

**МИНОБРНАУКИ РОССИИ**  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
«САНКТ ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Курсовая работа**

По дисциплине

«Объектно-ориентированное программирование»

на тему:

**«Серверное программирование»**

Студент группы 33331506/00401 \_\_\_\_\_

М.А. Гавриленко

Преподаватель \_\_\_\_\_

М.С. Ананьевский

«\_\_» \_\_\_\_\_ 2023

Санкт-Петербург

2023

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Основная часть</b>	<b>4</b>
1.1 Принцип работы . . . . .	4
1.1.1 Библиотека Winsock2.h . . . . .	4
1.1.2 Параметры функций . . . . .	5
<b>2 Программный код</b>	<b>8</b>
2.1 Код для сервера . . . . .	8
2.1.1 Подключение библиотек . . . . .	8
2.1.2 Работа с сокетом . . . . .	9
2.1.3 Выделение потока под клиента . . . . .	10
2.1.4 Обработка клиента . . . . .	11
2.1.5 Пример работы . . . . .	12
2.2 Код для клиента . . . . .	14
2.2.1 Чтение с сервера . . . . .	14
2.2.2 Ошибка MinGW . . . . .	14
2.2.3 Пример работы . . . . .	14
<b>Заключение</b>	<b>17</b>
<b>Список литературы</b>	<b>18</b>

# Введение

Сетевое программирование является ключевой технологией в области разработки современных приложений. С помощью сетевого программирования приложения могут обмениваться данными через сеть, что делает их более гибкими и масштабируемыми. Сокеты, в свою очередь, являются одним из основных инструментов сетевого программирования.

В данном курсовом проекте мы будем изучать сетевое программирование и сокеты. Мы рассмотрим основные понятия, такие как IP-адреса, протоколы и порты, и узнаем, как они используются при создании сетевых приложений. Мы также рассмотрим различные типы сокетов и поймем, как они работают.

- **Сокеты** — это интерфейс программирования приложений (API), который позволяет приложениям создавать сетевые соединения и обмениваться данными между собой через сеть. Сокеты могут использоваться для создания различных типов сетевых приложений, таких как клиент-серверные приложения, чат-приложения и другие. Сокеты работают на различных уровнях стека протоколов, таких как TCP/IP, UDP, HTTP и других.
- **IP-адреса** — это числовые идентификаторы, используемые для идентификации устройств, подключенных к сети. IP-адреса могут быть локальными или глобальными, и они используются для маршрутизации сетевых пакетов между устройствами. IP-адреса могут быть представлены в различных форматах, таких как IPv4 и IPv6.
- **TCP (Transmission Control Protocol)** — это протокол передачи данных, который обеспечивает надежное и упорядоченное соединение между двумя устройствами в сети. TCP используется в большинстве приложений, которые требуют надежной передачи данных, таких как веб-сайты, электронная почта и другие.
- **HTTP (Hypertext Transfer Protocol)** — это протокол передачи данных, который используется для обмена информацией между веб-серверами и клиентскими приложениями, такими как веб-браузеры. HTTP используется для передачи HTML-страниц, изображений, видео и других типов данных в Интернете.

# 1 Основная часть

## 1.1 Принцип работы

Для начала необходимо определиться, что необходимо для получения желаемого результата, путем поиска в различных инструментах был построен псевдокод программы:

1. Подключение библиотеки для работы с сокетами.
2. Создание сокета сервера и связывание его с портом.
3. Связывание сокета с адресом.
4. Прослушивание порта.
5. Ожидание новых соединений от клиентов.
6. Чтение и отправка сообщений адресатам.
7. Закрытие сокетов и освобождение памяти.

### 1.1.1 Библиотека Winsock2.h

Библиотека winsock2.h (Windows Sockets 2) предоставляет набор функций для работы с сокетами в операционной системе Microsoft Windows. Она позволяет создавать клиент-серверные приложения, используя протоколы TCP/IP, UDP/IP, и другие.

Winsock2.h содержит определения для структур данных, использующиеся для представления информации о сокетах, а также для функций, которые позволяют создавать, настраивать, и обрабатывать сокеты.

