Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе №8

на тему

**ИНТЕРФЕЙС СОКЕТОВ И ОСНОВЫ СЕТЕВОГО ПРОГРАММИРОВАНИЯ (WINDOWS). ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ ЧЕРЕЗ СЕТЬ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА СОКЕТОВ. РЕАЛИЗАЦИЯ СЕТЕВЫХ ПРОТОКОЛОВ: СОБСТВЕННЫХ ИЛИ СТАНДАРТНЫХ**

Студент О. Л. Дайнович

Преподаватель Н. Ю. Гриценко

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc151690205)

[2 Теоретические сведения 4](#_Toc151690206)

[3 Описание функций программы 6](#_Toc151690207)

[Заключение 9](#_Toc151690208)

[Список использованных источников 10](#_Toc151690209)

[Приложение А (обязательное) Листинг кода 11](#_Toc151690210)

# **1 ЦЕЛЬ РАБОТЫ**

Целью выполнения лабораторной работы является разработка клиент-серверного приложения для обмена текстовыми сообщениями с использованием TCP сокетов.

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Обмен данными, ориентированный на соединения, может использовать надежную связь, для обеспечения которой протокол уровня 4 посылает подтверждения о получении данных и запрашивает повторную передачу, если данные не получены или искажены. Протокол TCP использует именно такую надежную связь. TCP используется в таких прикладных протоколах, как HTTP, FTP, SMTP и Telnet. [1]

Протокол TCP требует, чтобы перед отправкой сообщения было открыто соединение. Серверное приложение должно выполнить так называемое пассивное открытие (passive open), чтобы создать соединение с известным номером порта, и, вместо того чтобы отправлять вызов в сеть, сервер переходит в ожидание поступления входящих запросов. Клиентское приложение должно выполнить активное открытие (active open), отправив серверному приложению синхронизирующий порядковый номер (SYN), идентифицирующий соединение. Клиентское приложение может использовать динамический номер порта в качестве локального порта.

Сервер должен отправить клиенту подтверждение (ACK) вместе с порядковым номером (SYN) сервера. В свою очередь клиент отвечает АСК, и соединение устанавливается.

После этого может начаться процесс отправки и получения сообщений. При получении сообщения в ответ всегда отправляется сообщение АСК. Если до получения АСК отправителем истекает тайм-аут, сообщение помещается в очередь на повторную передачу.

TCP — это сложный, требующий больших затрат времени протокол, что объясняется его механизмом установления соединения, но он берет на себя заботу о гарантированной доставке пакетов, избавляя нас от необходимости включать эту функциональную возможность в прикладной протокол.

Протокол TCP имеет встроенную возможность надежной доставки. Если сообщение не отправлено корректно, мы получим сообщение об ошибке. Протокол TCP определен в RFC 793.

В отличие от TCP UDP — очень быстрый протокол, поскольку в нем определен самый минимальный механизм, необходимый для передачи данных. Конечно, он имеет некоторые недостатки. Сообщения поступают в любом порядке, и то, которое отправлено первым, может быть получено последним. Доставка сообщений UDP вовсе не гарантируется, сообщение может потеряться, и могут быть получены две копии одного и того же сообщения. Последний случай возникает, если для отправки сообщений в один адрес использовать два разных маршрута.

UDP не требует открывать соединение, и данные могут быть отправлены сразу же, как только они подготовлены. UDP не отправляет подтверждающие сообщения, поэтому данные могут быть получены или потеряны. Если при использовании UDP требуется надежная передача данных, ее следует реализовать в протоколе более высокого уровня. [2]

Так в чем же преимущества UDP, зачем может понадобиться такой ненадежный протокол? Чтобы понять причину использования UDP, нужно различать однонаправленную передачу, широковещательную передачу и групповую рассылку.

Однонаправленное (unicast) сообщение отправляется из одного узла только в один другой узел. Это также называется связью "точка-точка". Протокол TCP поддерживает лишь однонаправленную связь. Если серверу нужно с помощью TCP взаимодействовать с несколькими клиентами, каждый клиент должен установить соединение, поскольку сообщения могут отправляться только одиночным узлам.

