Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Сетевое программирование. Обмен сообщения через сеть

по дисциплине «Объектно-ориентированное программирование»

Выполнил студент группы 3331506/00401 Казанцев Г.В.
Руководитель Ананьевский М.С.

Санкт-Петербург
2023

Оглавление

Введение

Сегодня миллионы компьютеров и устройств связаны в глобальную сеть интернет, либо в отдельные локальные подсети. В связи с этим возникает необходимость создания приложений, которые бы использовали все преимущества передачи данных по сети. Например, одним из распространенных приложений, которое использует передачу по сети, является веб-браузер. Язык программирования C++ предоставляет все необходимые возможности для создания приложений, которые могут взаимодействовать по сети и использовать различные сетевые протоколы.

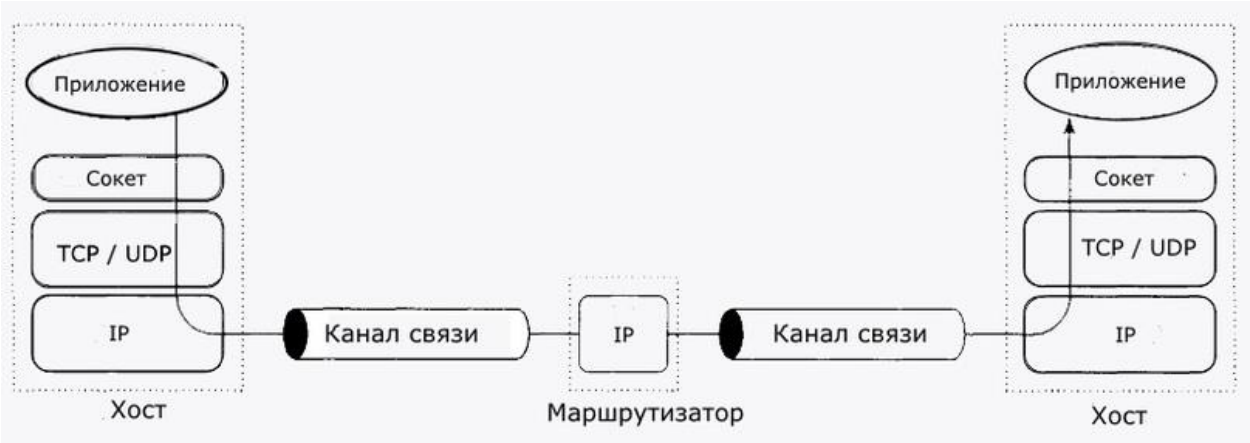
Но, прежде чем переходить непосредственно к созданию приложений, надо пару слов сказать, что вообще представляет собой коммуникация в сети.

Вся сеть состоит из отдельных элементов - хостов, которые представляют собой компьютеры и другие подключенные устройства. Между собой они соединены каналами связи (кабели Ethernet, Wi-Fi и т.д.) и маршрутизаторами. Маршрутизаторы объединяют компьютеры в подсети и контролируют передачу данных между ними. Наиболее простой пример маршрутизатора является wi-fi роутер.

Глава I. Теоретическая часть

Компьютеры-хосты не могут взаимодействуют без специальных правил и принципов. Для этого они применяют протоколы. Протокол представляет собой соглашения о том, как пакеты данных будут передаваться по каналам коммуникации. Таким образом, протокол упорядочивает взаимодействие.

Существует множество различных протоколов. Протоколы, которые используются для передачи данных по сети, составляют семейство протоколов TCP/IP. Основные из них: Internet Protocol (IP), Transmission Control Protocol (TCP) и User Datagram Protocol (UDP). Причем эти протоколы организованы в уровневую систему:



IP представляет сетевой уровень. Он использует нижележащие уровни, которые представляют физические каналы коммуникации - кабели Ethernet и т.д., для передачи пакетов с данными другому хосту.

Выше IP располагается транспортный уровень, который образуют протоколы TCP и UDP. Эти протоколы используют определенные порты для передачи данных. TCP позволяет отследить потерю пакетов и их дублирование при передаче. UDP подобного не позволяет сделать и нацелен на простую передачу данных.

