /*Rotate word then copy to pointer.*/
    uint8_t rot[] = {a[1], a[2], a[3], a[0]};
memcpy(a, rot, 4);
    return a;
}
uint8_t* Rcon(int a){
    /* Calculates the round constant and returns it in an array.*/
    uint8_t rcon = 0x8d;
    int i;
    for(i = 0; i< a; i++){
rcon = ((rcon<< 1) ^ (0x11b & - (rcon>> 7)));
    }
    /* The round constant array is always of the form [rcon, 0, 0, 0] */
    uint8_t* word = calloc(4, sizeof(uint8_t));
    word[0] = rcon;
    return word;
}
/*Word helper methods*/
uint8_t* xorWords(uint8_t* a, uint8_t* b){
    /* Takes the two pointers to the start of 4 byte words and
        XORs the words, overwriting the first. Returns a pointer to
        the first byte of the first word. */
    int i;
    uint8_t* init = a;
```

```c
    for(i = 0; i< 4; i++, a++, b++){

        *a ^= *b;

    }

    return init;

}

uint8_t* copyWord(uint8_t* start){

    /*Returns a pointer to a copy of a word.*/

    int i;

    uint8_t* word = malloc(sizeof(uint8_t) * 4);

    for(i = 0; i< 4; i++, start++){

        word[i] = *start;

    }

    return word;

}

uint8_t* getWord(uint8_t* w, int i){

    /*Takes a word number (w[i] in spec) and returns a pointer to the first of it's 4 bytes.*/

    return &w[4*i];

}
```

**OUTPUT:**



```
C:\Users\admin\Desktop\aes.exe
ENCRYPTION:
cfddbd3a3c7767d185714447874d4662
d44e6cfe2851c98ae4cf65c861a2b0bc
4d126de22cbf53cab90445bf1cdc3892
b87b852a19c97325cb46eb68a4644a22
DECRYPTION:
c0266ec96b52ae9a5940c4812736f88a
d81bb976d7403f40541ff8cf1f545d3c
8c4b06c926e1b3ab141e9ad527526511
7cf23077b4b94572d8089a54882267e0

--------------------------------
Process exited after 0.2378 seconds with return value 0
Press any key to continue . . .
```

4.  **What is the DES algorithm?**

2.  **What exactly is the AES algorithm?**

3.  **What are symmetric and asymmetric key systems?**

4. **What are the types of fields/levels in the encryption process of AES algorithm?**

5. **How many rounds and bits used in the DES algorithm?**

6. **What is the difference between DES and Triple DES?**

7. **What is the type of cipher is used in the AES algorithm for encrypt and decrypt message?**

**Week - 4 Asymmetric Cryptography**

**4.1) Write a C Program to implement RSA algorithm.**

**AIM:** To write a C Program to implement RSA algorithm.

**DESCRIPTION:**

RSA encryption algorithm is a type of public-key encryption algorithm. Asymmetric actually means that it works on two different keys i.e. **Public Key** and **Private Key.** As the name describes that the Public Key is given to everyone and the Private key is kept private.

**Public key encryption algorithm:**

RSA is the common public-key algorithm, named after its inventors **Rivest, Shamir, and Adelman (RSA).**Public Key encryption algorithm is also called the Asymmetric algorithm. Asymmetric algorithms are those algorithms in which sender and receiver use different keys for encryption and decryption. Each sender is assigned a pair of keys:

      **1.Public key**                                   **2.Private key**

The **Public key** is used for encryption, and the **Private Key** is used for decryption. Decryption cannot be done using a public key. The two keys are linked, but the private key cannot be derived from the public key. The public key is well known, but the private key is secret and it is known only to the user who owns the key. It means that everybody can send a message to the user using user's public key. But only the user can decrypt the message using his private key.

The RSA algorithm holds the following features −

- RSA algorithm is a popular exponentiation in a finite field over integers including prime numbers.
- The integers used by this method are sufficiently large making it difficult to solve.
- There are two sets of keys in this algorithm: private key and public key.

Encryption Formula

Consider a sender who sends the plain text message to someone whose public key is **(n,e).** To encrypt the plain text message in the given scenario, use the following syntax −

$$\text{Ciphertext} = P^e \bmod n$$

Decryption Formula

The decryption process is very straightforward and includes analytics for calculation in a systematic approach. Considering receiver **C** has the private key **d**, the result modulus will be calculated as –

$$\text{Plaintext} = C^d \bmod n$$

**PROGRAM:**

```c
#include<stdio.h>
#include<math.h>

//to find gcd
int gcd(int a, int h)
{
    int temp;
    while(1)
    {
        temp = a%h;
        if(temp==0)
        return h;
        a = h;
        h = temp;
    }
}

int main()
{
    //2 random prime numbers
    double p = 3;
    double q = 7;
    double n=p*q;
    double count;

    double totient = (p-1)*(q-1);

    //public key
    //e stands for encrypt
    double e=2;

    //for checking co-prime which satisfies e>1
    while(e<totient){
    count = gcd(e,totient);
    if(count==1)
        break;
    else
        e++;
    }

    //private key
    //d stands for decrypt
    double d;

    //k can be any arbitrary value
    double k = 2;
```
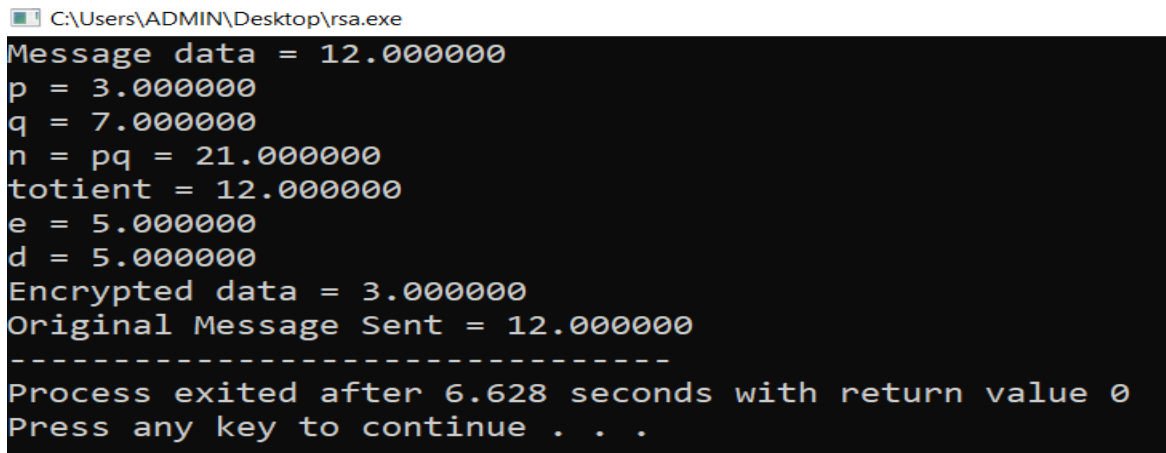
```c
    //choosing d such that it satisfies d*e = 1 + k * totient
    d = (1 + (k*totient))/e;
    double msg = 12;
    double c = pow(msg,e);
    double m = pow(c,d);
    c=fmod(c,n);
    m=fmod(m,n);

printf("Message data = %lf",msg);
printf("\np = %lf",p);
printf("\nq = %lf",q);
printf("\nn = pq = %lf",n);
printf("\ntotient = %lf",totient);
printf("\ne = %lf",e);
printf("\nd = %lf",d);
printf("\nEncrypted data = %lf",c);
printf("\nOriginal Message Sent = %lf",m);

    return 0;
}
```

**OUTPUT:**



```
C:\Users\ADMIN\Desktop\rsa.exe
Message data = 12.000000
p = 3.000000
q = 7.000000
n = pq = 21.000000
totient = 12.000000
e = 5.000000
d = 5.000000
Encrypted data = 3.000000
Original Message Sent = 12.000000
------------------------------
Process exited after 6.628 seconds with return value 0
Press any key to continue . . .
```

**4.2) Write a C Program to implement Diffie-Helman Key Exchange Algorithm.**

**AIM:** To write a C Program to implement Diffie-Helman Key Exchange Algorithm.

**DESCRIPTION:**

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

- For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.

- P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

| Alice | Bob |
|---|---|
| Public Keys available = P, G | Public Keys available = P, G |
| Private Key Selected = a | Private Key Selected = b |
| Key generated = $x = G^a mod P$ | Key generated = $y = G^b mod P$ |
| Exchange of generated keys takes place | |
| Key received = y | key received = x |
| Generated Secret Key = $k_a = y^a mod P$ | Generated Secret Key = $k_b = x^b mod P$ |
| Algebraically, it can be shown that $k_a = k_b$ | |
| Users now have a symmetric secret key to encrypt | |

**Vulnerabilities of Diffie-Hellman key exchange**

The most serious limitation of Diffie-Hellman in its basic form is the lack of <u>authentication</u>. Communications using Diffie-Hellman by itself are vulnerable to MitM. Ideally, Diffie-Hellman should be used in conjunction with a recognized authentication method, such as <u>digital signatures,</u> to verify the identities of the users over the public communications medium.

**PROGRAM:**

```c
#include<stdio.h>
#include<math.h>

// Power function to return value of a ^ b mod P
long long int power(long long int a, long long int b, long long int P)
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

//Driver program
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Both the persons will be agreed upon the
        // public keys G and P
    P = 23;

// A prime number P is taken
    printf("The value of P : %lld\n", P);

    G = 9;

// A primitive root for P, G is taken
    printf("The value of G : %lld\n\n", G);

    // Alice will choose the private key a
    a = 4;

// a is the chosen private key
    printf("The private key a for Alice : %lld\n", a);
    x = power(G, a, P);

// gets the generated key

    // Bob will choose the private key b
    b = 3;

// b is the chosen private key
    printf("The private key b for Bob : %lld\n\n", b);
    y = power(G, b, P);
```

// gets the generated key
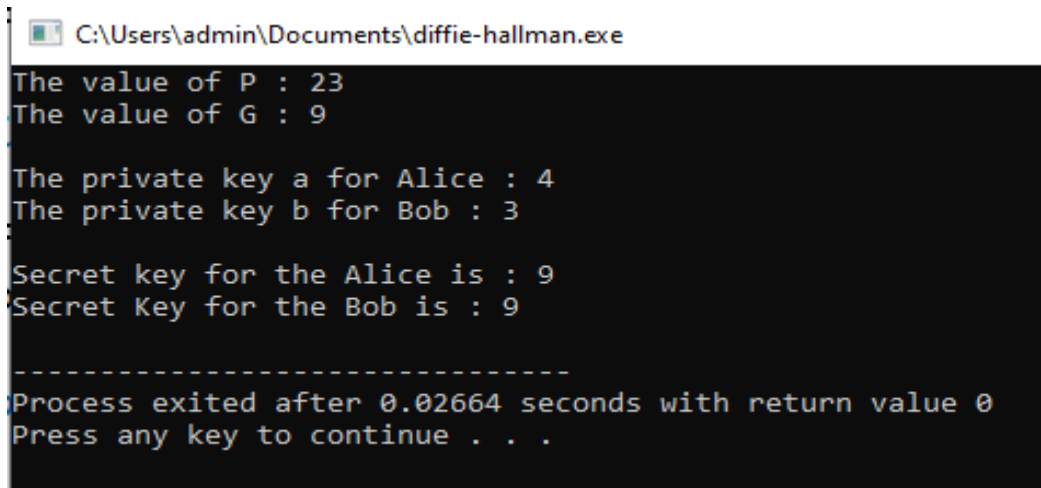
    // Generating the secret key after the exchange
      // of keys
   ka = power(y, a, P); // Secret key for Alice
   kb = power(x, b, P); // Secret key for Bob

   printf("Secret key for the Alice is : %lld\n", ka);
   printf("Secret Key for the Bob is : %lld\n", kb);

   return 0;
}

**OUTPUT:**

**4.3) Write a C Program to implement Elgamal Cryptographic System.**

**AIM:** To write a C Program to implement Elgamal Cryptographic System.

**DESCRIPTION:**

**ElGamal encryption** is a public-key cryptosystem. It uses asymmetric key encryption for communicating between two parties and encrypting the message.
This cryptosystem is based on the difficulty of finding **discrete logarithm** in a cyclic group that is even if we know $g^a$ and $g^k$, it is extremely difficult to compute $g^{ak}$.

It can be considered the asymmetric algorithm where the encryption and decryption happen by using public and private keys. In order to encrypt the message, the public key is used by the client, while the message could be decrypted using the private key on the server end.

This is considered an efficient algorithm to perform encryption and decryption as the keys are extremely tough to predict. The sole purpose of introducing the message transaction's signature is to protect it against MITM, which this algorithm could very effectively achieve.

**Idea of ElGamal cryptosystem**:

- Suppose Alice wants to communicate with Bob.

    1. Bob generates public and private keys:
        - Bob chooses a very large number **q** and a cyclic group $\mathbf{F_q}$.
        - From the cyclic group $\mathbf{F_q}$, he choose any element **g** and an element **a** such that $\gcd(a, q) = 1$.
        - Then he computes $h = g^a$.
        - Bob publishes **F**, $\mathbf{h = g^a}$, **q**, and **g** as his public key and retains **a** as private key.

    2. Alice encrypts data using Bob's public key :
        - Alice selects an element **k** from cyclic group **F** such that $\gcd(k, q) = 1$.
        - Then she computes $p = g^k$ and $s = h^k = g^{ak.}$
        - She multiples s with M.
        - Then she sends $(p, M*s) = (g^k, M*s)$.

    3. Bob decrypts the message :
        - Bob calculates $s' = p^a = g^{ak}$.
        - He divides $M*s$ by $s'$ to obtain M as $s = s'$.

It is mainly concerned about the difficulty of leveraging the cyclic group to find the discrete logarithm.

**PROGRAM:**