Широковещательная передача (broadcast) означает, что сообщение отправляется всем узлам сети. Групповая рассылка (multicast) - это промежуточный механизм: сообщения отправляются выбранным группам узлов.

UDP может использоваться для однонаправленной связи, если требуется быстрая передача, например для доставки мультимедийных данных, но главные преимущества UDP касаются широковещательной передачи и групповой рассылки. [3]

Обычно, когда мы отправляем широковещательные или групповые сообщения, не нужно получать подтверждения из каждого узла, поскольку тогда сервер будет наводнен подтверждениями, а загрузка сети возрастет слишком сильно. Примером широковещательной передачи является служба времени. Сервер времени отправляет широковещательное сообщение, содержащее текущее время, и любой хост, если пожелает, может синхронизировать свое время с временем из широковещательного сообщения.

# **3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ**

В ходе выполнения лабораторной работы было разработано оконное приложение для обмена текстовыми сообщениями по локальной сети с использованием сокетов.

Были созданы приложения с реализацией сервера и клиента. После запуска сервера он принимает подключение от клиента (рисунок 1).

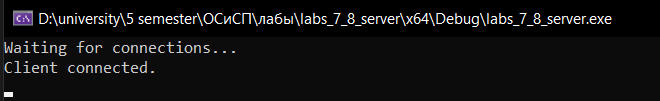


Рисунок 1 – Подключение клиента к серверу

В случае неудачного подключения к серверу выводится сообщение об ошибке (рисунок 2).

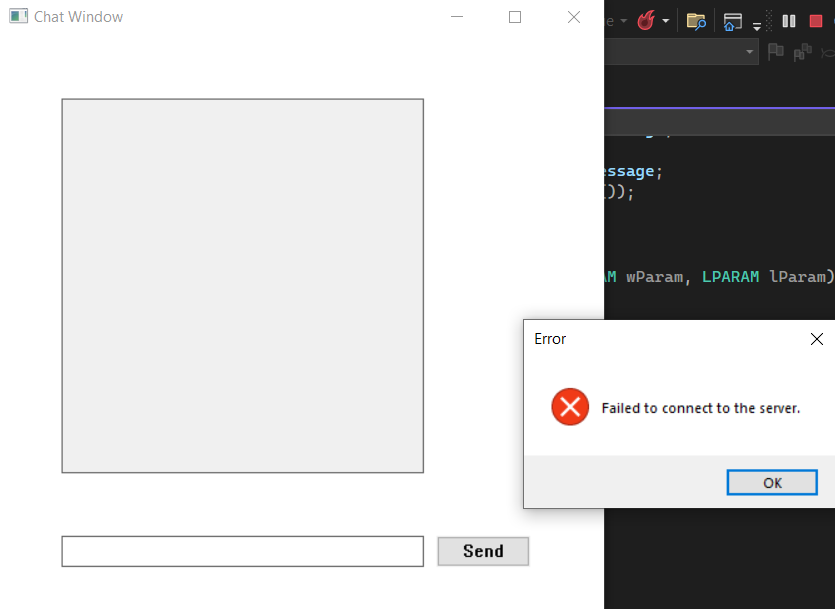


Рисунок 2 – Сообщение об ошибке подключения

После подключения клиента к серверу клиент и сервер могут обмениваться сообщениями, которые выводятся в окне приложения клиента (рисунок 3).

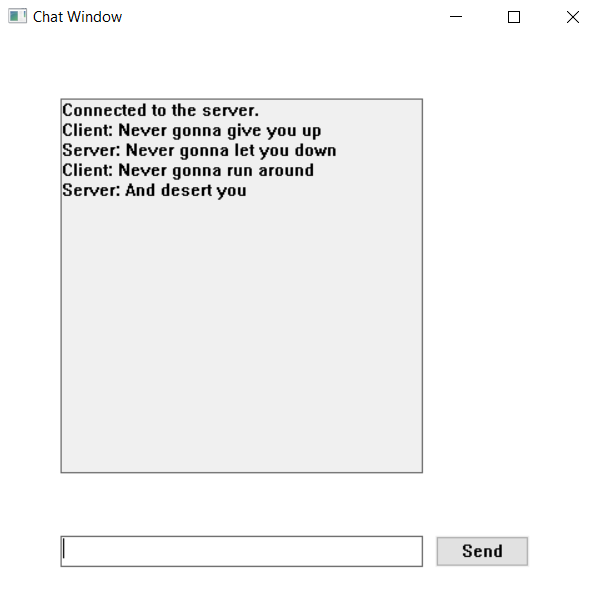


Рисунок 3 – Вывод сообщений на стороне клиента

Сообщения выводятся также и на стороне сервера (рисунок 4).