Однако приложение взаимодействует с уровнем TCP / UDP не напрямую, а через специальный API, который предоставляют сокеты. Сокеты – это интерфейс для создания сетевых приложений, который опирается на встроенные возможности операционной системы.

Сокеты

В зависимости от используемого протокола различают два вида сокетов: потоковые сокеты (stream socket) и дейтаграммные сокеты (datagram socket). Потоковые сокеты используют протокол TCP, дейтаграммные - протокол UDP. Существует во всех операционных системах (Unix, windows) и имеет схожий принцип работы. В операционной системе Windows есть свои особенности при работе с сокетами. Например, перед работой интерфейс необходимо инициализировать. В итоге, когда приложение посылает сообщение приложению, запущенному на другом хосте, то приложение обращается к сокетам для передачи данных на уровень TCP / UDP. Далее с этого транспортного уровня данные передаются сетевому уровню - уровню протокола IP. И этот протокол передает данные далее физическим уровням, и после этого данные уходят по сети.

Взаимодействие бывает двух видов: синхронное и асинхронное.
В этой курсовой работе рассматривается только синхронное взаимодействие.

IPv4

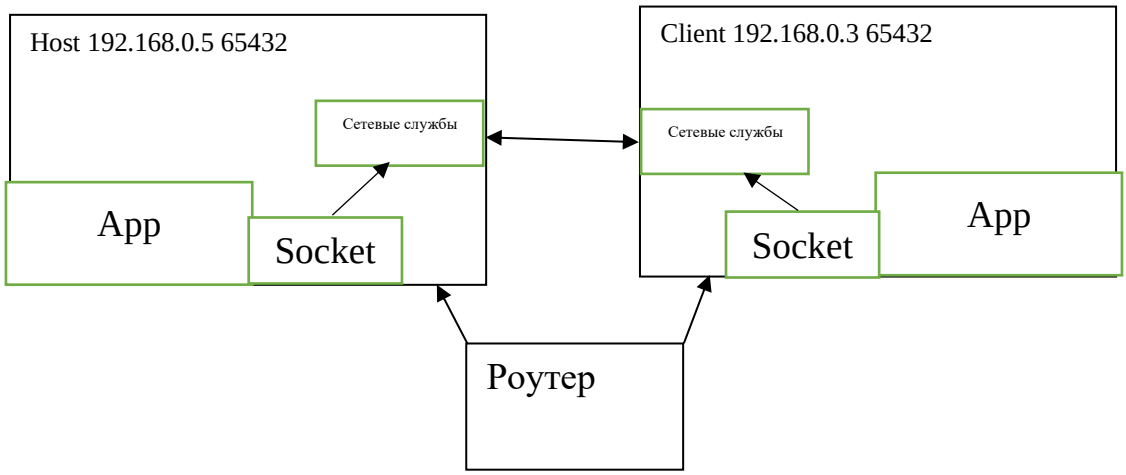
Чтобы уникально определять хосты в сети каждый хост имеет адрес. Существует несколько различных протоколов адресов. В настоящее время наиболее распространен протокол IPv4, который предполагает представление адреса в виде 32-битного числа, например, 192.168.0.1. Такой адрес содержит четыре числа, разделенных точками, и каждое число находится в диапазоне от 0 до 255. Однако также в последнее время набирает оборот использование адресов протокола IPv6, которые представляют собой 128-битное значение. Кроме адреса при сетевых взаимодействиях используются порты. Порт представляет 16-битное число в диапазоне от 1 до 65 535. Использование

портов позволяет разграничить несколько запущенных приложений на одном хосте. Причем, чаще всего первые 1024 портов зарезервированы системой и их использование нежелательно, поэтому выберем порт 65432.