```c
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

int e1, e2;

int p, d;

int C1, C2;

FILE *out1, *out2;

int gcd(int a, int b)

{

    int q, r1, r2, r;

    if (a > b)

    {

        r1 = a;

        r2 = b;

    }

    else {

        r1 = b;

        r2 = a;

    }
```

```c
    while (r2 > 0)

    {

        q = r1 / r2;

        r = r1 - q * r2;

        r1 = r2;

        r2 = r;

    }

    return r1;

}

int FastExponention(int bit, int n, int* y, int* a)

{

    if (bit == 1) {

        *y = (*y * (*a)) % n;

    }

    *a = (*a) * (*a) % n;

}

int FindT(int a, int m, int n)

{

    int r;

    int y = 1;

    while (m > 0)
```

```c
    {

        r = m % 2;

        FastExponention(r, n, &y, &a);

        m = m / 2;

    }


    return y;

}

int PrimarityTest(int a, int i)

{

    int n = i - 1;

    int k = 0;

    int m, T;

    while (n % 2 == 0)

    {

        k++;

        n = n / 2;

    }

    m = n;

    T = FindT(a, m, i);

    if (T == 1 || T == i - 1) {
```

```c
        return 1;

    }

        int j;

    for (j = 0; j < k; j++)

    {

        T = FindT(T, 2, i);

        if (T == 1) {

            return 0;

        }

        if (T == i - 1) {

            return 1;

        }

    }

    return 0;

}

int PrimitiveRoot(int p)

{

    int flag;

    int a;

    for (a = 2; a < p; a++)

    {
```

```
      flag = 1;

      int i;

      for (i = 1; i < p; i++)

      {

         if (FindT(a, i, p) == 1 && i < p - 1) {

            flag = 0;

         }

         else if (flag && FindT(a, i, p) == 1 && i == p - 1) {

            return a;

         }

      }

   }

}

int KeyGeneration()

{

   do {

      do

         p = rand() + 256;

      while (p % 2 == 0);

   } while (!PrimarityTest(2, p));

   p = 107;
```

```c
    e1 = 2;

    do {

        d = rand() % (p - 2) + 1;      // 1 <= d <= p-2

    } while (gcd(d, p) != 1);

    d = 67;

    e2 = FindT(e1, d, p);

}

int Encryption(int Plaintext)

{

    out1 = fopen("cipher1.txt", "a+");

    out2 = fopen("cipher2.txt", "a+");

    int r;

    do {

        r = rand() % (p - 1) + 1;      // 1 < r < p

    }

    while (gcd(r, p) != 1);

    C1 = FindT(e1, r, p);

    C2 = FindT(e2, r, p) * Plaintext % p;

    fprintf(out1, "%d ", C1);

    fprintf(out2, "%d ", C2);

    fclose(out1);
```

```c
        fclose(out2);

}

int Decryption(int C1, int C2)

{

    FILE* out = fopen("result.txt", "a+");

    int decipher = C2 * FindT(C1, p - 1 - d, p) % p;

    fprintf(out, "%c", decipher);

    fclose(out);

}

int main()

{

    FILE *out, *inp;

    // destroy contents of these files (from previous runs, if any)

    out = fopen("result.txt", "w+");

    fclose(out);

    out = fopen("cipher1.txt", "w+");

    fclose(out);

    out = fopen("cipher2.txt", "w+");

    fclose(out);

    KeyGeneration();

    inp = fopen("plain.txt", "r+");
```

```c
    if (inp == NULL)

    {

        printf("Error opening Source File.\n");

        exit(1);

    }

    while (1)

    {

        char ch = getc(inp);

        if (ch == EOF) {

            break;          // M < p

        }

        Encryption(toascii(ch));

    }

    fclose(inp);

    FILE *inp1, *inp2;

    inp1 = fopen("cipher1.txt", "r");

    inp2 = fopen("cipher2.txt", "r");

    int C1, C2;

    while (1)

    {

        int ret = fscanf(inp1, "%d", &C1);
```

```c
        fscanf(inp2, "%d", &C2);

        if (ret == -1) {

            break;

        }

        Decryption(C1, C2);

    }

    fclose(inp1);

    fclose(inp2);

    return 0;

}
```

**OUTPUT:**

```
Enter a prime number: 223
Enter the private key: 23
Enter the generator: 19
Enter the plain text: Elgamal
Enter the sender key: 31

Plain text: Elgamal

Encrypted Message: �UO�ﭬU

Decrypted Message: Elgamal
```

1. **What is the RSA algorithm?**

2. **What do you know about Diffie-Hellman key exchange algorithm?**

3. **What is Elgamal cryptographic algorithm?**

4. **What are the types of fields/levels in the encryption process of AES algorithm?**

5. **How many rounds and bits used in the DES algorithm?**

6. **What is the difference between DES and Triple DES?**

7. **What is the type of cipher is used in the AES algorithm for encrypt and decrypt message?**

**Week - 5 Message Authentication Codes**

**5.1) Write a C Program to implement HMAC**

**AIM:** To write a C Program to implement HMAC.

**DESCRIPTION:**

**HMAC** (Hash-based Message Authentication Code) is a type of a message authentication code (MAC) that is acquired by executing a cryptographic hash function on the data (that is) to be authenticated and a secret shared key. Like any of the MAC, it is used for both data integrity and authentication. Checking data integrity is necessary for the parties involved in communication. HTTPS, SFTP, FTPS, and other transfer protocols use HMAC.

The cryptographic hash function may be MD-5, SHA-1, or SHA-256. Digital signatures are nearly similar to HMACs i.e they both employ a hash function and a shared key. The difference lies in the keys i.e HMACs use symmetric key(same copy) while Signatures use asymmetric (two different keys).

**WORKING Of HMAC:**

HMACs provides client and server with a shared private key that is known only to them. The client makes a unique hash (HMAC) for every request. When the client requests the server, it hashes the requested data with a private key and sends it as a part of the request. Both the message and key are hashed in separate steps making it secure. When the server receives the request, it makes its own HMAC. Both the HMACS are compared and if both are equal, the client is considered legitimate.

The formula **for HMAC:**

$$HMAC = hashFunc(secret\ key + message)$$

There are three types of authentication functions.

1.encryption        2.message authentication code            3.hash functions.

The major difference between MAC and hash (HMAC here) is the dependence of a key. In HMAC we have to apply the hash function along with a key on the plain text. The hash function will be applied to the plain text message.

But before applying, we have to compute S bits and then append it to plain text and after that apply the hash function. For generating those S bits we make use of a key that is shared between the sender and receiver.

**PROGRAM:**

```c
#define TRACE_LEVEL CRYPTO_TRACE_LEVEL

//Dependencies
#include "core/crypto.h"
#include "mac/hmac.h"

//Check crypto library configuration
#if (HMAC_SUPPORT == ENABLED)

//HMAC with MD5 OID (1.3.6.1.5.5.8.1.1)
const uint8_t HMAC_WITH_MD5_OID[8] = {0x2B, 0x06, 0x01, 0x05, 0x05, 0x08, 0x01, 0x01};
//HMAC with Tiger OID (1.3.6.1.5.5.8.1.3)
const uint8_t HMAC_WITH_TIGER_OID[8] = {0x2B, 0x06, 0x01, 0x05, 0x05, 0x08, 0x01, 0x03};
//HMAC with RIPEMD-160 OID (1.3.6.1.5.5.8.1.4)
const uint8_t HMAC_WITH_RIPEMD160_OID[8] = {0x2B, 0x06, 0x01, 0x05, 0x05, 0x08, 0x01, 0x04};
//HMAC with SHA-1 OID (1.2.840.113549.2.7)
const uint8_t HMAC_WITH_SHA1_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x07};
//HMAC with SHA-224 OID (1.2.840.113549.2.8)
const uint8_t HMAC_WITH_SHA224_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x08};
//HMAC with SHA-256 OID (1.2.840.113549.2.9)
const uint8_t HMAC_WITH_SHA256_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x09};
//HMAC with SHA-384 OID (1.2.840.113549.2.10)
const uint8_t HMAC_WITH_SHA384_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x0A};
//HMAC with SHA-512 OID (1.2.840.113549.2.11)
const uint8_t HMAC_WITH_SHA512_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x0B};
//HMAC with SHA-512/224 OID (1.2.840.113549.2.12)
const uint8_t HMAC_WITH_SHA512_224_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x0C};
//HMAC with SHA-512/256 OID (1.2.840.113549.2.13)
const uint8_t HMAC_WITH_SHA512_256_OID[8] = {0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x0D};
//HMAC with SHA-3-224 object identifier (2.16.840.1.101.3.4.2.13)
const uint8_t HMAC_WITH_SHA3_224_OID[9] = {0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0D};
//HMAC with SHA-3-256 object identifier (2.16.840.1.101.3.4.2.14)
const uint8_t HMAC_WITH_SHA3_256_OID[9] = {0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0E};
//HMAC with SHA-3-384 object identifier (2.16.840.1.101.3.4.2.15)
const uint8_t HMAC_WITH_SHA3_384_OID[9] = {0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0F};
//HMAC with SHA-3-512 object identifier (2.16.840.1.101.3.4.2.16)
const uint8_t HMAC_WITH_SHA3_512_OID[9] = {0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04,
```

0x02, 0x10};


```c
/**
 * @brief Compute HMAC using the specified hash function
 * @param[in] hash Hash algorithm used to compute HMAC
 * @param[in] key Key to use in the hash algorithm
 * @param[in] keyLen Length of the key
 * @param[in] data The input data for which to compute the hash code
 * @param[in] dataLen Length of the input data
 * @param[out] digest The computed HMAC value
 * @return Error code
 **/

__weak_func error_t hmacCompute(const HashAlgo *hash, const void *key, size_t keyLen,
   const void *data, size_t dataLen, uint8_t *digest)
{
   error_t error;
   HmacContext *context;

   //Allocate a memory buffer to hold the HMAC context
   context = cryptoAllocMem(sizeof(HmacContext));

   //Successful memory allocation?
   if(context != NULL)
   {
      //Initialize the HMAC context
      error = hmacInit(context, hash, key, keyLen);

      //Check status code
      if(!error)
      {
         //Digest the message
         hmacUpdate(context, data, dataLen);
         //Finalize the HMAC computation
         hmacFinal(context, digest);
      }

      //Free previously allocated memory
      cryptoFreeMem(context);
   }
   else
   {
      //Failed to allocate memory
      error = ERROR_OUT_OF_MEMORY;
   }

   //Return status code
   return error;
```

```
}


/**
 * @brief Initialize HMAC calculation
 * @param[in] context Pointer to the HMAC context to initialize
 * @param[in] hash Hash algorithm used to compute HMAC
 * @param[in] key Key to use in the hash algorithm
 * @param[in] keyLen Length of the key
 * @return Error code
 **/

__weak_func error_t hmacInit(HmacContext *context, const HashAlgo *hash,
  const void *key, size_t keyLen)
{
  uint_t i;

  //Check parameters
  if(context == NULL || hash == NULL)
    return ERROR_INVALID_PARAMETER;

  //Make sure the supplied key is valid
  if(key == NULL && keyLen != 0)
    return ERROR_INVALID_PARAMETER;

  //Hash algorithm used to compute HMAC
  context->hash = hash;

  //The key is longer than the block size?
  if(keyLen > hash->blockSize)
  {
    //Initialize the hash function context
    hash->init(&context->hashContext);
    //Digest the original key
    hash->update(&context->hashContext, key, keyLen);
    //Finalize the message digest computation
    hash->final(&context->hashContext, context->key);

    //Key is padded to the right with extra zeros
    osMemset(context->key + hash->digestSize, 0,
      hash->blockSize - hash->digestSize);
  }
  else
  {
    //Copy the key
    osMemcpy(context->key, key, keyLen);
    //Key is padded to the right with extra zeros
    osMemset(context->key + keyLen, 0, hash->blockSize - keyLen);
  }
```