Среди функций, которые мы будем использовать при работе с библиотекой winsock2.h, выделим следующие:

- **socket()** — создает новый сокет;
- **bind()** — связывает сокет с определенным адресом;
- **listen()** — устанавливает сокет в режим ожидания входящих соединений;

- **accept()** — принимает входящее соединение и возвращает новый сокет для общения с клиентом;
- **connect()** — устанавливает соединение с сервером;
- **send()** — отправляет данные через сокет;
- **recv()** — получает данные через сокет;
- **closesocket()** — закрывает сокет;
- **WSAStartup()** — инициализация сетевого стека Windows;<sup>1</sup>
- **WSACleanup()** — очистка используемых ресурсов и закрытие соединений с сокетами.

### 1.1.2 Параметры функций

#### 1. Параметры функции Socket:

- **address\_family**: Семейство протоколов `AF_INET` используется для протокола IPv4, а `AF_INET6` для протокола IPv6;
- **type**: определяет тип сокета. Например, `SOCK_STREAM` для TCP или `SOCK_DGRAM` для UDP;
- **protocol**: указывает конкретный протокол в выбранном семействе протоколов. Обычно присваивается значение 0, чтобы автоматически выбирать подходящий протокол для заданного типа сокета.

Возвращает дескриптор<sup>2</sup> в случае, если функция выполнялась успешно или `INVALID_SOCKET` в случае ошибки.

#### 2. Параметры функции bind:

- **SOCKET s**: идентификатор сокета, который будет связан с адресом;

---

<sup>1</sup>Инициализация сетевого стека Windows (Winsock) — это процесс подготовки приложения к работе с сетевыми функциями

<sup>2</sup>Дескриптор (descriptor) — это целое число, используемое для идентификации открытого файла или сокета в операционной системе

- **const struct sockaddr\* name:** указатель на структуру, содержащую адрес, который будет связан с сокетом;
- **int namelen:** длина структуры, содержащей адрес.

Возвращает **0** в случае, если функция выполнялась успешно или **-1** в случае ошибки.

### 3. Параметры функции listen:

- **sockfd:** дескриптор сокета, для которого устанавливается очередь ожидающих соединений;
- **backlog:** максимальное число входящих соединений, которые могут находиться в очереди ожидания.

Возвращает **0** в случае, если функция выполнялась успешно или **-1** в случае ошибки.

### 4. Параметры функции accept:

- **sockfd:** дескриптор серверного сокета, для которого нужно принять соединение;
- **addr:** указатель на структуру 'sockaddr', которая будет заполнена информацией об адресе клиента. Этот параметр можно указать как NULL, если информация об адресе клиента не требуется;
- **addrlen:** указатель на целочисленную переменную, содержащую размер структуры 'sockaddr', на которую указывает 'addr'. При вызове функции 'addrlen' должен быть инициализирован значением, равным размеру структуры 'sockaddr'.

Возвращает новый дескриптор сокета для обмена данных с клиентом или **-1** в случае ошибки.

### 5. Параметры функции connect:

- **socket:** дескриптор сокета, который должен быть использован для установления соединения;

- **address**: указатель на структуру 'sockaddr', содержащую адрес и порт удаленного сокета;
- **addrlen**: размер структуры 'sockaddr', передаваемой в качестве параметра address.

Возвращает значение **0** в случае успешного соединения или **-1** в случае ошибки.

#### 6. Параметры функции send:

- **SOCKET s**: дескриптор идентифицирующий, подключенный сокет;
- **const char\* buf**: указатель на буфер, содержащий данные для отправки;
- **len**: размер буфера в байтах
- **flags**: дополнительные флаги, задающие способ отправки данных

Функция возвращает количество отправленных байт данных в случае успеха, либо значение **SOCKET\_ERROR** в случае ошибки. Для получения дополнительной информации об ошибке можно вызвать функцию **WSAGetLastError()**.

#### 7. Параметры функции recv:

- **SOCKET s**: дескриптор сокета, из которого будут приниматься данные;
- **char\* buf**: указатель на буфер, в который будут сохранены данные;
- **len**: размер буфера в байтах
- **flags**: дополнительные флаги, задающие способ отправки данных

Функция возвращает количество полученных байт данных в случае успеха, либо значение **SOCKET\_ERROR** в случае ошибки. Для получения дополнительной информации об ошибке можно вызвать функцию **WSAGetLastError()**.