Взаимодействие клиент-сервер

Ключевыми компонентами сетевого взаимодействия являются клиент и сервер. Клиент посылает запрос, а сервер получает запрос, обрабатывает его и посылает обратно клиенту некоторый ответ. Простейший пример - веб-браузер, который служит в качестве клиента, отправляя запрос на некоторый сайт. А сайт выступает в качестве сервера, отправляя браузеру некоторый ответ, который браузер затем отображает пользователю. Однако в реальности нередко одно приложение может выступать и в качестве сервера, и в качестве клиента.

В курсовой работе рассматривался вариант взаимодействия двух компьютеров, находящихся в одной подсети роутера. В данном случае не имеет значение какой компьютер является сервером, а какой клиентом.



Глава II. Практическая часть

В этой части будут показаны программные коды для сервера и клиента, а также рассмотрены их основные принципы и особенности работы. Для работы с сокетами в системе windows предусмотрены библиотеки *win2sock.h* и *ws2tcpip.h*, а также подключим *windows.h.*

Для более удобной работы с сокетами создадим класс, в котором будут описаны переменные (*str, ConnectSocket, ListenSocket, ClientSocket, recvdata*) и структуры (*hints и addrResult)* и методы (*init, run, close, stop и send_message*).

Клиент

В классе *SocketClient* опишем переменные и методы упомянутые выше. Рассмотрим процесс создания соединения со стороны клиента по протоколу *tcp/ip*. Сначала необходимо инициализировать сокет. За инициализацию сокета в *win2sock.h* отвечает функция **WSAStartup**. На вход принимает версию интерфейса *windows socket* и указатель на структуру, которая возвращает данная функция. Версия 2.2 – последняя версия сокетов, поэтому будем использовать именно ее. Поскольку функция WSAStartup (данная функция возвращает 0 при удачном выполнении или 1, если в результате работы функции возникла ошибка) может не всегда правильно отработать, то необходимо предусмотреть проверку правильности выполнения. Следующим шагом необходимо задать параметры соединения. Поскольку это код клиента, то в **ADDRINFO** необходимо задать параметры сервера, к которому будет подключаться клиент. Для этого необходимо использовать функцию *getaddrinfo*, которая получает на вход адрес и порт сервера, параметры подключения (назовем переменную hints). &hints – указатель на структуру ADDRINFO, которая содержит некоторые параметры соединения. Функция getaddrinfo() также возвращает значение 0 или 1.

Перед использованием этой функции объявим параметры соединения.

- Семейство адресов: hints.ai_family на данный момент поддерживаются два значения AF_INET или AF_INET6, которые являются форматами семейств интернет-адресов для IPv4 и IPv6. Для удобства и простоты демонстрации работы приложений выберем протокол IPv4.
- Далее необходимо задать тип сокета ai_socktype(SOCK_STREAM или SOCK_DGRAM). Поскольку в курсовой работе стоит задача передачи

сообщений между двумя компьютерами, то выберем SOCK_STREAM, так как он использует протокол передачи данных *tcp*.

- Далее зададим тип протокола ai_protocol, который зависит от указанного семейства и типа сокета. Выберем IPPROTO_TCP, так как мы уже выбрали SOCK_STREAM и AF_INET.

  IPPROTO_TCP - протокол управления передачей (TCP). Это возможное значение, когда элемент *ai_family* — AF_INET или AF_INET6, а элемент *ai_socktype* — SOCK_STREAM .

Поскольку функция *getaddrinf*o учитывает все ненулевые поля структуры параметров соединения hints, то необходимо перед заданием параметров соединения обнулить все параметры. Реализуем это встроенной функцией ZeroMemory.

В процессе выполнения функции getaddrainfo могут возникнуть ошибки. Если функция отработает неправильно, то необходимо деинициализировать сокет **WSACleanup**, поскольку уже была вызвана **WSAStartup**.

Рассмотрим функции для работы с сокетом.