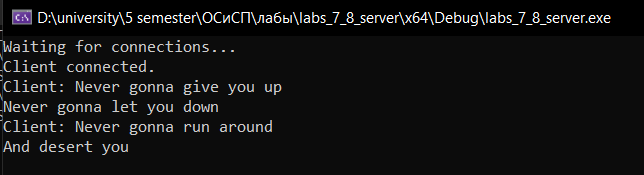


Рисунок 4 – Вывод сообщений на стороне сервера

После отключения клиента от сервера сервер получает соответствующее сообщение (рисунок 5).

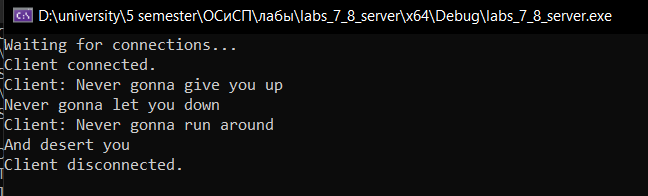


Рисунок 5 – Сообщение об отключении клиента от сервера

# **ЗАКЛЮЧЕНИЕ**

В результате лабораторной работы были изучены принципы работы с сокетами в Win32 API. Было создано клиент-серверное приложение для обмена текстовыми сообщениями с использованием TCP сокетов.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Microsoft Learn [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/learnwin32/.

[2] Начало работы с классическими приложениями для Windows, которые используют API Win32 [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/desktop-programming.

[3] Руководство по стеку протоколов TCP/IP [Электронный ресурс]. – Режим доступа: https://selectel.ru/blog/tcp-ip-for-beginners/

# **ПРИЛОЖЕНИЕ А (обязательное) Листинг кода**

Листинг 1 – Файл lab\_7\_8\_server.cpp

#include <iostream>

#include <winsock2.h>

#include <thread>

#include <string>

#pragma comment(lib, "WS2\_32.lib")

// Буфер для чтения и записи данных

const int BUFFER\_SIZE = 4096;

char buffer[BUFFER\_SIZE];

void ReceiveMessages(SOCKET clientSocket) {

while (true) {

int bytesRead = recv(clientSocket, buffer, BUFFER\_SIZE, 0);

if (bytesRead <= 0) {

std::cout << "Client disconnected." << std::endl;

break;

}

std::cout << "Client: " << buffer << std::endl;

// Очистка буфера

memset(buffer, 0, BUFFER\_SIZE);

}

}

void SendMessages(SOCKET clientSocket) {

while (true) {

std::string message;

std::getline(std::cin, message);

if (send(clientSocket, message.c\_str(), message.length(), 0) == SOCKET\_ERROR) {

std::cout << "Failed to send data to the client." << std::endl;

break;

}

}

}

int main() {

WSADATA wsaData;

SOCKET serverSocket, clientSocket;

struct sockaddr\_in serverAddress, clientAddress;

// Инициализация Winsock

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

std::cout << "Failed to initialize socket." << std::endl;

return 1;

}

// Создание сокета

if ((serverSocket = socket(AF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET) {

std::cout << "Failed to create socket." << std::endl;

WSACleanup();

return 1;

}

// Настройка адреса сервера

serverAddress.sin\_family = AF\_INET;

serverAddress.sin\_port = htons(12345); // Порт сервера

serverAddress.sin\_addr.s\_addr = INADDR\_ANY; // Принимать подключения на все IP-адреса

// Привязка сокета к адресу сервера

if (bind(serverSocket, (struct sockaddr\*)&serverAddress, sizeof(serverAddress)) == SOCKET\_ERROR) {

std::cout << "Failed to bind socket." << std::endl;

closesocket(serverSocket);