```c
   //XOR the resulting key with ipad
   for(i = 0; i < hash->blockSize; i++)
   {
      context->key[i] ^= HMAC_IPAD;
   }

   //Initialize context for the first pass
   hash->init(&context->hashContext);
   //Start with the inner pad
   hash->update(&context->hashContext, context->key, hash->blockSize);

   //Successful initialization
   return NO_ERROR;
}


/**
 * @brief Update the HMAC context with a portion of the message being hashed
 * @param[in] context Pointer to the HMAC context
 * @param[in] data Pointer to the buffer being hashed
 * @param[in] length Length of the buffer
 **/

__weak_func void hmacUpdate(HmacContext *context, const void *data, size_t length)
{
   const HashAlgo *hash;

   //Hash algorithm used to compute HMAC
   hash = context->hash;
   //Digest the message (first pass)
   hash->update(&context->hashContext, data, length);
}


/**
 * @brief Finish the HMAC calculation
 * @param[in] context Pointer to the HMAC context
 * @param[out] digest Calculated HMAC value (optional parameter)
 **/

__weak_func void hmacFinal(HmacContext *context, uint8_t *digest)
{
   uint_t i;
   const HashAlgo *hash;

   //Hash algorithm used to compute HMAC
   hash = context->hash;
   //Finish the first pass
```

```c
   hash->final(&context->hashContext, context->digest);

   //XOR the original key with opad
   for(i = 0; i < hash->blockSize; i++)
   {
      context->key[i] ^= HMAC_IPAD ^ HMAC_OPAD;
   }

   //Initialize context for the second pass
   hash->init(&context->hashContext);
   //Start with outer pad
   hash->update(&context->hashContext, context->key, hash->blockSize);
   //Then digest the result of the first hash
   hash->update(&context->hashContext, context->digest, hash->digestSize);
   //Finish the second pass
   hash->final(&context->hashContext, context->digest);

   //Copy the resulting HMAC value
   if(digest != NULL)
   {
      osMemcpy(digest, context->digest, hash->digestSize);
   }
}


/**
 * @brief Finish the HMAC calculation (no padding added)
 * @param[in] context Pointer to the HMAC context
 * @param[out] digest Calculated HMAC value (optional parameter)
 **/

void hmacFinalRaw(HmacContext *context, uint8_t *digest)
{
   uint_t i;
   const HashAlgo *hash;

   //Hash algorithm used to compute HMAC
   hash = context->hash;

   //XOR the original key with opad
   for(i = 0; i < hash->blockSize; i++)
   {
      context->key[i] ^= HMAC_IPAD ^ HMAC_OPAD;
   }

   //Initialize context for the second pass
   hash->init(&context->hashContext);
   //Start with outer pad
   hash->update(&context->hashContext, context->key, hash->blockSize);
```

```
        //Then digest the result of the first hash
        hash->update(&context->hashContext, context->digest, hash->digestSize);
        //Finish the second pass
        hash->final(&context->hashContext, context->digest);

        //Copy the resulting HMAC value
        if(digest != NULL)
        {
            osMemcpy(digest, context->digest, hash->digestSize);
        }
    }
```

**OUTPUT:**

```
hmac-sha512 - Hex Message Digest One:
8276c1121aea0fc07cfe058164cb0b0b1b4791f9624d603bb1b303256be33324fc0012eb3c28ac4d3f8641488fbf7480ba8ce3c7652514d612507aae2d95f388
hmac-sha512 - Hex Message Digest Two :
8276c1121aea0fc07cfe058164cb0b0b1b4791f9624d603bb1b303256be33324fc0012eb3c28ac4d3f8641488fbf7480ba8ce3c7652514d612507aae2d95f388
hmac-sha512 - Hex Message Digest Three :
7a5c5aa6f7040c78f8be34c28259a7cc484397d6bccb9182ea5e9d1f6d4520e24a62063c60330bb462460462fb2dd46376839a0b620e868222203b3ce8d0c19e

Message Digest Size for 1 : 64, 2 : 64 and 3 : 64
Message Block  Size for 1 : 128, 2 : 128 and 3 : 128
```

**5.2) Write a C Program to implement CMAC**

**AIM:** To write a C Program to implement CMAC.

**DESCRIPTION:**

Cipher-Based Message Authentication Code (CMAC) is a MAC that is based on the use of a block cipher mode of operations for use with AES and tripleDES. It is also adopted by NIST. The CMAC overcomes the limitations of the Data Authentication Algorithm (DAA) which is based on DES.

CMACs (Cipher-based message authentication codes) create message authentication codes (MACs) using a block cipher and a secret key. They differ from HMACs in that they use a block symmetric key method for the MACs rather than a hashing method.

Generally **CMAC will be slower than HMAC**, as hashing methods are generally faster than block cipher methods. In most cases HMAC will work best, but CMAC may work better where there is embedded hardware which has hardware acceleration for block ciphers. For this, CMAC would likely run faster than HMAC.

The operation of the CMAC can be defined as follows: when the message is an integer multiple n of the cipher block length b. For AES, b = 128, and for tripleDES, b = 64. The message is divided into n blocks ($M_1$, $M_2$,…, $M_n$). The algorithm makes use of a k-bit encryption key K and a b-bit constant, $K_1$. For AES, the key size k is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits.

$$
\begin{aligned}
C_1 &= E(K, M_1) \\
C_2 &= E(K, [M_2 \oplus C_1]) \\
C_3 &= E(K, [M_3 \oplus C_2]) \\
&\vdots \\
C_n &= E(K, [M_n \oplus C_{n-1} \oplus K_1]) \\
T &= \text{MSB}_{Tlen}(C_n)
\end{aligned}
$$

where

$$
\begin{aligned}
T &= \text{message authetication code, also referred to as the tag} \\
Tlen &= \text{bit length of T} \\
\text{MSB}_s(X) &= \text{the } s \text{ leftmost bits of the bit string } X
\end{aligned}
$$

If the message is not an integer multiple of the cipher block length, then the final block is padded to the right (least significant bits) with a 1 and as many 0s as necessary so that the final block is also of length $b$. The CMAC operation then pro- ceeds as before, except that a different $n$-bit key $K2$ is used instead of $K1$.

$$
\begin{aligned}
L &= E(K, 0^n) \\
K_1 &= L \bullet x \\
K_2 &= L \bullet x^2 = (L \bullet x) \bullet x
\end{aligned}
$$

**PROGRAM:**
```cpp
#include "cryptlib.h"
#include "secblock.h"
#include "osrng.h"
#include "files.h"
#include "cmac.h"
#include "aes.h"
#include "hex.h"
using namespace CryptoPP;

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

int main(int argc, char* argv[])
{
    AutoSeededRandomPool prng;

    SecByteBlock key(AES::DEFAULT_KEYLENGTH);
    prng.GenerateBlock(key, key.size());

    string mac, plain = "CMAC Test";
    HexEncoder encoder(new FileSink(cout));


    // Pretty print key
    cout << "key: ";
    encoder.Put(key, key.size());
    encoder.MessageEnd();
    cout << endl;

    cout << "plain text: ";
    encoder.Put((const byte*)plain.data(), plain.size());
    encoder.MessageEnd();
    cout << endl;

    try
    {
        CMAC<AES> cmac(key.data(), key.size());
        cmac.Update((const byte*)plain.data(), plain.size());

        mac.resize(cmac.DigestSize());
        cmac.Final((byte*)&mac[0]);
    }
    catch(const CryptoPP::Exception& e)
    {
        cerr << e.what() << endl;
        exit(1);
    }
```

```cpp
    // Pretty print
    cout << "cmac: ";
    encoder.Put((const byte*)mac.data(), mac.size());
    encoder.MessageEnd();
    cout << endl;

    // Verify
    try
    {
        CMAC<AES> cmac(key.data(), key.size());
        cmac.Update((const byte*)plain.data(), plain.size());

        // Call Verify() instead of Final()
        bool verified = cmac.Verify((byte*)&mac[0]);
        if (!verified)
            throw Exception(Exception::DATA_INTEGRITY_CHECK_FAILED, "CMAC: message MAC
not valid");

        cout << "Verified message MAC" << endl;
    }
    catch(const CryptoPP::Exception& e)
    {
        cerr << e.what() << endl;
        exit(1);
    }

    return 0;
}
```

**OUTPUT:**

```
$ ./test.exe
key: 54FE5717559053CF76A14C86582B1892
plain text: 434D41432054657374
cmac: 74A8A4E4200D945BECCA16314C3B4ED8
Verified message MAC
```

1.What is the HMAC algorithm?

2. What are the types of authentication functions in the HMAC?

3. What is the formula for the HMAC?

4. What is MAC and define the types of MAC?

5. Define CMAC algorithm?

6. What the limitations of DAA in DES and how CMAC overcome the limitations?

7. What methods are used in the HMAC?

**Week - 6 Hash Function**

**6.1) Write a C Program to implement SHA-512 Algorithm.**

**AIM:** To write a C Program to implement SHA-512 Algorithm.

**DESCRIPTION:**

SHA-512 is a member of the NIST-standardized SHA-2 family of cryptographic hash functions that produces a 512-bit digest and, therefore, provides 256 bits of security against collisions . The input message can have a length of up to 2 128 − 1 bits and is processed in blocks of 1024 bits.

Like other members of the SHA-2 family, SHA-512 is based on the well-known Merkle-Damgård structure with a Davies-Meyer compression function that uses solely Boolean operations (i.e. bitwise AND, XOR, OR, and NOT), modular additions, as well as shifts and rotations. All operations are applied to 64-bit words.

**SHA-512 consists of two stages:** preprocessing and hash computation**.** In the former stage, the eight working variables, denoted as a, b, c, d, e, f, g, and h in [17], are initialized to certain fixed constants. Furthermore, the input message is padded and then divided into 1024-bit blocks. The actual **hash computation** passes each message block (represented by 16 words m0, m1, . . . m15 of 64 bits each) through a message schedule to expand them to 80 words wi with $0 \le i \le 79$.

Then, the eight working variables are updated using a compression function that consists of 80 rounds. A round of the compression function is exemplarily depicted in Fig. 2. The **processing** of a 1024-bit message block results in eight 64-bit intermediate hash values. After the whole message has been processed, the 512-bit digest is generated by simply concatenating the eight intermediate hash values.
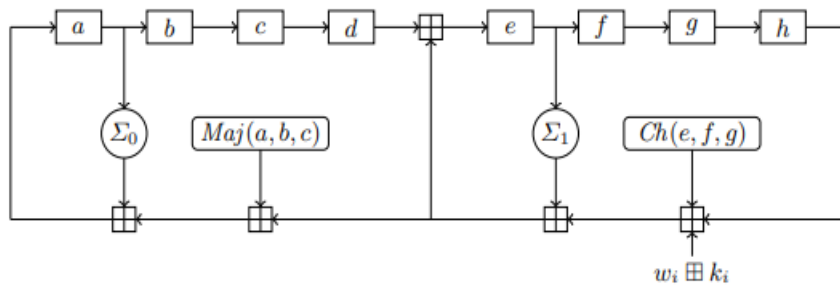


$$\Sigma_{0,i} = (a_i \ggg 28) \oplus (a_i \ggg 34) \oplus (a_i \ggg 39)$$
$$Maj_i = (a_i \wedge b_i) \oplus (a_i \wedge c_i) \oplus (b_i \wedge c_i)$$
$$t_{2,i} = \Sigma_{0,i} \boxplus Maj_i$$
$$\Sigma_{1,i} = (e_i \ggg 14) \oplus (e_i \ggg 18) \oplus (e_i \ggg 41)$$
$$Ch_i = (e_i \wedge f_i) \oplus (\bar{e}_i \wedge g_i)$$
$$t_{1,i} = h_i \boxplus \Sigma_{1,i} \boxplus Ch_i \boxplus k_i \boxplus w_i$$
$$(h_{i+1}, g_{i+1}, f_{i+1}, e_{i+1}) = (g_i, f_i, e_i, d_i \boxplus t_{1,i})$$
$$(d_{i+1}, c_{i+1}, b_{i+1}, a_{i+1}) = (c_i, b_i, a_i, t_{1,i} \boxplus t_{2,i})$$

**PROGRAM:**

```c
#include <stdlib.h>
#include <string.h>

#include "SHA512.h"
#include "config.h"