Таблица 1 – Используемые в работе функции сокетов

socket	Функция *socket* создает дескриптор, идентифицирующий сокет, привязанный к определенному поставщику транспортных услуг. На вход получает семейство адресов, тип сокета и тип протокола. Возвращает INVALID_SOCKET в случае возникновении ошибок.
connect	Функция *connect* устанавливает соединение с указанным сокетом. На вход получает дескриптор, идентифицирующий неподключенный сокет, указатель на структуру *ai_addr*, с которой должно быть установлено соединение и длинну в байтах структуры, на которую указывает *ai_addrlen*.

s*hutdown*	Функция *shutdown* отключает отправку или получение через сокет. На вход получает дескриптор, идентифицирующий сокет и флаг, описывающий какие действия будут запрещены с сокетом (SD_SEND – отключены операции отправки данных).
closesocket	Функция *closesocket* закрывает существующий сокет. На вход получает дескриптор, идентифицирующий сокет, который необходимо закрыть.
send	Функция отправки *send* отправляет данные в подключенный сокет. На вход принимает дескриптор, идентифицирующий подключенный сокет, буфер данных, предназначенный для передачи, длину в байтах данных в буфере и набор флагов (в нашем случае подадим 0).
recv	Функция приема *recv* получает данные из подключенного сокета или связанного сокета без установления соединения. На вход получает дескриптор, идентифицирующий подключенный сокет, указатель на буфер, предназначенный для приема данных, длину в байтах этого буфера и набор флагов, влияющих на поведении функции (в нашем случае зададим 0).
bind	Функция привязки b*ind* связывает локальный адрес с сокетом. На вход получает дескриптор, идентифицирующий несвязанный сокет, указатель на структуру *ai_addr*, с которой должно быть установлено соединение и длинну в байтах структуры, на которую указывает *ai_addrlen*.
accept	Функция *accept* разрешает попытку входящего соединения через сокет. На вход получает инициализированный

	прослушивающий сокет. Остальные параметры необязательные, поэтому *nullptr*.
listen	Функция прослушивания *listen* помещает сокет в состояние, в котором он ожидает входящего соединения. На вход получает связанный неподключенный сокет и максимальное количество подключений (данный параметр установим в 1, поскольку клиент будет всего один)

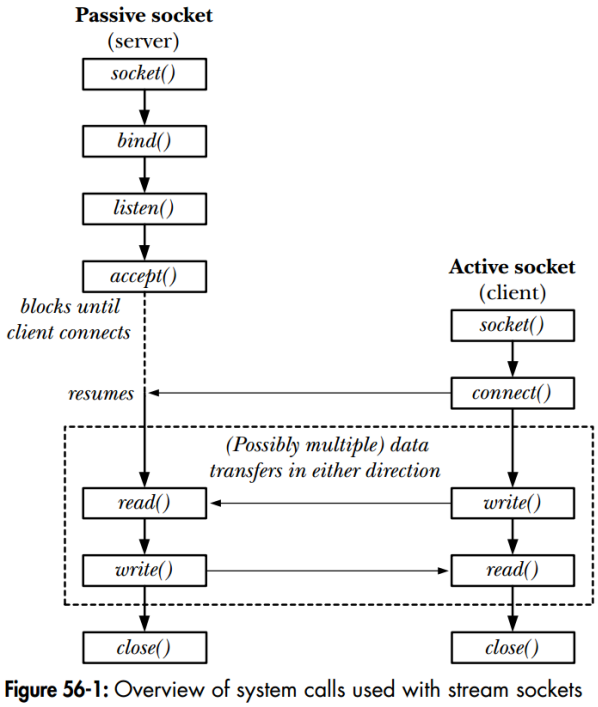


Figure 56-1: Overview of system calls used with stream sockets

Рисунок 1 – Общая схема работы сокетов

Сначала создадим сокет функцией socket и подключим его к серверу функцией connect. Для структурирования и удобства использования и доработки программы поместим все эти действия в метод класса *init*. Если сокет создан, то его необходимо останавливать (ограничивать передачу или прием данных) и закрывать. Для этого используем функции shutdown и *closesocket* и выделим эти действия в отдельные методы класса stop и close соответственно. Остается написать основной метод класса run, в котором будет происходит прием и отправка сообщений.