#### 8. Параметры функции closesocket:

- **SOCKET s**: дескриптор сокета, который будет закрыт;

## 2 Программный код

В этой главе нам нужно организовать два синтаксически подобных кода. Организовано многопоточное программирование при помощи **std::tread**. Код организован в соответствии с пунктом 1.1. Работа выполнена в **CLion**, также отдельно был создан **CMakeList.txt** с указанием библиотек

### 2.1 Код для сервера

В этом разделе отдельно рассматриваются блоки кода, их необходимость, реализация, а также передаваемые параметры.

#### 2.1.1 Подключение библиотек

Библиотеки подключаются в отдельном файле **chat.h**, его содержимое представлено ниже:

```
#pragma comment(lib, "Ws2_32.lib")
#include <cstring>
#include <iostream>
#include <Ws2tcpip.h>
#include <winsock2.h>
#include <thread>
#include <windows.h>
#include "vector"
#include <chrono>
#define CONNECTION_BREAK_SYMBOL '*'
bool connection_close(char* message)
{
    char* ptr = strchr(message, CONNECTION_BREAK_SYMBOL);
    if (ptr != nullptr) return true;
    return false;
}
void SET_CONSOLE_GREEN()
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
}
void SET_CONSOLE_RED()
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
}
void GET_CONSOLE_NORMAL()
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_SCREEN_BUFFER_INFO consoleInfo;
    WORD saved_attributes;
    GetConsoleScreenBufferInfo(hConsole, &consoleInfo);
    saved_attributes = consoleInfo.wAttributes;
    SetConsoleTextAttribute(hConsole, saved_attributes);
}
```



Функции `SET_CONSOLE_RED`, `SET_CONSOLE_GREEN` созданы для установки цвета консоли. В случае ошибки консоль становится красной. В случае успешной работы сервер отображает сообщения зеленым цветом.

Функция `connection_close` возвращающая булево значение, которое в других приложениях используется для условий закрытий сокета или закрытия программы.

## 2.1.2 Работа с сокетом

Этот подраздел опишет основные методы работы с сокетами. Код для последующего анализа представлен ниже:

```
#define DEFAULT_PORT 22222
#define ERROR_S "SERVER ERROR: "
#define CONNECTION_BREAK_SYMBOL '*'
#define BUFFER_SIZE 1024
#define MAX_CLIENTS 10
int main() {
    int server_socket, client_socket;
    WSADATA data;
    if(0 != WSStartup(MAKEWORD(2,1), &data)) return 101;
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
    SOCKADDR_IN server_addr, client_addr;
    socklen_t addr_size;
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        std::cerr << ERROR_S << "Error creating server socket." <<
            std::endl;
        return -1;
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(DEFAULT_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(server_socket, (sockaddr*)&server_addr, sizeof(
        server_addr)) == -1) {
        std::cerr << ERROR_S << "Error binding server socket." <<
            std::endl;
        return -1;
    }
}
```

### Анализ функции `WSStartup`

```
if(0 != WSStartup(MAKEWORD(2,1), &data)) return 101;
```

Строка инициализирует библиотеку 'WinSock2'. Принимает два параметра 'MAKEWORD' - версия 'WinSock' (в этом случае версия 2.1), вторым параметром принимает указатель на структуру 'WSADATA', которая будет заполнена информацией о версии 'Winsock' после вызова функции.