// K: first 64 bits of the fractional parts of the cube roots of the first 80 primes
const static uint64_t K[80] =
{
   0x428A2F98D728AE22, 0x7137449123EF65CD, 0xB5C0FBCFEC4D3B2F,
0xE9B5DBA58189DBBC,
   0x3956C25BF348B538, 0x59F111F1B605D019, 0x923F82A4AF194F9B, 0xAB1C5ED5DA6D8118,
   0xD807AA98A3030242, 0x12835B0145706FBE, 0x243185BE4EE4B28C, 0x550C7DC3D5FFB4E2,
   0x72BE5D74F27B896F, 0x80DEB1FE3B1696B1, 0x9BDC06A725C71235, 0xC19BF174CF692694,
   0xE49B69C19EF14AD2, 0xEFBE4786384F25E3, 0x0FC19DC68B8CD5B5,
0x240CA1CC77AC9C65,
   0x2DE92C6F592B0275, 0x4A7484AA6EA6E483, 0x5CB0A9DCBD41FBD4,
0x76F988DA831153B5,
   0x983E5152EE66DFAB, 0xA831C66D2DB43210, 0xB00327C898FB213F, 0xBF597FC7BEEF0EE4,
   0xC6E00BF33DA88FC2, 0xD5A79147930AA725, 0x06CA6351E003826F, 0x142929670A0E6E70,
   0x27B70A8546D22FFC, 0x2E1B21385C26C926, 0x4D2C6DFC5AC42AED, 0x53380D139D95B3DF,
   0x650A73548BAF63DE, 0x766A0ABB3C77B2A8, 0x81C2C92E47EDAEE6, 0x92722C851482353B,
   0xA2BFE8A14CF10364, 0xA81A664BBC423001, 0xC24B8B70D0F89791, 0xC76C51A30654BE30,
   0xD192E819D6EF5218, 0xD69906245565A910, 0xF40E35855771202A, 0x106AA07032BBD1B8,
   0x19A4C116B8D2D0C8, 0x1E376C085141AB53, 0x2748774CDF8EEB99, 0x34B0BCB5E19B48A8,
   0x391C0CB3C5C95A63, 0x4ED8AA4AE3418ACB, 0x5B9CCA4F7763E373,
0x682E6FF3D6B2B8A3,
   0x748F82EE5DEFB2FC, 0x78A5636F43172F60, 0x84C87814A1F0AB72, 0x8CC702081A6439EC,
   0x90BEFFFA23631E28, 0xA4506CEBDE82BDE9, 0xBEF9A3F7B2C67915, 0xC67178F2E372532B,
   0xCA273ECEEA26619C, 0xD186B8C721C0C207, 0xEADA7DD6CDE0EB1E,
0xF57D4F7FEE6ED178,
   0x06F067AA72176FBA, 0x0A637DC5A2C898A6, 0x113F9804BEF90DAE, 0x1B710B35131C471B,
   0x28DB77F523047D84, 0x32CAAB7B40C72493, 0x3C9EBE0A15C9BEBC,
0x431D67C49C100D4C,
   0x4CC5D4BECB3E42B6, 0x597F299CFC657E2A, 0x5FCB6FAB3AD6FAEC,
0x6C44198C4A475817
 };

// Utility functions
// Rotate x to the right by numBits
#define ROTR(x, numBits) ( (x >> numBits) | (x << (64 - numBits)) )

// Compression functions
#define Ch(x,y,z) ( (x & y) ^ ((~x) & z) )
#define Maj(x,y,z) ( (x & y) ^ (x & z) ^ (y & z) )

#define BigSigma0(x) ( ROTR(x,28) ^ ROTR(x,34) ^ ROTR(x,39) )
```

```c
#define BigSigma1(x) ( ROTR(x,14) ^ ROTR(x,18) ^ ROTR(x,41) )

#define SmallSigma0(x) ( ROTR(x,1) ^ ROTR(x,8) ^ (x >> 7) )
#define SmallSigma1(x) ( ROTR(x,19) ^ ROTR(x,61) ^ (x >> 6) )

// SHA512 message schedule
// Calculate the Nth block of W
uint64_t *W(int N, uint64_t *M)
{
   uint64_t *w = (uint64_t*) malloc(sizeof(uint64_t) * 80);
   uint64_t *mPtr = &M[(N * 16)];

   //printf("Message block %d : ", N);
   for (int i = 0; i < 16; ++i)
   {
      w[i] = *mPtr;
      ++mPtr;

   //printf("%" PRIx64 , w[i]);
   }
   //printf("\n");
   for (int i = 16; i < 80; ++i)
   {
      w[i] = SmallSigma1(w[i - 2]) + w[i - 7] + SmallSigma0(w[i - 15]) + w[i - 16];
   }
   return w;
}

// Step 1:
// Preprocesses a given message of l bits.
// Appends "1" to end of msg, then k 0 bits such that l + 1 + k = 896 mod 1024
// and k is the smallest nonnegative solution to said equation. To this is appended
// the 128 bit block equal to the bit length l.
//char *preprocess(char *msg)
PaddedMsg preprocess(uint8_t *msg, size_t len)
{
   PaddedMsg padded;

   // resulting msg wll be multiple of 1024 bits
   //size_t len = strlen(msg);
   if (msg == NULL || len == 0)
   {
      padded.length = 0;
      padded.msg = NULL;
      return padded;
   }

   size_t l = len * 8;
   size_t k = (896 - ( (l  + 1) % 1024 )) % 1024;
```

```c
    //printf("k = %zu\n", k);
    //printf("l = %zu\n", l);
    //printf("l + k + 1 = %zu bits, %zu bytes\n", (l+k+1), ((l+k+1)/8));

    padded.length = ((l + k + 1) / 8) + 16;
    //printf("padded.length = %zu\n", padded.length);
    padded.msg = (uint8_t*) malloc(sizeof(uint8_t) * padded.length);
    memset(&padded.msg[0], 0, padded.length);
    for (size_t i = 0; i < len; ++i)
        padded.msg[i] = msg[i];
    // append to the binary string a 1 followed by k zeros
    padded.msg[len] = 0x80;

    // last 16 bytes reserved for length
    __uint128_t bigL = l;
    endianSwap128(&bigL);
    memcpy(&padded.msg[padded.length - sizeof(__uint128_t)], &bigL, sizeof(__uint128_t));

    return padded;
}

// Step 2: Parse the padded message into N 1024-bit blocks | Each block separated into 64-bit words
// (therefore 16 per block) | Returns an array of 8 64 bit words corresponding to the hashed value
uint64_t *getHash(PaddedMsg *p)
{
    size_t N = p->length / SHA512_MESSAGE_BLOCK_SIZE;
    //printf("Number of blocks = %zu\n", N);

    // initial hash value
    uint64_t h[8] = {
        0x6A09E667F3BCC908,
        0xBB67AE8584CAA73B,
        0x3C6EF372FE94F82B,
        0xA54FF53A5F1D36F1,
        0x510E527FADE682D1,
        0x9B05688C2B3E6C1F,
        0x1F83D9ABFB41BD6B,
        0x5BE0CD19137E2179
    };

#if MACHINE_BYTE_ORDER == LITTLE_ENDIAN
    // Convert byte order of message to big endian
    uint64_t *msg = ((uint64_t*)&p->msg[0]);
    for (int i = 0; i < N * 16; ++i)
        endianSwap64(msg++);
#endif

    for (size_t i = 0; i < N; ++i)
    {
```

```c
    uint64_t T1, T2;
    // initialize registers
    uint64_t reg[HASH_ARRAY_LEN];
    for (int i = 0; i < HASH_ARRAY_LEN; ++i)
        reg[i] = h[i];

    uint64_t *w = W(i, ((uint64_t*)(p->msg)));
    for (int j = 0; j < 80; ++j)
    {
        T1 = reg[7] + BigSigma1(reg[4]) + Ch(reg[4], reg[5], reg[6]) + K[j] + w[j];
        T2 = BigSigma0(reg[0]) + Maj(reg[0], reg[1], reg[2]);

        reg[7] = reg[6];
        reg[6] = reg[5];
        reg[5] = reg[4];
        reg[4] = reg[3] + T1;
        reg[3] = reg[2];
        reg[2] = reg[1];
        reg[1] = reg[0];
        reg[0] = T1 + T2;
    }
    for (int i = 0; i < HASH_ARRAY_LEN; ++i)
        h[i] += reg[i];

    free(w);
  }
  free(p->msg);

  uint64_t *retVal = (uint64_t*) malloc(sizeof(uint64_t) * HASH_ARRAY_LEN);
  memcpy(retVal, h, sizeof(uint64_t) * HASH_ARRAY_LEN);
  return retVal;
}
uint64_t *SHA512Hash(uint8_t *input, size_t len)
{
  PaddedMsg paddedMsg = preprocess(input, len);
  return getHash(&paddedMsg);
}
```

**OUTPUT:**

*Input:* S = "hello world"

*Output:*

309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f989dd35bc5ff4

99670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f

1.What is the SHA algorithm?

2. What are the types of SHA algorithm?

3. What mechanism is used in the SHA-512?

4. What is difference between SHA-512 and SHA-256?

5. Explain the stages in the SHA-512 algorithm?

**Week - 7 TCP Server Applications**
**7.1) Design TCP iterative Client and server application to reverse the given input sentence.**

**AIM:** To design TCP iterative Client and server application to reverse the given input sentence.

**DESCRIPTION:**

Here we will see how we can create a system, where we will create one client, and a server, and the client can send one string to the server, and the server will reverse the string, and return back to the client.

Here we will use the **concept of socket programming**. To make the client server connection, we have to create port. The port number is one arbitrary number that can be used by the socket. We have to use the same port for client and the server to establish the connection.

**Socket function:** The protocol argument to the socket function is set to zero except for raw sockets.

```
#include <sys/socket.h>

int socket int family, int type, int protocol);
```

**Connect function:** The connect function is used by a TCP client to establish a connection with a TCP server.

**int connect(int sockfd, const struct sockaddr \*servaddr, socklen_t addrlen);**

**Close function:** The normal UNIX close function is also used to close a socket and terminate a TCP connection.

**Listen function:** The second argument to this function specifies the maximum number of connections that the kernel should queue for this socket.

**int listen(int sockfd, int backlog);**

**Accept function:** The cliaddr and addrlen argument are used to ret urn the protocol address of the connected peer processes (client).

**Bzero:** It sets the specified number of bytes to 0(zero) in the destination. We often use this function to initialize a socket address structure to 0(zero).

```
#include<strings.h>
void bzer(void *dest,size_t nbytes);
```

**Bind function:** The bind function assigns alocal protocol address to a socket.

**int bind(int sockfd, const struct sockaddr \*myaddr, s ocklen_t addrlen);**

To start the program, start the server program first –
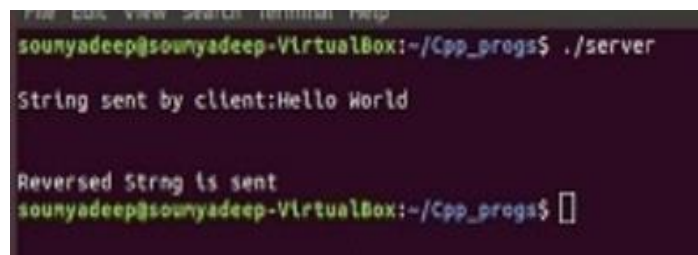
**gcc Server.c –o server**

Then start client program –

**gcc Client.c –o server**

**PROGRAM:**

**TCP Server:**
```c
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#define MAXLINE 20
#define SERV_PORT 5777
main(int argc,char *argv) {
    int i,j;
    ssize_t n;
    char line[MAXLINE];
    char revline[MAXLINE];
    int listenfd,connfd,clilen;
    struct sockaddr_in servaddr,cliaddr;
    listenfd=socket(AF_INET,SOCK_STREAM,0);
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family=AF_INET; servaddr.sin_port=htons(SERV_PORT);
    bind(listenfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
    listen(listenfd,1);
    for( ; ; ) {
        clilen=sizeof(cliaddr);
        connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&clilen);
        printf("connect to client");
        while(1)  {
            if((n=read(connfd,line,MAXLINE))==0)
            break;
            line[n-1]='\0';
            j=0;
            for(i=n-2;i>=0;i--)
                    revline[j++]=line[i];
                    revline[j]='\0';
                    write(connfd,revline,n);
        }
    }
}
```

**OUTPUT:**
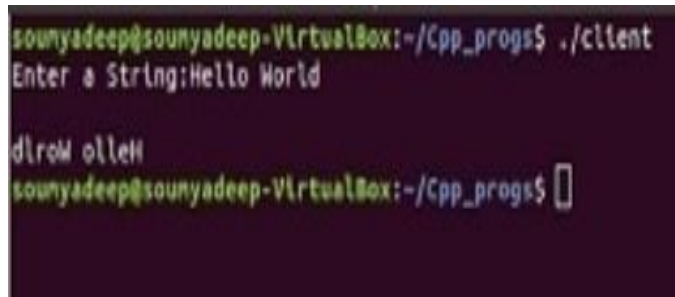
## TCP Client:

```c
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#define MAXLINE 20
#define SERV_PORT 5777
main(int argc,char *argv)
{
        char sendline[MAXLINE],revline[MAXLINE];
        int sockfd;
        struct sockaddr_in servaddr;
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=ntohs(SERV_PORT);
        connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
        printf("\n enter the data to be send");
        while(fgets(sendline,MAXLINE,stdin)!=NULL)
        {
                write(sockfd,sendline,strlen(sendline));
                printf("\n line send");
                read(sockfd,revline,MAXLINE);
                printf("\n reverse of the given sentence is : %s",revline);
                printf("\n");
        }
exit(0);
}
```

**OUTPUT:**

**7.2) Design TCP client and server application to transfer file.**

**AIM:** To design TCP client and server application to transfer file.

**DESCRIPTION:**

If we are creating a connection between client and server using TCP then it has a few functionalities like, TCP is suited for applications that require high reliability, and transmission time is relatively less critical. It is used by other protocols like HTTP, HTTPs, FTP, SMTP, Telnet.

TCP rearranges data packets in the order specified. There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent. TCP does Flow Control and requires three packets to set up a socket connection before any user data can be sent. TCP handles reliability and congestion control. It also does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination.

The entire process can be broken down into the following steps:
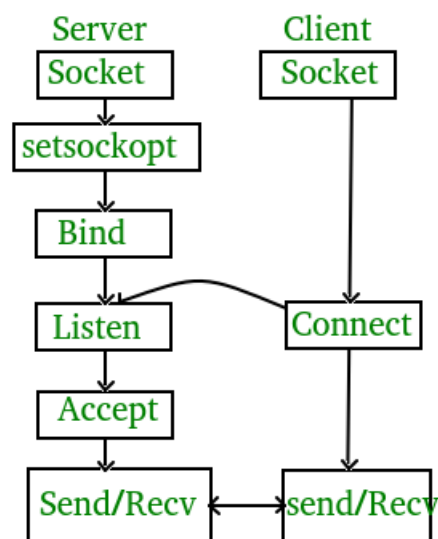


**TCP Server –**
1. using create(), Create TCP socket.
2. using bind(), Bind the socket to server address.
3. using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. using accept(), At this point, connection is established between client and server, and they are ready to transfer data.
5. Go back to Step 3.

**TCP Client –**
1. Create TCP socket.
2. connect newly created client socket to server.

## TCP Server:

```c
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h> // read(), write(), close()
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int connfd)        // Function designed for chat between client and server.
{
        char buff[MAX];
        int n;
        // infinite loop for chat
        for (;;) {
                bzero(buff, MAX);
                read(connfd, buff, sizeof(buff));       // read the message from client and copy it in buffer
                printf("From client: %s\t To client : ", buff);
                bzero(buff, MAX);
                n = 0;
                while ((buff[n++] = getchar()) != '\n')         // copy server message in the buffer
                write(connfd, buff, sizeof(buff));              // and send that buffer to client

                // if msg contains "Exit" then server exit and chat ended.
                if (strncmp("exit", buff, 4) == 0) {
                        printf("Server Exit...\n");
                        break;
                }
        }
}

// Driver function
int main()
{
        int sockfd, connfd, len;
        struct sockaddr_in servaddr, cli;

        // socket create and verification
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd == -1) {
                printf("socket creation failed...\n");
                exit(0);
        }
        else
```

```c
        printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
}
else
        printf("Socket successfully binded..\n");

// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
        printf("Listen failed...\n");
        exit(0);
}
else
        printf("Server listening..\n");
len = sizeof(cli);

// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
        printf("server accept failed...\n");
        exit(0);
}
else
        printf("server accept the client...\n");
func(connfd);                     // Function for chatting between client and server
close(sockfd);
}
```

**OUTPUT:**

```
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
From client: hi
      To client : hello
From client: exit
      To client : exit
Server Exit...
```

## TCP Client:

```c
#include <arpa/inet.h> // inet_addr()

#include <netdb.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h> // bzero()

#include <sys/socket.h>

#include <unistd.h> // read(), write(), close()

#define MAX 80

#define PORT 8080

#define SA struct sockaddr

void func(int sockfd)

{

        char buff[MAX];

        int n;

        for (;;) {

                bzero(buff, sizeof(buff));

                printf("Enter the string : ");

                n = 0;

                while ((buff[n++] = getchar()) != '\n')

                        ;

                write(sockfd, buff, sizeof(buff));

                bzero(buff, sizeof(buff));

                read(sockfd, buff, sizeof(buff));

                printf("From Server : %s", buff);

                if ((strncmp(buff, "exit", 4)) == 0) {

                        printf("Client Exit...\n");
```

```c
                break;

        }

    }

}


int main()

{

    int sockfd, connfd;

    struct sockaddr_in servaddr, cli;


    // socket create and verification

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockfd == -1) {

            printf("socket creation failed...\n");

            exit(0);

    }

    else

            printf("Socket successfully created..\n");

    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT

    servaddr.sin_family = AF_INET;

    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    servaddr.sin_port = htons(PORT);

    // connect the client socket to server socket

    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr))

            != 0) {

            printf("connection with the server failed...\n");

            exit(0);
```

```
    }
    else
            printf("connected to the server..\n");
    // function for chat
    func(sockfd);
    // close the socket
    close(sockfd);
}
```

**OUTPUT:**

```
Socket successfully created..
connected to the server..
Enter the string : hi
From Server : hello
Enter the string : exit
From Server : exit
Client Exit...
```

1.What is the TCP?

2. What is client and what is server systems?

3. How to connect the client and server?

4. What is the networking?

5. Explain connect, bind, listen statements?

6. What is the command to start the server system in TCP?

7. What is the command to start the client system in TCP?

**Week - 8 TCP Concurrent Server Applications**

**8.1) Design a TCP concurrent server to convert a given text into upper case using multiplexing system call "select".**

**AIM:** To design a TCP concurrent server to convert a given text into upper case using multiplexing system call "select".

**DESCRIPTION:**

When the TCP client is handling two inputs at the same time: standard input and a TCP socket, we encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later).

The **Select** function is used to select between TCP and UDP sockets. This function gives instructions to the kernel to wait for any of the multiple events to occur and awakens the process only after one or more events occur or a specified time passes.

**TCP Client:**
1. Create a TCP socket.
2. Call connect to establish a connection with the server.
3. When the connection is accepted write a message to a server.
4. Read the response of the Server.
5. Close socket descriptor and exit.

**select Function**

The select function allows the process to instruct the kernel to either:

- Wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs, or
- When a specified amount of time has passed.

This means that we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using select.

Any of the middle three arguments to **select,** *readset***,** *writeset***, or** *exceptset***,** can be specified as a null pointer if we are not interested in that condition. Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix sleep function. The poll function provides similar functionality.

**Return value of select:**

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

**PROGRAM:**

**TCP Server:**

```c
#include <arpa/inet.h>

#include <errno.h>

#include <netinet/in.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <strings.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <unistd.h>
#define PORT 5000
#define MAXLINE 1024
int max(int x, int y)
{
        if (x > y)
                return x;
        else
                return y;
}
int main()
{
        int listenfd, connfd, udpfd, nready, maxfdp1;
        char buffer[MAXLINE];
        pid_t childpid;
        fd_set rset;
        ssize_t n;
```

```c
socklen_t len;

const int on = 1;

struct sockaddr_in cliaddr, servaddr;

char* message = "Hello Client";

void sig_chld(int);


/* create listening TCP socket */

listenfd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

servaddr.sin_port = htons(PORT);


// binding server addr structure to listenfd

bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));

listen(listenfd, 10);


/* create UDP socket */

udpfd = socket(AF_INET, SOCK_DGRAM, 0);

// binding server addr structure to udp sockfd

bind(udpfd, (struct sockaddr*)&servaddr, sizeof(servaddr));


// clear the descriptor set

FD_ZERO(&rset);

// get maxfd

maxfdp1 = max(listenfd, udpfd) + 1;

for (;;) {

        // set listenfd and udpfd in readset
```

```
FD_SET(listenfd, &rset);

FD_SET(udpfd, &rset);

// select the ready descriptor

nready = select(maxfdp1, &rset, NULL, NULL, NULL);

// if tcp socket is readable then handle

// it by accepting the connection

if (FD_ISSET(listenfd, &rset)) {

        len = sizeof(cliaddr);

        connfd = accept(listenfd, (struct sockaddr*)&cliaddr, &len);

        if ((childpid = fork()) == 0) {

                close(listenfd);

                bzero(buffer, sizeof(buffer));

                printf("Message From TCP client: ");

                read(connfd, buffer, sizeof(buffer));

                puts(buffer);

                write(connfd, (const char*)message, sizeof(buffer));

                close(connfd);

                exit(0);

        }

        close(connfd);

}

// if udp socket is readable receive the message.

if (FD_ISSET(udpfd, &rset)) {

        len = sizeof(cliaddr);

        bzero(buffer, sizeof(buffer));

        printf("\nMessage from UDP client: ");

        n = recvfrom(udpfd, buffer, sizeof(buffer), 0, (struct sockaddr*)&cliaddr, &len);

        puts(buffer);
```

```
                        sendto(udpfd, (const char*)message, sizeof(buffer), 0,

                            (struct sockaddr*)&cliaddr, sizeof(cliaddr));

            }

        }

}
```

**TCP Client:**

```c
#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#define PORT 5000

#define MAXLINE 1024

int main()

{

        int sockfd;

        char buffer[MAXLINE];

        char* message = "Hello Server";

        struct sockaddr_in servaddr;

        int n, len;

        // Creating socket file descriptor

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {

                printf("socket creation failed");

                exit(0);

        }

        memset(&servaddr, 0, sizeof(servaddr));

        // Filling server information
```
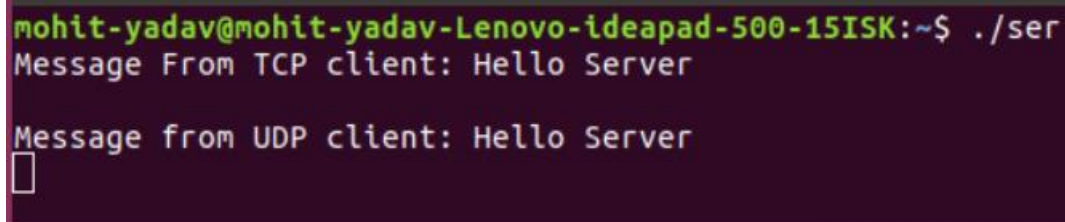
```
servaddr.sin_family = AF_INET;

servaddr.sin_port = htons(PORT);

servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");

if (connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0) {

        printf("\n Error : Connect Failed \n");

}

memset(buffer, 0, sizeof(buffer));

strcpy(buffer, "Hello Server");

write(sockfd, buffer, sizeof(buffer));

printf("Message from server: ");

read(sockfd, buffer, sizeof(buffer));

puts(buffer);

close(sockfd);
```
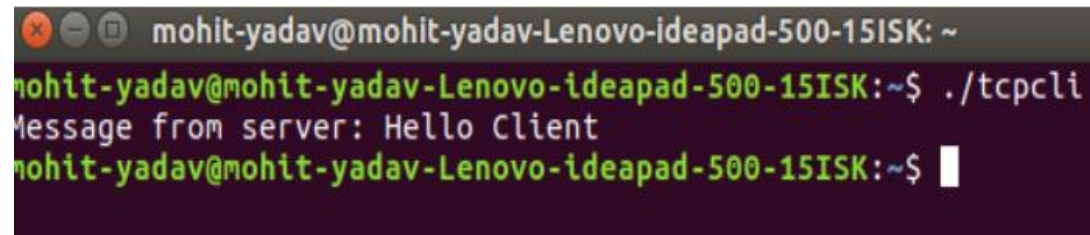
}

**OUTPUT:**

**Server:**



**TCP Client:**

**8.2) Design a TCP concurrent server to echo given set of sentences using poll functions.**

**AIM:** To design a TCP concurrent server to echo given set of sentences using poll functions.

**DESCRIPTION:**

When the TCP client is handling two inputs at the same time: standard input and a TCP socket, we encountered a problem when the client was blocked in a call to fgets (on standard input) and the server process was killed. The server TCP correctly sent a FIN to the client TCP, but since the client process was blocked reading from standard input, it never saw the EOF until it read from the socket (possibly much later).

We want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output). This capability is called **I/O multiplexing** and is provided by the **select and poll functions**, as well as a newer POSIX variation of the former, called **pselect.**

**pselect Function:** The pselect function was invented by POSIX and is now supported by many of the Unix variants.

**poll Function:** poll provides functionality that is similar to select, but poll provides additional information when dealing with STREAMS devices.

**Arguments:** The first argument (*fdarray*) is a pointer to the first element of an array of structures. Each element is a pollfd structure that specifies the conditions to be tested for a given descriptor, fd.

| Constant | Input to events ? | Result from revents ? | Description |
|---|---|---|---|
| POLLIN | • | • | Normal or priority band data can be read |
| POLLRDNORM | • | • | Normal data can be read |
| POLLRDBAND | • | • | Priority band data can be read |
| POLLPRI | • | • | High-priority data can be read |
| POLLOUT | • | • | Normal data can be written |
| POLLWRNORM | • | • | Normal data can be written |
| POLLWRBAND | • | • | Priority band data can be written |
| POLLERR | | • | Error has occurred |
| POLLHUP | | • | Hangup has occurred |
| POLLNVAL | | • | Descriptor is not an open file |

**Return values from poll:**

- –1 if an error occurred
- 0 if no descriptors are ready before the timer expires
- Otherwise, it is the number of descriptors that have a nonzero revents member.

**PROGRAM:**

