client.cpp

```
#include "headers.hpp"
#include "client.hpp"
#include "tools.hpp"
#include <iomanip>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <fstream>

Client::Client()
{
    p = bigint("170141183460469231731687303715884105757");
    g = bigint("340282366920938463463374607431768211507");
    unsigned char secret_buffer[16];
    RAND_bytes(secret_buffer, 16);

    secret = 0;
    for (int i = 0; i < 16; i++)
    {
        secret <<= 8;
        secret |= secret_buffer[i];
    }

}


bool Client::connectToServer(string& error)
{
    // parse server address file

    int server_tcp_port = -1, server_udp_port = -1;
    try
    {
        std::ifstream file("server/server_address.txt");
        // comment lines

        string line;
        while (std::getline(file, line))
        {
            if (line[0] != '#')
                break;
        }
        // ip address
        ip = line;

        // tcp & udp ports
        std::getline(file, line);
        server_tcp_port = std::stoi(line);
        std::getline(file, line);
        server_udp_port = std::stoi(line);

        cout << "ports: " << server_tcp_port << ":" << server_udp_port
        << "\n";
    }
```

```
catch (const std::exception&)
{
    error = "Invalid File at server/server_address.txt";
    return false;
}


this->ip = ip;
this->udp_sender_port = server_udp_port;
this->udp_listener_port = server_udp_port;

sf::Socket::Status connection_status = tcp_socket.connect(ip,
server_tcp_port, sf::milliseconds(50));
if (connection_status != sf::Socket::Done)
{
    error = "Failed To Connect To Server";
    return false;
}

cout << "address: " << tcp_socket.getRemoteAddress() << "\n";

// Diffie Hellman
// X1
bigint x1 = powm(g, secret, p);

string message = "X" + x1.str() + "X";

if (!sendTCP(tcp_socket, message, error))
    return false;

// X2
void* buffer;
int buffer_size;
if (!recvTCP(tcp_socket, buffer, buffer_size))
{
    error = "Failed To Receive Message From Server";
    return false;
}

string x2_str((char*)buffer, buffer_size);
free(buffer);

if (x2_str == "ERROR" || x2_str[0] != 'X' || x2_str[x2_str.size() -
1] != 'X')
{
    error = "Invalid X2 Received From Server: " + x2_str;
    return false;
}

bigint x2(x2_str.substr(1, x2_str.size() - 2));


// Key
```

```
    bigint key = powm(x2, secret, p);
    bigintToBytes(key, key_bytes);


    if (udp_socket.bind(sf::Socket::AnyPort) != sf::Socket::Done)
    {
        error = "Error binding UDP socket";
        return false;
    }

    udp_sender_address = udp_listener_address =
    tcp_socket.getRemoteAddress();
    cout << "UDP Socket Local Port: " << udp_socket.getLocalPort() <<
    "\n";

    udp_socket.setBlocking(false);

    connected = true;
    cout << "Connected To Server At ('" << ip << "', " <<
    server_tcp_port << ")\n";


    return true;
}


bool Client::tryLogIn(const string& username, const string& password,
string& error)
{

    if (username.size() <= 4)
    {
        error = "Username Is Too Short";
        return false;
    }
    if (password.size() <= 4)
    {
        error = "Password Is Too Short";
        return false;
    }

    if (username.find('~') != username.npos)
    {
        error = "Username Cannot Have Tilde ~";
        return false;
    }




    // Connect to server
    if (!connectToServer(error))
```

```
        return false;


    string creds = "LOGIN~" + username + "~" + password + "~";

    sendEncryptedTCP(creds, error);



    void* buffer;
    int buffer_size;
    if (!recvEncryptedTCP(buffer, buffer_size, error))
    {
        error = "Failed To Receive Message From Server";
        return false;
    }

    string response((char*)buffer, buffer_size);
    free(buffer);

    vector<string> parts = split(response);

    if (parts[0] == "SUCCESS")
    {
        this->username = username;
        player_id = std::stoi(parts[1]);
        cout << "Player ID: " << player_id << "\n";
        return true;
    }



    error = parts[1];

    return false;



}
bool Client::trySignUp(const string& username, const string& password,
string& error)
{

    if (username.size() <= 4)
    {
        error = "Username Is Too Short";
        return false;
    }
    if (password.size() <= 4)
    {
        error = "Password Is Too Short";
        return false;
    }
```

```
    if (username.find('~') != username.npos)
    {
        error = "Username Cannot Have Tilde ~";
        return false;
    }


    // Connect to server
    if (!connectToServer(error))
        return false;



    string creds = "SIGNUP~" + username + "~" + password + "~";

    sendEncryptedTCP(creds, error);



    void* buffer;
    int buffer_size;
    if (!recvEncryptedTCP(buffer, buffer_size, error))
    {
        error = "Failed To Receive Message From Server";
        return false;
    }

    string response((char*)buffer, buffer_size);
    free(buffer);

    vector<string> parts = split(response);

    if (parts[0] == "SUCCESS")
    {

        return true;
    }



    error = parts[1];

    return false;



}
//UDP
//encrypts message, sends the (char)id + the encrypted message.