### Анализ функции `socket`

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
```

Функция создает сокет, в качестве аргументов которого переданы параметры:

- **AF\_INET** — использование адресов типа IPv4;
- **SOCK\_STREAM** — использование протокола TCP;
- **0** — означает использование протокола по умолчанию для указанных выше параметров, в данном случае для TCP/IP это будет **IPPROTO\_TCP**.

### Анализ инициализации `sockaddr_in`

- `server_addr.sin_family = AF_INET` — задает семейство адресов (Address Family) для сокета. В данном случае это **AF\_INET**, что соответствует IPv4;
- `server_addr.sin_port = htons(DEFAULT_PORT)` — задает номер порта для сокета. Функция `htons()` переводит номер порта в сетевой порядок байтов (Network Byte Order);
- `server_addr.sin_addr.s_addr = INADDR_ANY` — задает IP-адрес для сокета. В данном случае используется специальное значение **INADDR\_ANY**, которое означает, что сокет будет связан с любым доступным IP-адресом на устройстве.

### 2.1.3 Выделение потока под клиента

```
while (true) {
    addr_size = sizeof(client_addr);
    client_socket = accept(server_socket,
        (sockaddr*)&client_addr, &addr_size);
    std::thread A (handle_client, client_socket);
    A.detach();
}
```

Как можно видеть здесь возникает работа с многопоточным программированием, которое реализуется при помощи встроенного в язык инструментария **std::thread**. Многопоточное программирование реализовано с целью работы с каждым клиентом отдельно, так как это один из самых простых случаев использования **std::thread**, то детально описывать методы не имеет смысла. В данном случае поток `A` выполняет задачу по выполнению функции `handle_client` (контекст функции будет рассмотрен в пункте 2.1.4). Отметим только то, что метод **std::thread.detach()**

запускает поток в фоновом режиме. Такая реализация позволяет работать функции без ожидания завершения других процессов. Это полезно, так как необходимо выполнять длительную операцию, не блокируя главный поток приложения.

## 2.1.4 Обработка клиента

Далее мы рассмотрим функцию, которая работает с клиентами сервера, ее код приведен ниже:

```
std::vector<int> clients;
void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    int read_size;
    clients.push_back(client_socket);
    std::cout << "Client " << client_socket << " connected." << std::endl;
    send(client_socket, buffer, BUFFER_SIZE, 0);
    while (((read_size = recv(client_socket, buffer, sizeof(buffer), 0)) > 0)) {
        buffer[read_size] = '\0';
        std::cout << "Received message from client " <<
            client_socket << ": " << buffer << std::endl;
        if (connection_close(buffer)) {
            closesocket(client_socket);
            std::cout << "Client " << client_socket << "
                disconnected." << std::endl;
        }
        for (int i = 0; i < clients.size(); i++) {
            if (clients[i] != client_socket) {
                send(clients[i], buffer, strlen(buffer), 0);
            }
        }
    }
}
```

Данный код представляет собой функцию обработки клиентского соединения для многопользовательского чата, который отправляет сообщения от одного клиента всем остальным клиентам.

В начале функции клиентский сокет добавляется в вектор 'clients', который содержит дескрипторы всех подключенных клиентов. Затем сервер отправляет клиенту пустой буфер, чтобы убедиться в том, что соединение работает.

Далее сервер начинает бесконечный цикл приема и отправки данных от клиента/клиенту. Для приема данных используется функция 'recv', которая читает данные из клиентского сокета и помещает их в буфер. Если количество прочитанных байт больше нуля, то сервер отправляет принятые данные всем остальным клиентам, кроме отправителя.

Если в принятых данных содержится команда на закрытие соединения, то сервер удаляет клиента из вектора 'clients', закрывает его сокет и завершает работу с ним.

Функция `handle_client` вызывается из главного цикла сервера для каждого нового клиентского соединения.

## 2.1.5 Пример работы

Полный код программы:

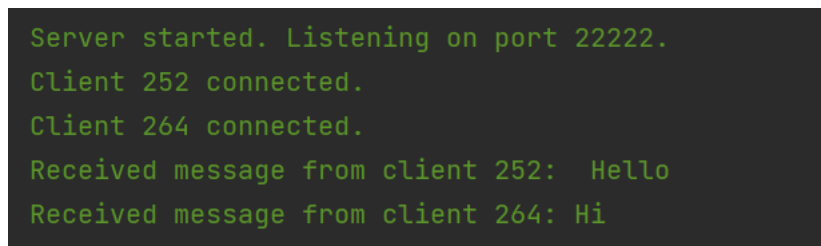
```
#include "chat.h"
#define DEFAULT_PORT 22222
#define ERROR_S "SERVER ERROR: "
#define CONNECTION_BREAK_SYMBOL '*'
#define BUFFER_SIZE 1024
#define MAX_CLIENTS 10
void cleanup() {
    WSACleanup();
}
std::vector<int> clients;
void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    int read_size;
    clients.push_back(client_socket);
    std::cout << "Client " << client_socket << " connected." << std::endl;
    send(client_socket, buffer, BUFFER_SIZE, 0);
    while (((read_size = recv(client_socket, buffer, sizeof(buffer), 0)) > 0)) {
        buffer[read_size] = '\0';
        std::cout << "Received message from client " << client_socket << ": " << buffer << std::endl;
        if (connection_close(buffer)) {
            closesocket(client_socket);
            std::cout << "Client " << client_socket << " disconnected." << std::endl;
        }
        for (int i = 0; i < clients.size(); i++) {
            if (clients[i] != client_socket) {
                send(clients[i], buffer, strlen(buffer), 0);
            }
        }
    }
}
int main() {
    int server_socket, client_socket;
    WSADATA data;
    if(0 != WSAStartup(MAKEWORD(2,1), &data)) return 101;
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
    SOCKADDR_IN server_addr, client_addr;
    socklen_t addr_size;
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        std::cerr << ERROR_S << "Error creating server socket." << std::endl;
        return -1;
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(DEFAULT_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(server_socket, (sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        std::cerr << ERROR_S << "Error binding server socket." << std::endl;
        return -1;
    }
    if (listen(server_socket, MAX_CLIENTS) == -1) {
        std::cerr << ERROR_S << "Error listening on server socket." << std::endl;
        return -1;
    }
}
```

