Socket Programming for practices:

Q.Write a program in c++ Socket Programming to Perform message passing from Client to the Server.

```
//Client application in the socket programming
#include <cstring>
#include <iostream>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

int main()
{
    // creating socket
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);

    // specifying address
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8080);
    serverAddress.sin_addr.s_addr = INADDR_ANY;

    // sending connection request
    connect(clientSocket, (struct sockaddr*)&serverAddress,sizeof(serverAddress));

    // sending data
    const char* message = "Hello, server!";
    send(clientSocket, message, strlen(message), 0);

    // closing socket
    close(clientSocket);

    return 0;
}

//server application in socket programming
#include <cstring>
#include <iostream>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
#include <unistd.h>

using namespace std;

int main()
{
    // creating socket
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // specifying the address
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8080);
    serverAddress.sin_addr.s_addr = INADDR_ANY;

    // binding socket.
    bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress));

    // listening to the assigned socket
    listen(serverSocket, 5);

    // accepting connection request
    int clientSocket
        = accept(serverSocket, nullptr, nullptr);

    // recieving data
    char buffer[1024] = { 0 };
    recv(clientSocket, buffer, sizeof(buffer), 0);
    cout << "Message from client: " << buffer
             << endl;

    // closing the socket.
    close(serverSocket);

    return 0;
}
```

Q.Write a program in c++ Socket Programming to Perform asynchronous connection.

```
// async_client.cpp
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};

    // Create a socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return -1;
    }

    // Set up the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 address from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection Failed" << std::endl;
        return -1;
    }

    // Send a message to the server
```

```
    const char *message = "Hello from asynchronous client";
    send(sock, message, strlen(message), 0);
    std::cout << "Message sent to server" << std::endl;

    // Read the server's response
    read(sock, buffer, BUFFER_SIZE);
    std::cout << "Received from server: " << buffer << std::endl;

    // Close the socket
    close(sock);

    return 0;
}

// async_server.cpp
#include <iostream>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <cstring>
#include <vector>

#define PORT 8080
#define MAX_EVENTS 10
#define BUFFER_SIZE 1024

// Function to make socket non-blocking
int make_socket_non_blocking(int sockfd) {
    int flags = fcntl(sockfd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl");
        return -1;
    }
    if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl");
        return -1;
    }
    return 0;
```

```
}

int main() {
    int server_fd, epoll_fd;
    struct sockaddr_in address;
    char buffer[BUFFER_SIZE];

    // Create a socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        std::cerr << "Socket creation failed" << std::endl;
        return -1;
    }

    // Make server socket non-blocking
    make_socket_non_blocking(server_fd);

    // Set up the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind the socket to the address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        std::cerr << "Bind failed" << std::endl;
        return -1;
    }

    // Listen for incoming connections
    if (listen(server_fd, SOMAXCONN) < 0) {
        std::cerr << "Listen failed" << std::endl;
        return -1;
    }

    // Create an epoll instance
  //  epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        return -1;
    }
```

```
// Add the server socket to the epoll instance
struct epoll_event event;
event.data.fd = server_fd;
event.events = EPOLLIN;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event) == -1) {
    perror("epoll_ctl: server_fd");
    return -1;
}

std::vector<struct epoll_event> events(MAX_EVENTS);

std::cout << "Server is running asynchronously on port " << PORT << std::endl;

// Event loop
while (true) {
    int n = epoll_wait(epoll_fd, events.data(), MAX_EVENTS, -1);
    for (int i = 0; i < n; i++) {
        if (events[i].data.fd == server_fd) {
            // Handle new incoming connection
            int client_fd;
            struct sockaddr_in client_address;
            socklen_t client_len = sizeof(client_address);
            client_fd = accept(server_fd, (struct sockaddr *)&client_address, &client_len);
            if (client_fd == -1) {
                perror("accept");
                continue;
            }

            // Make the client socket non-blocking
            make_socket_non_blocking(client_fd);

            // Add new client socket to epoll
            event.data.fd = client_fd;
            event.events = EPOLLIN | EPOLLET;
            epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event);
            std::cout << "New client connected" << std::endl;

        } else {
            // Handle data from an existing connection
```

```
            int client_fd = events[i].data.fd;
            memset(buffer, 0, BUFFER_SIZE);
            int count = read(client_fd, buffer, BUFFER_SIZE);
            if (count == -1) {
                perror("read");
                close(client_fd);
            } else if (count == 0) {
                // Client disconnected
                std::cout << "Client disconnected" << std::endl;
                close(client_fd);
            } else {
                std::cout << "Received: " << buffer << std::endl;
                // Send response
                const char *response = "Hello from asynchronous server";
                send(client_fd, response, strlen(response), 0);
            }
        }
    }
}

close(server_fd);
close(epoll_fd);
return 0;
}
```