Поскольку функция *send* отправляет *char* массив символов возникла проблема с тем, что символы после первого пробела отправляются следующим пакетом данных. Для решения проблемы реализуем временный массив *char s_data*, которой будет создаваться перед приемом нового пакета данных и удаляться после обработки.

```
const char* s_data = new char[512];
s_data = str.c_str();
result = con.send_message(ConnectSocket, s_data);
s_data = nullptr;
delete[] s_data;
```

После отправки данных используем функцию сокетов приема данных recv. Сформируем цикл из этих двух примитивных действий, поставив условия для выхода булеву переменную _exit, которая будет являться флагом. В случае возникновения ошибки передачи данных или закрытия сокета со стороны сервера _exit будет устанавливаться в ноль и при следующей проверке условия, произойдет выход из цикла.

В основной функции main() вызовем три метода класса *SocketClient: init, run и close*. Полный программный код клиента представлен в приложении А.

В данном подразделе описано полное описание создание соединения *tcp/ip* со стороны клиента, написаны обработчики некоторых ошибок и реализована правильная логика работы сокета.

Сервер

Программный код сервера (полный программный код представлен в приложении Б) писался на основе клиента для экономии человеческих ресурсов. Поэтому, исходя из рисунка (1) будет уместней описать отличающиеся части в программной коде.

Первым делом необходимо создать переменную слушающего сокета.

У структуры **ADDRINFO** есть поле отвечающий за флаги ai_flags. Установим в это поле = **AI_PASSIVE**

Задание флага AI_PASSIVE указывает, что вызывающий объект намерен использовать возвращаемую структуру адреса сокета в вызове функции привязки. Если задан этот флаг, то сокет становится серверным и имеет функции привязки и прослушки.

В функции *getaddrinfo* вместо значения сокета подаем *nullptr*, потому что мы ожидаем запроса клиента на порт 65432.

1. Создаем сокет функцией *socket*.
2. Привязываем сокет клиента (*ClientSocket*) к слушающему сокету (*ListenSocket*).
3. Функцией accept разрешаем соединение и дальнейшую передачу данных через сокет.

Остановка работы сокета, его закрытие, передача и прием данных реализованы также, как и у клиентского сокета. В методе класса run функции отправки и приема данных поменяны местами для того, чтобы обмен данных синхронизовался.

Заключение

В ходе курсовой работы были изучены особенности и основные принципы работы сокетов для windows (с помощью стандартной библиотеки *winsock2*) на языке программирования c++. Освоены основные способы взаимодействия с сокетами: инициализация, создание, привязка, прослушка сокета, отправка и прием данных.

Результатом работы является два приложения для сервера и клиента, которые синхронно обмениваются сообщения в режиме реального времени. Программа также в полной мере предусматривает все ошибки, которые могут возникнуть в результате работы.

Список литературы

1. –URL:https://learn.microsoft.com/ru-ru/windows/win32/api/_winsock// (дата обращения: 22.04.2023)
2. Джонс, Энтони; Олунд, Джим (2002). Сетевое программирование для Microsoft Windows. ISBN 0-7356-1579-9.