```

std::cout << "Server started. Listening on port " <<
    DEFAULT_PORT << "." << std::endl;
std::atexit(cleanup);
while (true) {
    addr_size = sizeof(client_addr);
    client_socket = accept(server_socket, (sockaddr*)&
        client_addr, &addr_size);
    if (client_socket == -1) {
        std::cerr << ERROR_S << "Error accepting client socket."
            << std::endl;
        continue;
    }
    std::thread A (handle_client, client_socket);
    A.detach();
}
}

```

Скриншоты работы программы представлены на рисунке 1.



```

Server started. Listening on port 22222.
Client 252 connected.
Client 264 connected.
Received message from client 252: Hello
Received message from client 264: Hi

```

Рис. 1: Пример работы сервера

## 2.2 Код для клиента

Организация клиентского кода почти идентична той, что была для сервера, поэтому большая часть анализа кода будет пропущена за исключением пары моментов.

### 2.2.1 Чтение с сервера

```
void receive_messages(int client_socket) {
    char buffer[BUFFER_SIZE];
    int read_size;
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    while ((read_size = recv(client_socket, buffer, sizeof(buffer),
        0)) > 0) {
        buffer[read_size] = '\0';
        std::cout << "\nReceived message: " << buffer << std::endl;
    }
}
```

### 2.2.2 Ошибка MinGW

При настройке структуры 'SOCKADDR' рекомендуется использовать:  
`InetPtonA(AF_INET, SERVER_IP, &server_address.sin_addr);`

Путем множества проверок было доказано, что компиляторы MINGW/Ninja не поддерживают этот метод библиотеки 'WinSock2'. Было принято решение о замене `InetPtonA` на несколько аналогичных функций в совокупности выполняющие ту же самую функцию:

```
server_address.sin_addr.s_addr = inet_addr(SERVER_IP);
server_address.sin_family = AF_INET;
server_address.sin_port = ntohs(DEFAULT_PORT);
```

### 2.2.3 Пример работы

Полный код программы:

```
#include "..\..\SERVER_FOR_CLIENTS\chat.h"
#define DEFAULT_PORT 22222
#define ERROR_S "CLIENT ERROR: "
#define CONNECTION_BREAK_SYMBOL '*'
#define SERVER_IP "192.168.0.103"
#define BUFFER_SIZE 1024
void receive_messages(int client_socket) {
    char buffer[BUFFER_SIZE];
    int read_size;
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    while ((read_size = recv(client_socket, buffer, sizeof(buffer),
        0)) > 0) {
        buffer[read_size] = '\0';
        std::cout << "\nReceived message: " << buffer << std::endl;
    }
}
```