```c
#include <stdio.h>

#include <stdlib.h>

#include <sys/ioctl.h>

#include <sys/poll.h>

#include <sys/socket.h>

#include <sys/time.h>

#include <netinet/in.h>

#include <errno.h>

#define SERVER_PORT  12345


#define TRUE           1
#define FALSE          0


main (int argc, char *argv[])
{
  int   len, rc, on = 1;
  int   listen_sd = -1, new_sd = -1;
  int   desc_ready, end_server = FALSE, compress_array = FALSE;
  int   close_conn;
  char  buffer[80];
  struct sockaddr_in6   addr;
  int   timeout;
  struct pollfd fds[200];
  int   nfds = 1, current_size = 0, i, j;


  listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
  if (listen_sd < 0)
```

```c
  {
    perror("socket() failed");
    exit(-1);
  }
  rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
            (char *)&on, sizeof(on));
  if (rc < 0)
  {
    perror("setsockopt() failed");
    close(listen_sd);
    exit(-1);
  }
  rc = ioctl(listen_sd, FIONBIO, (char *)&on);
  if (rc < 0)
  {
    perror("ioctl() failed");
    close(listen_sd);
    exit(-1);
  }
  memset(&addr, 0, sizeof(addr));
  addr.sin6_family      = AF_INET6;
  memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
  addr.sin6_port        = htons(SERVER_PORT);
  rc = bind(listen_sd,
        (struct sockaddr *)&addr, sizeof(addr));
  if (rc < 0)
  {
    perror("bind() failed");
```

```c
      close(listen_sd);
      exit(-1);
   }
   rc = listen(listen_sd, 32);
   if (rc < 0)
   {
      perror("listen() failed");
      close(listen_sd);
      exit(-1);
   }
   memset(fds, 0 , sizeof(fds));
   fds[0].fd = listen_sd;
   fds[0].events = POLLIN;
   timeout = (3 * 60 * 1000);
   do
   {
      printf("Waiting on poll()...\n");
      rc = poll(fds, nfds, timeout);
      if (rc < 0)
      {
         perror("  poll() failed");
         break;
      }
      if (rc == 0)
      {
         printf("  poll() timed out.  End program.\n");
         break;
      }
```

```c
current_size = nfds;
for (i = 0; i < current_size; i++)
{
  if(fds[i].revents == 0)
    continue;
  if(fds[i].revents != POLLIN)
  {
    printf("  Error! revents = %d\n", fds[i].revents);
    end_server = TRUE;
    break;
  }
  if (fds[i].fd == listen_sd)
  {
    printf("  Listening socket is readable\n");
    do
    {
      new_sd = accept(listen_sd, NULL, NULL);
      if (new_sd < 0)
      {
        if (errno != EWOULDBLOCK)
        {
          perror("  accept() failed");
          end_server = TRUE;
        }
        break;
      }
      printf("  New incoming connection - %d\n", new_sd);
      fds[nfds].fd = new_sd;
```

```
      fds[nfds].events = POLLIN;

      nfds++;

   } while (new_sd != -1);

}

else

{

  printf("  Descriptor %d is readable\n", fds[i].fd);

  close_conn = FALSE;

  do

  {

    rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);

    if (rc < 0)

    {

      if (errno != EWOULDBLOCK)

      {

        perror("  recv() failed");

        close_conn = TRUE;

      }

      break;

    }

    if (rc == 0)

    {

      printf("  Connection closed\n");

      close_conn = TRUE;

      break;

    }

    len = rc;

    printf("  %d bytes received\n", len);
```
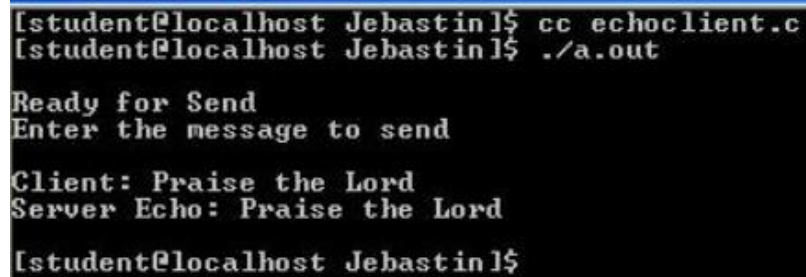
```
          rc = send(fds[i].fd, buffer, len, 0);

          if (rc < 0)

          {

            perror("  send() failed");

            close_conn = TRUE;

            break;

          }

        } while(TRUE);

        if (close_conn)

        {

          close(fds[i].fd);

          fds[i].fd = -1;

          compress_array = TRUE;

        }

      }  /* End of existing connection is readable        */

    } /* End of loop through pollable descriptors          */

    if (compress_array)

    {

      compress_array = FALSE;

      for (i = 0; i < nfds; i++)

      {

        if (fds[i].fd == -1)

        {

          for(j = i; j < nfds; j++)

          {

            fds[j].fd = fds[j+1].fd;

          }

          i--;
```

```
          nfds--;

        }

      }

    }

  } while (end_server == FALSE); /* End of serving running.    */

  for (i = 0; i < nfds; i++)

  {

    if(fds[i].fd >= 0)

      close(fds[i].fd);

  }

}
```

**OUTPUT:**

```
[student@localhost Jebastin]$ cc echoclient.c
[student@localhost Jebastin]$ ./a.out

Ready for Send
Enter the message to send

Client: Praise the Lord
Server Echo: Praise the Lord

[student@localhost Jebastin]$
```

**1.What is the TCP concurrent server?**

**2. What are the TCP concurrent server applications?**

**3. What is multiplexing?**

**4. Explain the "Select" function?**

**5. Explain the "poll" function?**

**6. What are the return values of the poll function?**

**7. What are the 3 main arguments of the select function?**

**Week – 9 UDP Applications**

**9.1) Design UDP Client and server application to reverse the given input sentence.**

**AIM:** To design UDP Client and server application to reverse the given input sentence.

**DESCRIPTION:**

In UDP, the client does not form a connection with the server like in TCP and instead sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of the sender which the server uses to send data to the correct client.
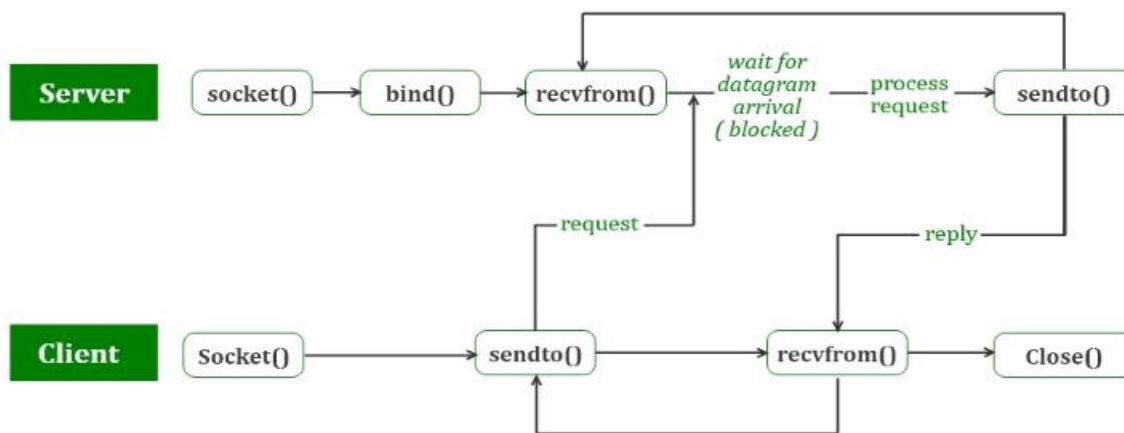


The entire process can be broken down into the following steps :

**UDP Server :**
1. Create a UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.
5. Go back to Step 3.

**UDP Client :**
1. Create a UDP socket.
2. Send a message to the server.
3. Wait until a response from the server is received.
4. Process the reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

**Advantages of using UDP**

- Faster than TCP

- Does not restrict you to connection based communication model

- Can be used to send large data into smaller packets quickly

**PROGRAM:**

<u>**UDP Client:**</u>

```c
#include <sys/socket.h>

#include <netdb.h>

#include <string.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>


#define S_PORT 43454

#define C_PORT 43455

#define ERROR -1

#define IP_STR "127.0.0.1"


int main(int argc, char const *argv[]) {
        int sfd, len;
        char str_buf[2048];
        struct sockaddr_in servaddr, clientaddr;
        socklen_t addrlen;
        sfd = socket(AF_INET, SOCK_DGRAM,IPPROTO_UDP);
        if (sfd == ERROR) {
                perror("Could not open a socket");
                return 1;
        }
        memset((char *) &servaddr, 0, sizeof(servaddr));
```

```c
        servaddr.sin_family=AF_INET;

        servaddr.sin_addr.s_addr=inet_addr(IP_STR);

        servaddr.sin_port=htons(S_PORT);


        memset((char *) &clientaddr, 0, sizeof(clientaddr));

        clientaddr.sin_family=AF_INET;

        clientaddr.sin_addr.s_addr=inet_addr(IP_STR);

        clientaddr.sin_port=htons(C_PORT);


        if((bind(sfd,(struct sockaddr *)&clientaddr,sizeof(clientaddr)))!=0) {

                perror("Could not bind socket");

                return 2;

        }


        printf("Client is running on %s:%d\n", IP_STR, C_PORT);

        printf("Enter a string: ");

        scanf("%[^\n]%*c",str_buf);

        len = strlen(str_buf);

        sendto(sfd, &len, sizeof(len), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));

        sendto(sfd, str_buf, len, 0, (struct sockaddr *)&servaddr, sizeof(servaddr));

        addrlen = sizeof(clientaddr);

        recvfrom(sfd, &len, sizeof(len), 0, (struct sockaddr *)&clientaddr, &addrlen);

        recvfrom(sfd, str_buf, len, 0, (struct sockaddr *)&clientaddr, &addrlen);

        printf("Server Replied: %s\n", str_buf);


        return 0;

}
```

**UDP Server:**

```c
#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>


#define S_PORT 43454

#define C_PORT 43455

#define ERROR -1

#define IP_STR "127.0.0.1"


void strrev(char *str, int len) {
    int i, j;
    char temp;
    for (i = 0, j = len -1; i < j; ++i, --j) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}


int main(int argc, char const *argv[]) {
        int sfd, len;
        char *str_buf;
        struct sockaddr_in servaddr, clientaddr;
```

```c
sfd = socket(AF_INET, SOCK_DGRAM,IPPROTO_UDP);

if (sfd == ERROR) {

        perror("Could not open a socket");

        return 1;

}

memset((char *) &servaddr, 0, sizeof(servaddr));

servaddr.sin_family=AF_INET;

servaddr.sin_addr.s_addr=htonl(INADDR_ANY);

servaddr.sin_port=htons(S_PORT);


memset((char *) &clientaddr, 0, sizeof(clientaddr));

clientaddr.sin_family=AF_INET;

clientaddr.sin_addr.s_addr=inet_addr(IP_STR);

clientaddr.sin_port=htons(C_PORT);


if((bind(sfd,(struct sockaddr *)&servaddr,sizeof(servaddr)))!=0) {

        perror("Could not bind socket");

        return 2;

}


printf("Server is running on %s:%d\n", IP_STR, S_PORT);

while(1) {

        recvfrom(sfd, &len, sizeof(len), 0, (struct sockaddr *)&clientaddr, (socklen_t *)&clientaddr);

        str_buf = (char *) malloc(len*sizeof(char));

        recvfrom(sfd, str_buf, len, 0, (struct sockaddr *)&clientaddr, (socklen_t *)&clientaddr);

        printf("Client at %s:%d said: %s\t", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), str_buf);

        strrev(str_buf,len);

        sendto(sfd, &len, sizeof(len), 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
```
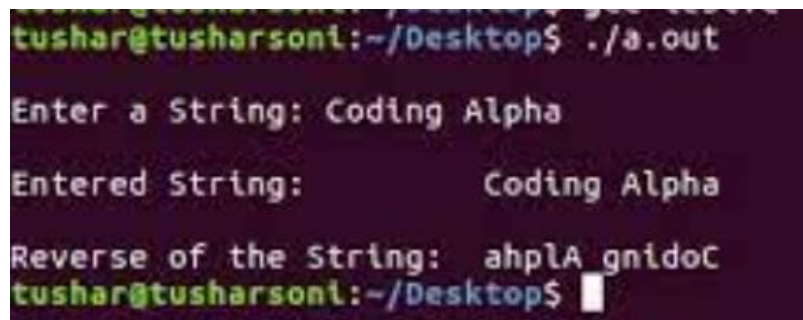
```
sendto(sfd, str_buf, len, 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));

printf("The reverse is: %s\n", str_buf);

free(str_buf);