bool Client::sendEncryptedUDP(void* buffer, int size, string& error)
{
```

```
    string encrypted = encryptAES(string((char*)buffer, size),
    key_bytes, error);
    if (encrypted == "")
    {
        error += "encryption empty";
        return false;
    }

    string msg = (char)player_id + encrypted;

    if (!sendUDP(msg, error))
        return false;
}

bool Client::recvEncryptedUDP(void*& buffer, int& buffer_size, string&
error)
{
    if (!recvUDP(buffer, buffer_size))
    {
        error = "Error Trying to receive encrypted udp";
        return false;
    }

    string ciphertext((char*)buffer, buffer_size);


    string decrypted = decryptAES(ciphertext, key_bytes, error);
    if (decrypted == "")
    {
        cout << "ERROR when decrypting the shit from the cunt: " <<
        error << "\n";
        return false;
    }

    buffer_size = decrypted.size();

    memcpy(buffer, decrypted.c_str(), decrypted.size());
}


bool Client::sendUDP(const string& message, string& error)
{

    int buffer_size = 128;
    void* buffer = malloc(buffer_size);
    if (buffer == 0)
    {
        error = "couldn't allocate space for message buffer";
        return false;
    }

    if (message.size() > buffer_size)
    {
```

```
        error = "message size bigger than 128 bytes, " +
        std::to_string(message.size()) + " bytes without null";
        free(buffer);
        return false;
    }

    // message string
    memcpy(buffer, message.c_str(), message.size());

    if (udp_socket.send(buffer, message.size(), udp_sender_address,
    udp_sender_port) != sf::Socket::Done)
    {
        error = "Failure When Sending The Message: " + message;
        free(buffer);
        return false;
    }


    free(buffer);
}

// returns true on success
bool Client::recvUDP(void*& buffer, int& buffer_size)
{
    //buffer
    int msg_length = 256;
    buffer = malloc(msg_length);
    if (buffer == 0)
    {
        cout << "couldn't allocate space for payload\n";
        return false;
    }

    size_t amount_received;
    sf::Socket::Status status = udp_socket.receive(buffer, msg_length,
    amount_received, udp_listener_address, udp_listener_port);

    if (status == sf::Socket::Done)
    {
        buffer_size = amount_received;
        return true;
    }

    if (status == sf::Socket::Error)
        cout << "Error Receiving UDP\n";

    free(buffer);

    return false;
}

// TCP
bool Client::sendEncryptedTCP(const string& msg, string& error)
```

```
{
    string encrypted = encryptAES(msg, key_bytes, error);
    if (encrypted == "")
        return false;

    sendTCP(tcp_socket, encrypted, error);
}

bool Client::recvEncryptedTCP(void*& buffer, int& buffer_size, string&
error)
{
    if (!recvTCP(tcp_socket, buffer, buffer_size))
    {
        error = "Error Trying to receive encrypted tcp";
        return false;
    }

    string ciphertext((char*)buffer, buffer_size);


    string decrypted = decryptAES(ciphertext, key_bytes, error);
    if (decrypted == "")
    {
        cout << "ERROR when decrypting the shit from the cunt: " <<
        error << "\n";
        return false;
    }


    memcpy(buffer, decrypted.c_str(), decrypted.size());

}

// returns true on success
bool Client::recvTCP(sf::TcpSocket& socket, void*& buffer, int&
buffer_size)
{
    socket.setBlocking(false);

    sf::SocketSelector selector;
    selector.add(socket);


    if (!selector.wait(sf::milliseconds(1000)))
    {
        cout << "Nothing Received\n";
        socket.setBlocking(true);
        return false;
    }

    socket.setBlocking(true);

    //length
```

```
    short msg_len = 0;
    size_t amount_received;
    if (socket.receive(&msg_len, 2, amount_received) !=
    sf::Socket::Done || amount_received != 2)
    {
        cout << "Error when receiving msg length\n";
        return false;
    }

    buffer = malloc(msg_len);
    if (buffer == 0)
    {
        cout << "couldn't allocate space for payload\n";
        return false;
    }


    // message string
    socket.receive(buffer, msg_len, amount_received);


    if (amount_received != msg_len)
    {
        cout << "Error when receiving message with length: " << msg_len
        << "\n";
        return false;
    }

    buffer_size = amount_received;
    return true;
}

bool Client::sendTCP(sf::TcpSocket& socket, const string& message,
string& error)
{
    //cout << "Sending: " << message <<  "\n";

    int buffer_size = 2 + message.size();
    void* buffer = malloc(buffer_size);
    if (buffer == 0)
    {
        error = "couldn't allocate space for message buffer";