```

    }
}
int main() char const* argv[]) {
    int client;
    WSADATA data;
    GET_CONSOLE_NORMAL();
    SET_CONSOLE_GREEN();
    if (0 != WSASStartup(MAKEWORD(2, 1), &data)) return 101;
    SOCKADDR_IN server_address;
    client = socket(AF_INET, SOCK_STREAM, 0);
    if (client == -1) {
        SET_CONSOLE_RED();
        std::cerr << ERROR_S << "establishing socket error" << std
            ::endl;
        std::cerr << "Error " << strerror(errno) << std::endl;
        exit(EXIT_FAILURE);
    }
    server_address.sin_addr.s_addr = inet_addr(SERVER_IP);
    server_address.sin_family = AF_INET;
    server_address.sin_port = ntohs(DEFAULT_PORT);
    std::cout << "<=> Client socket created\n";
    int ret = connect(client, (sockaddr*)&server_address, sizeof(
        sockaddr));
    if (ret != 0) {
        SET_CONSOLE_RED();
        std::cerr << "<=> Connection to " << inet_ntoa(
            server_address.sin_addr)
                << " with port: " << DEFAULT_PORT <<
                " FAILED" << std::endl;

        return 101;
    }
    std::cout << "<=> Connection to " << inet_ntoa(server_address.
        sin_addr)
            << " with port: " << DEFAULT_PORT << std::endl;
    char buffer[BUFFER_SIZE];
    std::cout << "Waiting for server confirmation...";
    recv(client, buffer, BUFFER_SIZE, 0);
    std::cout << "<=> Connection established.\n" <<
        "Enter " << CONNECTION_BREAK_SYMBOL << " for disconnection\
        n";
    GET_CONSOLE_NORMAL();
    std::thread read(receive_messages, client);
    while (true) {
        std::this_thread::sleep_for(std::chrono::milliseconds(200))
            ;
        std::cout << "Your message: ";
        std::cin.getline(buffer, BUFFER_SIZE);
        send(client, buffer, strlen(buffer), 0);
        if (connection_close(buffer))
            break;
    }
    read.join();
    closesocket(client);
    return 0;
}

```

Как можно видеть из кода. Данные считываются в отдельном потоке. Также стоит заметить, что здесь использована задержка потока на 200 миллисекунд — это было сделано с целью использования разнообразного инструментария языка C++.

Скриншоты работы программы представлены на рисунке 2.

```
=> Client socket created
=> Connection to 127.0.0.1 with port: 22222
Waiting for server confirmation...=> Connection established.
Enter * for disconnection
Your message Hello
Received message: Hello, my name Maxim

Received message: What's up?
Your message i'm good
Your message
```

Рис. 2: Пример работы клиента



## Заключение

Работа с серверным чатом на C++ — это отличный способ познакомиться с основами сетевого программирования и создания многопоточных приложений. В процессе работы были изучены основные функции библиотеки 'Winsock2', включая создание и настройку сокетов, привязку сокета к адресу, прием и отправку данных, а также управление потоками с помощью функций 'detach'.

Был создан серверный чат, который позволяет подключаться клиентам к серверу и обмениваться сообщениями между собой в режиме реального времени. Было реализовано добавление клиентов в список при подключении, отправка сообщений всем клиентам, кроме отправителя, а также удаление клиента из списка при отключении.

При работе с серверным чатом необходимо учитывать некоторые важные моменты, такие как обработка ошибок, установление соединения и правильное управление потоками для избежания блокировок и ошибок сегментации. Кроме того, важно следить за безопасностью приложения и защитой от атак, таких как брутфорс или DoS.

В целом, работа с серверным чатом на C++ представляет собой интересный и познавательный опыт для разработчика, который может быть полезен при работе с другими сетевыми приложениями.

## Список литературы

- [1] Michael J. Donahoo, *TCP/IP Sockets in C: Practical Guide for Programmers*.  
Kenneth L. Calvert, Massachusetts, 1st Edition, 2000.
- [2] *Beej's Guide to Network Programming*.