}

return 0;

}
```

**OUTPUT:**

**9.2) Design UDP Client server to transfer a file.**

**AIM:** To design UDP Client server to transfer a file.

## DESCRIPTION:

The UDP or User Datagram Protocol, a communication protocol used for transferring data across the network. It is an unreliable and connectionless communication protocol as it does not establish a proper connection between the client and the server. It is used for time-sensitive applications like gaming, playing videos, or Domain Name System (DNS) lookups. UDP is a faster communication protocol as compared to the TCP

Some of the features of UDP are:

- It's a connectionless communication protocol.
- It is much faster in comparison with TCP.

In UDP, the client does not form a connection with the server like in TCP and instead sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of the sender which the server uses to send data to the correct client.
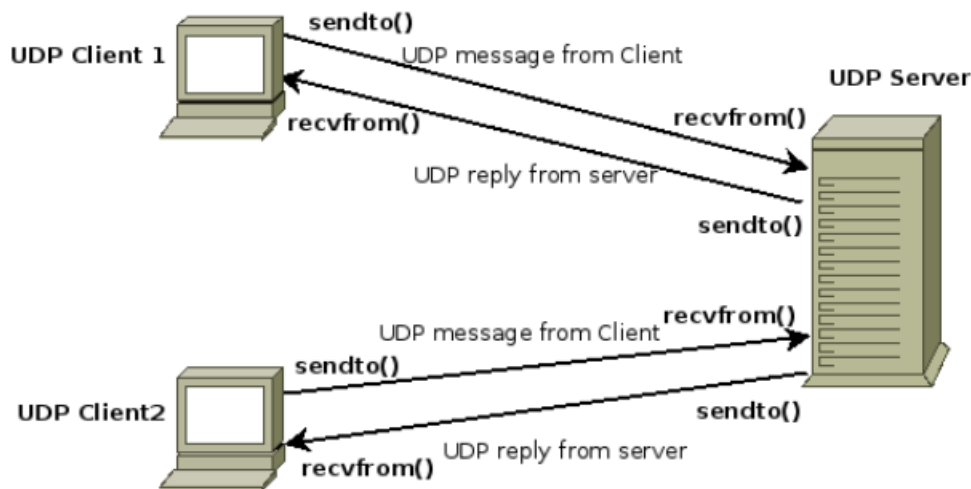


**Algorithm :**
1. The server starts and waits for filename.
2. The client sends a filename.
3. The server receives filename.
   If file is present,
   server starts reading file
   and continues to send a buffer filled with
   file contents encrypted until file-end is reached.
4. End is marked by EOF.
5. File is received as buffers until EOF is
   received. Then it is decrypted.
6. If Not present, a file not found is sent.

**PROGRAM:**

**Server Implementation:**

```c
#include <arpa/inet.h>

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <unistd.h>


#define IP_PROTOCOL 0

#define PORT_NO 15050

#define NET_BUF_SIZE 32

#define cipherKey 'S'

#define sendrecvflag 0

#define nofile "File Not Found!"


// function to clear buffer
void clearBuf(char* b)
{
        int i;
        for (i = 0; i < NET_BUF_SIZE; i++)
                b[i] = '\0';
}


// function to encrypt
```

```c
char Cipher(char ch)

{

        return ch ^ cipherKey;

}


// function sending file
int sendFile(FILE* fp, char* buf, int s)

{

        int i, len;

        if (fp == NULL) {

                strcpy(buf, nofile);

                len = strlen(nofile);

                buf[len] = EOF;

                for (i = 0; i <= len; i++)

                        buf[i] = Cipher(buf[i]);

                return 1;

        }


        char ch, ch2;

        for (i = 0; i < s; i++) {

                ch = fgetc(fp);

                ch2 = Cipher(ch);

                buf[i] = ch2;

                if (ch == EOF)

                        return 1;

        }

        return 0;

}
```

```c
// driver code
int main()
{
        int sockfd, nBytes;
        struct sockaddr_in addr_con;
        int addrlen = sizeof(addr_con);
        addr_con.sin_family = AF_INET;
        addr_con.sin_port = htons(PORT_NO);
        addr_con.sin_addr.s_addr = INADDR_ANY;
        char net_buf[NET_BUF_SIZE];
        FILE* fp;

        // socket()
        sockfd = socket(AF_INET, SOCK_DGRAM, IP_PROTOCOL);

        if (sockfd < 0)
                printf("\nfile descriptor not received!!\n");
        else
                printf("\nfile descriptor %d received\n", sockfd);

        // bind()
        if (bind(sockfd, (struct sockaddr*)&addr_con, sizeof(addr_con)) == 0)
                printf("\nSuccessfully binded!\n");
        else
                printf("\nBinding Failed!\n");

        while (1) {
```

```c
printf("\nWaiting for file name...\n");


// receive file name
clearBuf(net_buf);


nBytes = recvfrom(sockfd, net_buf,

                            NET_BUF_SIZE, sendrecvflag,

                            (struct sockaddr*)&addr_con, &addrlen);


fp = fopen(net_buf, "r");
printf("\nFile Name Received: %s\n", net_buf);
if (fp == NULL)
        printf("\nFile open failed!\n");
else
        printf("\nFile Successfully opened!\n");


while (1) {


        // process
        if (sendFile(fp, net_buf, NET_BUF_SIZE)) {
                sendto(sockfd, net_buf, NET_BUF_SIZE,

                        sendrecvflag,

                        (struct sockaddr*)&addr_con, addrlen);

                break;

        }


        // send
        sendto(sockfd, net_buf, NET_BUF_SIZE,
```

```
                          sendrecvflag,

                          (struct sockaddr*)&addr_con, addrlen);

                   clearBuf(net_buf);

            }

            if (fp != NULL)

                   fclose(fp);

      }

      return 0;

}
```

## Client Implementation:

```
// client code for UDP socket programming

#include <arpa/inet.h>

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <unistd.h>


#define IP_PROTOCOL 0

#define IP_ADDRESS "127.0.0.1" // localhost

#define PORT_NO 15050

#define NET_BUF_SIZE 32

#define cipherKey 'S'

#define sendrecvflag 0


// function to clear buffer
```

```c
void clearBuf(char* b)
{
        int i;
        for (i = 0; i < NET_BUF_SIZE; i++)
                b[i] = '\0';
}
char Cipher(char ch)
{
        return ch ^ cipherKey;
}
int recvFile(char* buf, int s)
{
        int i;
        char ch;
        for (i = 0; i < s; i++) {
                ch = buf[i];
                ch = Cipher(ch);
                if (ch == EOF)
                        return 1;
                else
                        printf("%c", ch);
        }
        return 0;
}
int main()
{
        int sockfd, nBytes;
        struct sockaddr_in addr_con;
```

```c
int addrlen = sizeof(addr_con);

addr_con.sin_family = AF_INET;

addr_con.sin_port = htons(PORT_NO);

addr_con.sin_addr.s_addr = inet_addr(IP_ADDRESS);

char net_buf[NET_BUF_SIZE];

FILE* fp;

sockfd = socket(AF_INET, SOCK_DGRAM,

                              IP_PROTOCOL);

if (sockfd < 0)

        printf("\nfile descriptor not received!!\n");

else

        printf("\nfile descriptor %d received\n", sockfd);

while (1) {

        printf("\nPlease enter file name to receive:\n");

scanf("%s", net_buf);

sendto(sockfd, net_buf, NET_BUF_SIZE, sendrecvflag, (struct sockaddr*)&addr_con, addrlen);

printf("\n---------Data Received---------\n");

while (1) {

      clearBuf(net_buf);

      nBytes = recvfrom(sockfd, net_buf, NET_BUF_SIZE,sendrecvflag, (struct sockaddr*)

      &addr_con,&addrlen);

      if (recvFile(net_buf, NET_BUF_SIZE)) {

              break;

      }

      }

      printf("\n------------------------------\n");

}

return 0;
```

}

**OUTPUT:**

**Server Side:**

```
Socket file descriptor 3 received
Successfully binded!
Waiting for file name...
File Name Received: dm.txt
File Successfully opened!
Waiting for file name...
File Name Received: /home/dmayank/Documents/dm.txt
File Successfully opened!
```

**Client side:**

```
Socket file descriptor 3 received
Please enter file name to receive:
dm.txt

---------Data Received---------
30
------------------------------

Please enter file name to receive:
/home/dmayank/Documents/dm.txt

---------Data Received---------
30
------------------------------
```

**1.What is the UDP?**

**2. What are the features of UDP?**

**3. What is the datagram?**

**4. Explain the process how the datagram flow does in UDP?**

**5. Explain the advantages of UDP?**

**6. What are the drawbacks of the UDP?**

**7. What is the operation is used in encryption in UDP?**

**Week - 10 IPC**

**Implement the following forms of IPC. a) Pipes b) FIFO**

**AIM:** To implement the following forms of IPC a. Pipes b. FIFO.

**DESCRIPTION:**

**1.Pipes**

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.
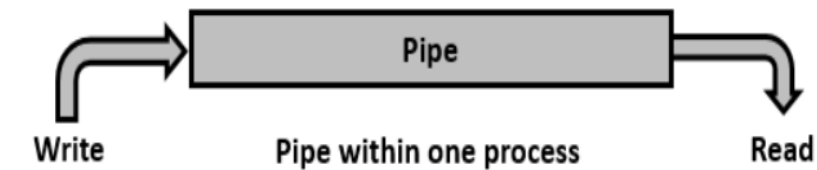


This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

**Algorithm**

**Step 1** – Create a pipe.

**Step 2** – Create a child process.

**Step 3** – Parent process writes to the pipe.

**Step 4** – Child process retrieves the message from the pipe and writes it to the standard output.

**Step 5** – Repeat step 3 and step 4 once again.

**PROGRAM:**

```c
#include<stdio.h>
#include<unistd.h>
int main() {
    int pipefds[2];
    int returnstatus;
    int pid;
    char writemessages[2][20]={"Hi", "Hello"};
    char readmessage[20];
    returnstatus = pipe(pipefds);
    if (returnstatus == -1) {
        printf("Unable to create pipe\n");
        return 1;
    }
    pid = fork();
    if (pid == 0) {
        read(pipefds[0], readmessage, sizeof(readmessage));
        printf("Child Process - Reading from pipe – Message 1 is %s\n", readmessage);
        read(pipefds[0], readmessage, sizeof(readmessage));
        printf("Child Process - Reading from pipe – Message 2 is %s\n", readmessage);
    } else { //Parent process
        printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);
        write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
        printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);
        write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
    }
    return 0;
}
```

**OUTPUT:**

```
Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe – Message 1 is Hi
Child Process - Reading from pipe – Message 2 is Hello
```

**DESCRIPTION:**

**2.FIFO**

Pipes were meant for communication between related processes. Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal? The answer is No. Then how can we achieve unrelated processes communication, the simple answer is Named Pipes. Even though this works for related processes, it gives no meaning to use the named pipes for related process communication.

We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes. Another name for named pipe is **FIFO (First-In-First-Out)**. Let us see the system call (mknod()) to create a named pipe, which is a kind of a special file.

This system call would create a special file or file system node such as ordinary file, device file, or FIFO. The arguments to the system call are pathname, mode and dev. The pathname along with the attributes of mode and device information. The pathname is relative, if the directory is not specified it would be created in the current directory. The mode specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables. The dev field is to specify device information such as major and minor device numbers.

| File Type | Description | File Type | Description |
|---|---|---|---|
| S_IFBLK | block special | S_IFREG | Regular file |
| S_IFCHR | character special | S_IFDIR | Directory |
| S_IFIFO | FIFO special | S_IFLNK | Symbolic Link |

File mode can also be represented in octal notation such as 0XYZ, where X represents owner, Y represents group, and Z represents others. The value of X, Y or Z can range from 0 to 7. The values for read, write and execute are 4, 2, 1 respectively. If needed in combination of read, write and execute, then add the values accordingly.

Say, if we mention, 0640, then this means read and write (4 + 2 = 6) for owner, read (4) for group and no permissions (0) for others.

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

**PROGRAM:**

```c
#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FIFO_FILE "MYFIFO"

int main() {

  int fd;

  int end_process;

  int stringlen;

  char readbuf[80];

  char end_str[5];

  printf("FIFO_CLIENT: Send messages, infinitely, to end enter \"end\"\n");

  fd = open(FIFO_FILE, O_CREAT|O_WRONLY);

  strcpy(end_str, "end");


  while (1) {

    printf("Enter string: ");

    fgets(readbuf, sizeof(readbuf), stdin);

    stringlen = strlen(readbuf);

    readbuf[stringlen - 1] = '\0';

    end_process = strcmp(readbuf, end_str);