Q. Write a program in c++ Socket Programming to Perform Synchronous connection.

```
// simple_client.cpp
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#define PORT 8080

using namespace std;

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[1024] = {0};
```

```
    // Create socket (IPv4, TCP)
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return -1;
    }

    // Set up the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 address from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address or address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection to server failed" << std::endl;
        return -1;
    }

    // Send message to server
    const char *message = "Hello from client";
    send(sock, message, strlen(message), 0);
    cout << "Message sent to server" << std::endl;

    // Read response from server
    read(sock, buffer, 1024);
    cout << "Received from server: " << buffer << std::endl;

    // Close the socket
    close(sock);

    return 0;
}

// simple_server.cpp
```

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

#define PORT 8080

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};

    // Create a socket (IPv4, TCP)
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        std::cerr << "Socket creation failed" << std::endl;
        return -1;
    }

    // Set up the server address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; // Bind to all available interfaces
    address.sin_port = htons(PORT);

    // Bind the socket to the specified address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        std::cerr << "Bind failed" << std::endl;
        return -1;
    }

    // Listen for incoming connections (up to 3 pending connections)
    if (listen(server_fd, 3) < 0) {
        std::cerr << "Listen failed" << std::endl;
        return -1;
    }

    std::cout << "Server is waiting for a connection..." << std::endl;
```

```
    // Accept an incoming connection
    new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen);
    if (new_socket < 0) {
        std::cerr << "Connection acceptance failed" << std::endl;
        return -1;
    }

    // Read message from client
    read(new_socket, buffer, 1024);
    std::cout << "Received from client: " << buffer << std::endl;

    // Send response to client
    const char *response = "Hello from server";
    send(new_socket, response, strlen(response), 0);
    std::cout << "Response sent to client" << std::endl;

    // Close the sockets
    close(new_socket);
    close(server_fd);

    return 0;
}
```

To Execute:
g++ client.cpp -o client
./client
g++ server.cpp -o server
./server

Q. Write a program to encrypt input string by using SecretKey of the following algorithms, and then decrypt the encrypted string and compare the decrypted string with the input string. Use the following algorithms for encryption and decryption:
a. DES
b. BlowFish
c. IDEA
d. Triple DES

```
#include <iostream>
#include <openssl/evp.h>
#include <openssl/rand.h>
```

```
#include <openssl/hmac.h>
using namespace std;

void generateSymmetricKey(int keyLength, const string& algorithmName) {
    unsigned char key[keyLength];

    // Generate a random key of specified length
    if (!RAND_bytes(key, keyLength)) {
        cerr << "Error generating key for " << algorithmName << endl;
        return;
    }
    cout << algorithmName << " Key (" << keyLength * 8 << " bits): ";
    for (int i = 0; i < keyLength; i++) {
        printf("%02x", key[i]);
    }
    cout << endl;
}

void generateHmacKey(int keyLength, const string& algorithmName) {
    unsigned char key[keyLength];

    // Generate a random key for HMAC
    if (!RAND(keyLength)) {
        cerr << "Error generating HMAC key for " << algorithmName << endl;
        return;
    }

    cout << algorithmName << " HMAC Key (" << keyLength * 8 << " bits): ";
    for (int i = 0; i < keyLength; i++) {
        printf("%02x", key[i]);
    }
    cout << endl;
}

int main() {
    // DES uses a 56-bit key (7 bytes)
    generateSymmetricKey(8, "DES");  // DES key length is technically 56 bits, but 8 bytes are
used

    // TripleDES (3DES) uses a 168-bit key (21 bytes)
```

```
    generateSymmetricKey(24, "TripleDES");

    // AES supports 128-bit, 192-bit, or 256-bit keys
    generateSymmetricKey(16, "AES-128");
    generateSymmetricKey(24, "AES-192");
    generateSymmetricKey(32, "AES-256");

    // Blowfish supports key sizes from 32 bits up to 448 bits
    generateSymmetricKey(16, "Blowfish");  // Blowfish with 128-bit key

    // HMAC using MD5 and SHA1
    generateHmacKey(16, "HmacMD5");   // 128-bit HMAC key for MD5
    generateHmacKey(20, "HmacSHA1");  // 160-bit HMAC key for SHA1

    return 0;
}
```

Q. Write a program to generate Symmetric Keys for the following Cipher algorithms DES, AES, Blowfish, TripleDES, HmacMD5 and HmacSHA1.