        return false;
    }

    // length
    short msg_len = message.size();
    memcpy(buffer, &msg_len, 2);

    // message string
    memcpy((char*)buffer + 2, message.c_str(), msg_len);
```

```
    size_t amount_sent;
    if (socket.send(buffer, buffer_size, amount_sent) !=
    sf::Socket::Done)
    {
        error = "Failure When Sending The Message: " + message;
        free(buffer);
        return false;
    }

    free(buffer);

    if (amount_sent != buffer_size)
    {
        error = "Error sending message, not whole message sent: " +
        message;
        return false;
    }

}

void printBytes(const unsigned char* pBytes, const uint32_t nBytes) //
should more properly be std::size_t
{
    for (uint32_t i = 0; i < nBytes; i++)
    {
        std::cout <<
            std::hex <<           // output in hex
            std::setw(2) <<       // each byte prints as two characters
            std::setfill('0') <<  // fill with 0 if not enough
            characters
            (int)pBytes[i] << " ";
    }
    cout << std::dec << "\n";

}

void printBytes(void* pBytes, const uint32_t nBytes)
{
    printBytes((unsigned char*)pBytes, nBytes);
}

string encryptAES(const void*& buffer, int size, unsigned char* key,
string& error) {
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        error = "Error creating EVP_CIPHER_CTX.";
        return "";
    }

    if (EVP_EncryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL) !=
    1) {
        cout << "Error initializing AES encryption.";
        EVP_CIPHER_CTX_free(ctx);
```

```
        return "";
    }

    string ciphertext(size + EVP_CIPHER_block_size(EVP_aes_128_ecb()),
    '\0');
    int outLen;
    if (EVP_EncryptUpdate(ctx, reinterpret_cast<unsigned
    char*>(&ciphertext[0]), &outLen, reinterpret_cast<const unsigned char*>(buffe
        error = "Error encrypting data.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    int finalLen;
    if (EVP_EncryptFinal_ex(ctx, reinterpret_cast<unsigned
    char*>(&ciphertext[outLen]), &finalLen) != 1) {
        error = "Error finalizing encryption.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    EVP_CIPHER_CTX_free(ctx);
    ciphertext.resize(outLen + finalLen);
    return ciphertext;
}

string encryptAES(const string& plaintext, unsigned char* key, string&
error) {
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        error = "Error creating EVP_CIPHER_CTX.";
        return "";
    }

    if (EVP_EncryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL) !=
    1) {
        cout << "Error initializing AES encryption.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    string ciphertext(plaintext.size() +
    EVP_CIPHER_block_size(EVP_aes_128_ecb()), '\0');
    int outLen;
    if (EVP_EncryptUpdate(ctx, reinterpret_cast<unsigned
    char*>(&ciphertext[0]), &outLen, reinterpret_cast<const unsigned char*>(plain
        error = "Error encrypting data.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    int finalLen;
    if (EVP_EncryptFinal_ex(ctx, reinterpret_cast<unsigned
```

```
    char*>(&ciphertext[outLen]), &finalLen) != 1) {
        error = "Error finalizing encryption.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    EVP_CIPHER_CTX_free(ctx);
    ciphertext.resize(outLen + finalLen);
    return ciphertext;
}

string decryptAES(const string& ciphertext, unsigned char* key, string&
error) {
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        error = "Error creating EVP_CIPHER_CTX.";
        return "";
    }

    if (EVP_DecryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL) !=
    1) {
        error = "Error initializing AES decryption.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    string plaintext(ciphertext.size(), '\0');
    int outLen;
    if (EVP_DecryptUpdate(ctx, reinterpret_cast<unsigned
    char*>(&plaintext[0]), &outLen, reinterpret_cast<const unsigned char*>(cipher
        error = "Error decrypting data.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    int finalLen;
    if (EVP_DecryptFinal_ex(ctx, reinterpret_cast<unsigned
    char*>(&plaintext[outLen]), &finalLen) != 1) {
        error = "Error finalizing decryption.";
        EVP_CIPHER_CTX_free(ctx);
        return "";
    }

    EVP_CIPHER_CTX_free(ctx);
    plaintext.resize(outLen + finalLen);
    return plaintext;
}

void bigintToBytes(bigint key, unsigned char* buffer)
{
    for (int i = 15; i >= 0; i--)
    {
        buffer[i] = (unsigned char)(key & 255);
```

```
        key >>= 8;
    }
}
```