    //printf("end_process is %d\n", end_process);

    if (end_process != 0) {

      write(fd, readbuf, strlen(readbuf));
```

```c
        printf("Sent string: \"%s\" and string length is %d\n", readbuf, (int)strlen(readbuf));

    } else {

        write(fd, readbuf, strlen(readbuf));

        printf("Sent string: \"%s\" and string length is %d\n", readbuf, (int)strlen(readbuf));

        close(fd);

        break;

    }

  }
  return 0;

}
```

**OUTPUT:**

```
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: this is string 1
Sent string: "this is string 1" and string length is 16
Enter string: fifo test
Sent string: "fifo test" and string length is 9
Enter string: fifo client and server
Sent string: "fifo client and server" and string length is 22
Enter string: end
Sent string: "end" and string length is 3
```

1.What is IPC stands for? Explain IPC.

2. What are the different forms of IPC?

3. What is use of the IPC?

4. Explain about the pipes?

5. Explain the 2-way communication in the pipe?

6. What is FIFO in IPC?

7. What is the functionality of the FIFO?

**Week - 11 IPC Continued**

**Implement file transfer using Message Queue form of IPC**

**AIM:** To implement file transfer using Message Queue form of IPC

**DESCRIPTION:**

once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.
If we want to communicate with small message formats. Shared memory data need to be protected with synchronization when multiple processes communicating at the same time. Frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality.

What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to **implement with message queues.** If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simplier to implement with message queues. To simplify the given message type as 1, 10, 20, it can be either 0 or +ve or –ve as discussed below.

Of course, the order of message queue is FIFO (First In First Out). The first message inserted in the queue is the first one to be retrieved. Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.
Communication using message queues can happen in the following ways −

- Writing into the shared memory by one process and reading from the shared memory by another process. As we are aware, reading can be done with multiple processes as well.
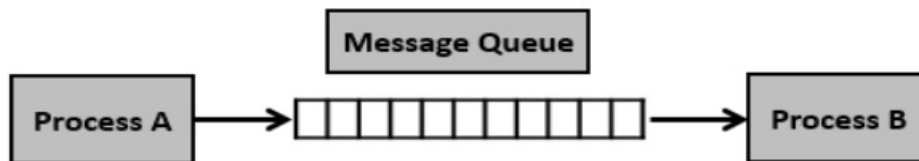


- Writing into the shared memory by one process with different data packets and reading from it by multiple processes, i.e., as per message type.
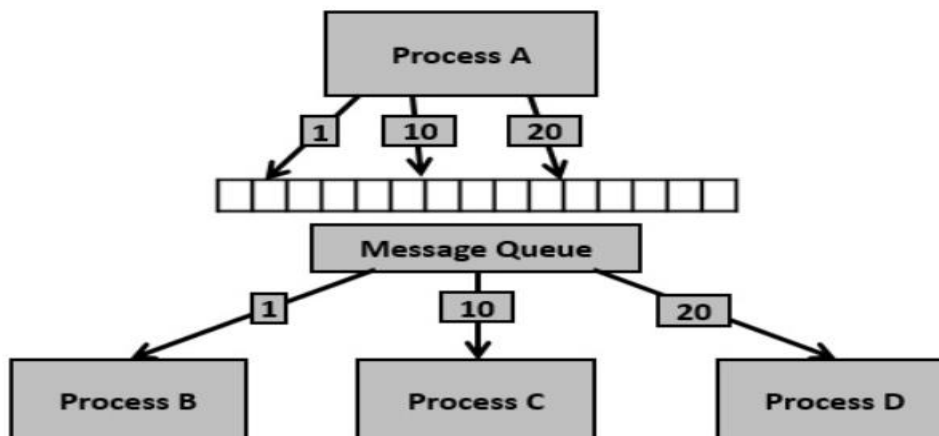
**PROGRAM:**

```c
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

// structure for message queue

struct msg_buffer {

  long msg_type;

  char msg[100];

} message;

main() {

  key_t my_key;

  int msg_id;

  my_key = ftok("progfile", 65); //create unique key

  msg_id = msgget(my_key, 0666 | IPC_CREAT); //create message queue and return id

  message.msg_type = 1;

  printf("Write Message : ");

  fgets(message.msg, 100, stdin);

  msgsnd(msg_id, &message, sizeof(message), 0); //send message

  printf("Sent message is : %s \n", message.msg);

}
```
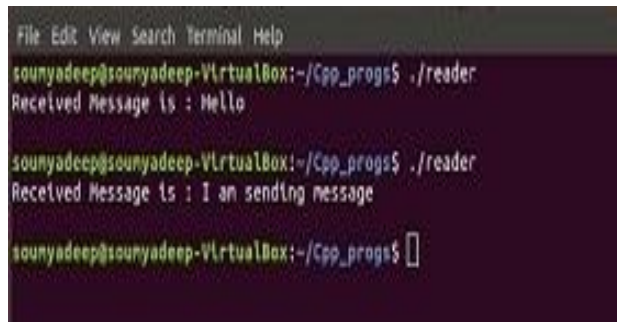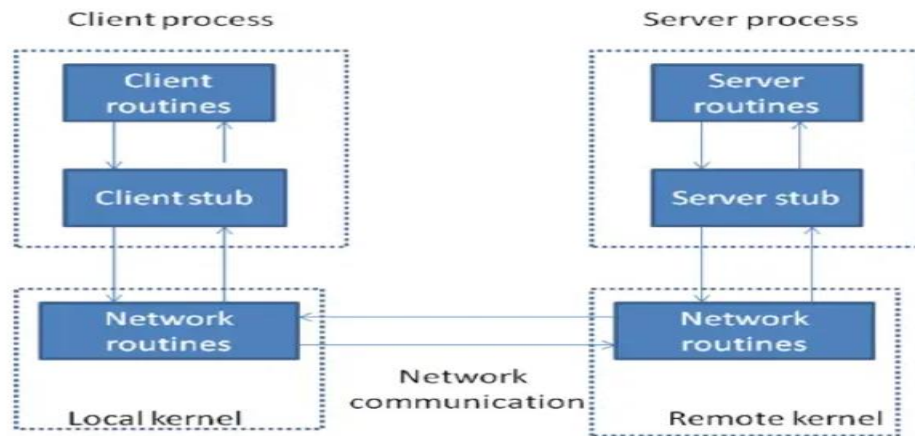
**OUTPUT:**

1.What is IPC stands for? Explain IPC.

2. What are the different forms of IPC?

3. What is use of the IPC?

4. Explain about the message queues?

5. Explain why we need shared memory?

6. What is difference between shared memory and message queue?

7. What is the functionality of the message queues?

**Week - 12 RPC**

**Design a RPC application to add and subtract a given pair of integers**

**AIM:** To design a RPC application to add and subtract a given pair of integers

**DECRIPTION:**

Client process    Server process
Client routines    Server routines
Client stub    Server stub
Network routines    Network routines
Local kernel    Network communication    Remote kernel

**Remote Procedure Call model**

The steps in the Figure Remote Procedure Call ( RPC ) Model are.

- The client calls a local procedure, called the clients stub. It appears to the client that the client stub is the actual server procedure that it wants to call. the purpose of the stub is to package up the arguments to the remote procedure, possibly put them into some standard format and then build one or more network messages. The packaging of the clients arguments into a network message is termed marshaling.

- These network messages are sent to the remote system by the client stub. This requires a system call into the kernel. The network messages are transferred to the remote system. Either a connection-oriented or a connectionless protocol is used.

- A Server stub procedure is waiting on the remote system for the client's request. It unmartial the arguments from the network messages and possibly converts them.

- The server stub executes a local procedure call to invoke the actual server function, passing it the arguments that it received in the network messages from the clients tub.

- When the server procedure is finished, it returns to the server stub, returning whatever its return values are. The server stub converts the return values, if necessary and marshals them into one or more network messages to send back to the client stub.

- To message get transferred back across the network to client stub. The client stub reads the network message from the local kernel.

- After possibly converting the return values the client stub finally returns to the client functions this appears to be a normal procedure returns to the client

**PROGRAM:**

```c
#include "rpctime.h"

#include <stdio.h>

#include <stdlib.h>

#include <rpc/pmap_clnt.h>

#include <string.h>

#include <memory.h>

#include <sys/socket.h>

#include <netinet/in.h>

#ifndef SIG_PF

#define SIG_PF void(*)(int)

#endif

static void

rpctime_1(struct svc_req *rqstp, register SVCXPRT *transp)

{

union {int fill;

} argument;

char *result;

xdrproc_t _xdr_argument, _xdr_result;

char *(*local)(char *, struct svc_req *);

switch (rqstp->rq_proc) {

case NULLPROC:(void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);

return;

case GETTIME:

_xdr_argument = (xdrproc_t) xdr_void;

_xdr_result = (xdrproc_t) xdr_long;

local = (char *(*)(char *, struct svc_req *)) gettime_1_svc;

break;
```

```c
default:

svcerr_noproc (transp);

return;

}

memset ((char *)&argument, 0, sizeof (argument));

if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {

svcerr_decode (transp);

return;

}

result = (*local)((char *)&argument, rqstp);

if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {

svcerr_systemerr (transp);

}

if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {

fprintf (stderr, "%s", "unable to free arguments");

exit (1);

}

return;

}

intmain (int argc, char **argv){

register SVCXPRT *transp;

pmap_unset (RPCTIME, RPCTIMEVERSION);

transp = svcudp_create(RPC_ANYSOCK);

if (transp == NULL) {

fprintf (stderr, "%s", "cannot create udp service.");

exit(1);

}

if (!svc_register(transp, RPCTIME, RPCTIMEVERSION, rpctime_1, IPPROTO_UDP)) {
```

```
fprintf (stderr, "%s", "unable to register (RPCTIME, RPCTIMEVERSION,udp).");

exit(1);

}

transp = svctcp_create(RPC_ANYSOCK, 0, 0);if (transp == NULL) {

fprintf (stderr, "%s", "cannot create tcp service.");

exit(1);

}

if (!svc_register(transp, RPCTIME, RPCTIMEVERSION, rpctime_1, IPPROTO_TCP)) {

fprintf (stderr, "%s", "unable to register (RPCTIME, RPCTIMEVERSION, tcp).");

exit(1);

}

svc_run ();

fprintf (stderr, "%s", "svc_run returned");

exit (1);

}
```

## Client Side:

```
#include "rpctime.h"

voidrpctime_1(char *host){

CLIENT *clnt;47long *result_1;

char *gettime_1_arg;

#ifndef DEBUGclnt = clnt_create (host, RPCTIME, RPCTIMEVERSION, "udp");

if (clnt == NULL) {

clnt_pcreateerror (host);

exit (1);

}

#endif /* DEBUG */

result_1 = gettime_1((void*)&gettime_1_arg, clnt);

if (result_1 == (long *) NULL) {
```

```c
clnt_perror (clnt, "call failed");

}

Else

printf("%d |%s", *result_1, ctime(result_1));

#ifndef DEBUGclnt_destroy (clnt);

#endif /* DEBUG */}

intmain (int argc, char *argv[]){

char *host;

if (argc < 2) {

printf ("usage: %s server_host\n", argv[0]);

exit (1);

}

host = argv[1];

rpctime_1 (host);

exit (0);

}

rpctime_cntl.c

#include <memory.h> /* for memset */

#include "rpctime.h"/* Default timeout can be changed using clnt_control() */

static struct timeval TIMEOUT = { 25, 0 };

long *

gettime_1(void *argp, CLIENT *clnt){

static long clnt_res;memset((char *)&clnt_res, 0, sizeof(clnt_res));

if (clnt_call (clnt, GETTIME,(xdrproc_t) xdr_void, (caddr_t) argp,(xdrproc_t) xdr_long, (caddr_t) &clnt_res,TIMEOUT) != RPC_SUCCESS) {

return (NULL);

}

return (&clnt_res);

}
```

**OUTPUT:**

```
Step 1:     $rpcgen –C –a simp.x
//This creates simp.h, simp_clnt.c, simp_svc.c simp_xdr.c files in the folder //
Step 2: $cc –o client simp_client.c simp_clnt.c simp_xdr.c –lrpcsvc –lnsl
Step 3: $ cc –o server simp_server.c simp_svc.c simp_xdr.c –lrpcsvc –lnsl
Step 4: $ ./server &
$./client 10.0.0.1  10  5
Add = 10 + 5 = 15
Sub = 10 – 5 = 5
```

<u>VIVA VOICE:</u>

**1.What is RPC stands for? Explain RPC.**

**2. What are the applications of RPC?**

**3. What is use of the RPC?**

**4. Explain about the functionality of RPC?**

**5. Explain why we need RPC?**