```
#include <iostream>
#include <cstring>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/err.h>
#include <openssl/provider.h>

void handleErrors() {
    ERR_print_errors_fp(stderr);
    abort();
}

void encryptDecrypt(const std::string& input, const EVP_CIPHER* cipherType, const
std::string& algorithmName) {
    // Key and IV buffers
    unsigned char key[EVP_MAX_KEY_LENGTH];
    unsigned char iv[EVP_MAX_IV_LENGTH];

    // Generate random key and IV
```

```
    if (!RAND_bytes(key, EVP_CIPHER_key_length(cipherType)) || !RAND_bytes(iv,
EVP_CIPHER_iv_length(cipherType))) {
        std::cerr << "Error generating key/IV for " << algorithmName << std::endl;
        handleErrors();
    }

    // Create a context for encryption
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        handleErrors();
    }

    // Encrypt the input string
    int outlen, cipherTextLen;
    unsigned char cipherText[1024];

    if (!EVP_EncryptInit_ex(ctx, cipherType, NULL, key, iv)) {
        handleErrors();
    }

    if (!EVP_EncryptUpdate(ctx, cipherText, &outlen, (unsigned char*)input.c_str(), input.size()))
{
        handleErrors();
    }
    cipherTextLen = outlen;

    if (!EVP_EncryptFinal_ex(ctx, cipherText + outlen, &outlen)) {
        handleErrors();
    }
    cipherTextLen += outlen;

    EVP_CIPHER_CTX_free(ctx);

    // Now decrypt the cipherText
    unsigned char decryptedText[1024];
    int decryptedTextLen;

    ctx = EVP_CIPHER_CTX_new();
    if (!EVP_DecryptInit_ex(ctx, cipherType, NULL, key, iv)) {
        handleErrors();
```

```
    }

    if (!EVP_DecryptUpdate(ctx, decryptedText, &outlen, cipherText, cipherTextLen)) {
        handleErrors();
    }
    decryptedTextLen = outlen;

    if (!EVP_DecryptFinal_ex(ctx, decryptedText + outlen, &outlen)) {
        handleErrors();
    }
    decryptedTextLen += outlen;

    EVP_CIPHER_CTX_free(ctx);

    // Null-terminate the decrypted string
    decryptedText[decryptedTextLen] = '\0';

    // Print the results
    std::cout << "Algorithm: " << algorithmName << std::endl;
    std::cout << "Input: " << input << std::endl;
    std::cout << "Decrypted Text: " << decryptedText << std::endl;

    if (input == std::string((char*)decryptedText)) {
        std::cout << "Decryption successful: Input matches Decrypted Text!" << std::endl;
    } else {
        std::cerr << "Decryption failed: Input does not match Decrypted Text!" << std::endl;
    }

    std::cout << "----------------------------------------" << std::endl;
}

int main() {
    // Initialize OpenSSL algorithms and load providers
    OpenSSL_add_all_algorithms();
    ERR_load_crypto_strings();

    // Load both legacy and default providers
    OSSL_PROVIDER* legacy = OSSL_PROVIDER_load(NULL, "legacy");
    OSSL_PROVIDER* defaultProvider = OSSL_PROVIDER_load(NULL, "default");
    if (!legacy || !defaultProvider) {
```

```
        std::cerr << "Failed to load providers" << std::endl;
        return 1;
    }

    std::string inputText = "This is a test message to be encrypted and decrypted";

    // Encrypt and decrypt using different algorithms
    encryptDecrypt(inputText, EVP_des_cbc(), "DES");
    encryptDecrypt(inputText, EVP_bf_cbc(), "Blowfish");
    encryptDecrypt(inputText, EVP_aes_256_cbc(), "AES-256");
    encryptDecrypt(inputText, EVP_des_ede3_cbc(), "Triple DES (3DES)");

    // Unload the providers
    OSSL_PROVIDER_unload(legacy);
    OSSL_PROVIDER_unload(defaultProvider);

    // Cleanup OpenSSL
    EVP_cleanup();
    ERR_free_strings();

    return 0;
}
```

To Execute openssl program:
g++ -o encrypt encrypt.cpp -lssl -lcrypto
./encrypt