Приложение А

```
//client
#define WIN32_LEAN_AND_MEAN
#include <iostream>
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <config.h>
#include <stdlib.h>
#include <string>
//#define DEBUG

using namespace std;


class SocketClient {
public:
   void init();
   void run(ADDRINFO* addrresult, SOCKET connectsocket, SocketClient con);
   void close(ADDRINFO* addrresult, SOCKET connectsocket);
   void stop(ADDRINFO* addrresult, SOCKET connecttsocket);
   int send_message(SOCKET connectsocket, const char s_data[]);

   string str;
   WSADATA wsaData;
   ADDRINFO hints;
   ADDRINFO* addrResult = nullptr;
   SOCKET ConnectSocket = INVALID_SOCKET;
   char recvdata[512];
};

void SocketClient::init()
{
   result = WSAStartup(MAKEWORD(2, 2), &wsaData);
```

```
    if (result != 0)
    {
#ifdef DEBUG
        cout << "WSAStartup failed, result = " << result << endl;
#endif
        exit(1);
    }
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    result = getaddrinfo(HOST, PORT, &hints, &addrResult);
    if (result != 0)
    {
#ifdef DEBUG
        cout << "getaddrinfo failed with error: " << result << endl;
#endif
        WSACleanup();
        exit(2);
    }

    ConnectSocket = socket(addrResult->ai_family, addrResult->ai_socktype,
addrResult->ai_protocol);
    if (ConnectSocket == INVALID_SOCKET)
    {
#ifdef DEBUG
        cout << "Socket creation failed" << endl;
#endif

        freeaddrinfo(addrResult);
        WSACleanup();
        exit(3);
    }
    result = connect(ConnectSocket, addrResult->ai_addr, (int)addrResult-
>ai_addrlen);
    if (result == SOCKET_ERROR)
    {
#ifdef DEBUG
        cout << "Unable to connect to server";
#endif
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
        freeaddrinfo(addrResult);
        WSACleanup();
```

```
        exit(4);
    }
    cout << "Connected to " << HOST << " " << endl;
}


void SocketClient::close(ADDRINFO* addrresult, SOCKET connectsocket)
{
    int result = shutdown(connectsocket, SD_SEND);
    if (result == SOCKET_ERROR)
    {
#ifdef DEBUG
        cout << "send failed, error: " << result << endl;
#endif
        closesocket(connectsocket);
        freeaddrinfo(addrresult);
        WSACleanup();
        exit(6);
    }
}

void SocketClient::stop(ADDRINFO* addrresult, SOCKET connectsocket)
{
    closesocket(connectsocket);
    freeaddrinfo(addrresult);
    WSACleanup();
}



int SocketClient::send_message(SOCKET connectsocket, const char s_data[])
{
    return send(ConnectSocket, s_data, (int)strlen(s_data), 0);
}

void SocketClient::run(ADDRINFO* addrresult, SOCKET connectsocket,
SocketClient con)
{
    bool _exit = 0;
    int result;
    do
    {
        cout <<endl << "Enter message";
        getline(cin, str);
```

```
        const char* s_data = new char[512];
        s_data = str.c_str();
        result = con.send_message(ConnectSocket, s_data);
        s_data = nullptr;
        delete[] s_data;

        result = recv(connectsocket, recvdata, 512, 0);
        if (result > 0)
        {
#ifdef DEBUG
            endl << "Received " << result << "bytes";
#endif
            cout << "Recieved data: " << recvdata;

        }
        else if (result == 0)
        {
            cout << "Connected closed" << endl;
            _exit = 1;
        }
        else
        {
            cout << "recv failed with error " << endl;
            _exit = 1;
        }

    } while (_exit != 1);
}


int main(int argc, char const* argv[])
{
    SocketClient newcon;
    newcon.init();
    newcon.run(newcon.addrResult, newcon.ConnectSocket, newcon);
    newcon.close(newcon.addrResult, newcon.ConnectSocket);
    return 0;
}
```

Приложение Б

```
//server

#define WIN32_LEAN_AND_MEAN
#include <iostream>
#include <windows.h>
```

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <string>
#include <stdlib.h>
//#define DEBUG
const char* HOST = "192.168.0.5";
const char* PORT = "65432";
using namespace std;

class SocketServer {
public:
    void init();
    void run(ADDRINFO* addrresult, SOCKET clientsocket, SOCKET
listensocket, SocketServer con);
    void close(ADDRINFO* addrresult, SOCKET clientsocket);
    void stop(ADDRINFO* addrresult, SOCKET clientsocket);
    int send_message(SOCKET clientsocket, const char s_data[]);
    string str;
    char recieved_data[512];
    WSADATA wsaData;
    ADDRINFO hints;
    ADDRINFO* addrResult = nullptr;
    SOCKET ClientSocket = INVALID_SOCKET;
    SOCKET ListenSocket = INVALID_SOCKET;
    int result;

};

void SocketServer::init()
{
    result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0)
    {
#ifdef DEBUG
        cout << "WSAStartup failed, result = " << result << endl;
#endif
        exit(1);
    }
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;
        result = getaddrinfo(nullptr, PORT, &hints, &addrResult);
```

```
    if (result != 0)
    {
#ifdef DEBUG
        cout << "getaddrinfo failed with error: " << result << endl;
#endif
        WSACleanup();
        exit(1);
    }


    ListenSocket = socket(addrResult->ai_family, addrResult->ai_socktype,
addrResult->ai_protocol);
    if (ListenSocket == INVALID_SOCKET)
    {
#ifdef DEBUG
        cout << "Socket creation failed" << endl;
#endif
        freeaddrinfo(addrResult);
        WSACleanup();
        exit(1);
    }
    cout << "+New server started from ipv4 address: " << HOST << " port: " <<
PORT << endl;
    result = bind(ListenSocket, addrResult->ai_addr, (int)addrResult->ai_addrlen);
    if (result == SOCKET_ERROR)
    {
#ifdef DEBUG
        cout << "Binding socket failed";
#endif
        closesocket(ListenSocket);
        ListenSocket = INVALID_SOCKET;
        freeaddrinfo(addrResult);
        WSACleanup();
        exit(1);
    }
    result = listen(ListenSocket, 1);
    if (result == SOCKET_ERROR)
    {
#ifdef DEBUG
        cout << "Listening socket failed";
#endif
        closesocket(ListenSocket);
        freeaddrinfo(addrResult);
        WSACleanup();
```

```
        exit(1);
    }

    ClientSocket = accept(ListenSocket, nullptr, nullptr);
    if (ClientSocket == INVALID_SOCKET)
    {
#ifdef DEBUG
        cout << "accept failed, error: " << result << endl;
#endif
        closesocket(ListenSocket);
        freeaddrinfo(addrResult);
        WSACleanup();
        exit(1);
    }
    cout << "Connected has been established " << HOST << " " << PORT << endl;
}


void SocketServer::close(ADDRINFO* addrresult, SOCKET clientnsocket)
{
    closesocket(clientsocket);
    freeaddrinfo(addrresult);
    WSACleanup();
    exit(1);
}

void SocketServer::stop(ADDRINFO* addrresult, SOCKET clientsocket)
{
    int result = shutdown(clientsocket, SD_SEND);
    if (result == SOCKET_ERROR)
    {
#ifdef DEBUG
        cout << "shutdown failed, error: " << result << endl;
#endif // DEBUG
        closesocket(clientsocket);
        freeaddrinfo(addrresult);
        WSACleanup();
        exit(1);
    }
}

int SocketServer::send_message(SOCKET clientsocket, const char s_data[])
{
    return send(clientsocket, s_data, (int)strlen(s_data), 0);
```

```
}

void SocketServer::run(ADDRINFO* addrresult, SOCKET clientsocket, SOCKET
listensocket, SocketServer con)
{
   bool _exit = 0;
   int result;

   do
   {
      result = recv(clientsocket, recieved_data, 512, 0);
      if (result > 0)
      {
#ifdef DEBUG
         cout << "Received " << result << "bytes" << endl;
#endif // DEBUG
         cout << "Recieved data: " << recieved_data << endl;
         cout << "Enter message";
         getline(cin, str);
         const char* s_data = new char[512];
         s_data = str.c_str();
         result = con.send_message(ClientSocket, s_data);
         s_data = nullptr;
         delete[] s_data;
      }
      else if (result == 0)
      {
#ifdef DEBUG
         cout << "Connected closing" << endl;
#endif // DEBUG
         _exit = 1;
      }
      else
      {
#ifdef DEBUG
         cout << "recv failed with error " << endl;
#endif // DEBUG
         _exit = 1;
      }
   }
   while (_exit != 1);
}
```

```
int main(int argc, char const* argv[])
{
    SocketServer newcon;
    newcon.init();
    newcon.run(newcon.addrResult, newcon.ClientSocket, newcon.ListenSocket,
newcon);
    newcon.close(newcon.addrResult, newcon.ClientSocket);
    return 0;
}
```