WSACleanup();

return 1;

}

// Прослушивание входящих подключений

if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR) {

std::cout << "Failed to listen on socket." << std::endl;

closesocket(serverSocket);

WSACleanup();

return 1;

}

std::cout << "Waiting for connections..." << std::endl;

// Принятие входящих подключений

int clientAddressSize = sizeof(clientAddress);

clientSocket = accept(serverSocket, (struct sockaddr\*)&clientAddress, &clientAddressSize);

if (clientSocket == INVALID\_SOCKET) {

std::cout << "Failed to accept client connection." << std::endl;

closesocket(serverSocket);

WSACleanup();

return 1;

}

std::cout << "Client connected." << std::endl;

std::thread receiveThread(ReceiveMessages, clientSocket);

std::thread sendThread(SendMessages, clientSocket);

receiveThread.join();

sendThread.join();

closesocket(clientSocket);

closesocket(serverSocket);

WSACleanup();

return 0;

}

Листинг 2 – Файл lab\_7\_8\_client.cpp

#include <iostream>

#include <winsock2.h>

#include <thread>

#include <string>

#include <Windows.h>

#include <CommCtrl.h>

#pragma comment(lib, "Comctl32.lib")

#pragma comment(linker, "\"/manifestdependency:type='win32' \ name = 'Microsoft.Windows.Common-Controls' version = '6.0.0.0' \

processorArchitecture = '\*' publicKeyToken = '6595b64144ccf1df' language = '\*'\"")

#pragma once

#pragma warning(disable: 4996)

#pragma comment(lib, "WS2\_32.lib")

#define ID\_BUTTON\_SEND 1000

#define EDIT\_MESSAGE 1001

const int BUFFER\_SIZE = 4096;

char buffer[BUFFER\_SIZE];

HWND hwnd; // Handle to the main window

HWND hShowMessage;

HWND hWriteMessage;

HWND hSendButton;

SOCKET clientSocket;

const UINT WM\_SOCKET\_MESSAGE = WM\_USER + 1; // Custom message for socket events

void ReceiveMessages(SOCKET clientSocket) {

while (true) {

int bytesRead = recv(clientSocket, buffer, BUFFER\_SIZE, 0);

if (bytesRead <= 0) {

PostMessage(hwnd, WM\_SOCKET\_MESSAGE, 0, 0);

break;

}

std::string message = "\nServer: ";

message += buffer;

int bufferSize = GetWindowTextLength(hShowMessage) + 1;

std::wstring history(bufferSize, L'\0');

GetWindowText(hShowMessage, &history[0], bufferSize);

size\_t buffer\_Size = history.length() + 1;

char\* nbuffer = new char[buffer\_Size];

size\_t convertedChars = 0;

wcstombs\_s(&convertedChars, nbuffer, buffer\_Size, history.c\_str(), \_TRUNCATE);

std::string str\_history(nbuffer);

delete[] nbuffer;

std::string new\_history = str\_history + message;

SetWindowTextA(hShowMessage, new\_history.c\_str());

memset(buffer, 0, BUFFER\_SIZE);

}

}

void SendMessages(SOCKET clientSocket, HWND hwnd) {

WCHAR message[256];

GetDlgItemText(hwnd, EDIT\_MESSAGE, message, 256);

int size = WideCharToMultiByte(CP\_UTF8, 0, message, -1, nullptr, 0, nullptr, nullptr);

char\* c\_message = new char[size];

WideCharToMultiByte(CP\_UTF8, 0, message, -1, c\_message, size, nullptr, nullptr);

if (send(clientSocket, c\_message, size, 0) == SOCKET\_ERROR) {

MessageBox(hwnd, L"Failed to send data to the server", L"Error", MB\_OK | MB\_ICONERROR);

}

else {

int bufferSize = GetWindowTextLength(hShowMessage) + 1;

std::wstring history(bufferSize, L'\0');

GetWindowText(hShowMessage, &history[0], bufferSize);

size\_t buffer\_Size = history.length() + 1;

char\* buffer = new char[buffer\_Size];

size\_t convertedChars = 0;

wcstombs\_s(&convertedChars, buffer, buffer\_Size, history.c\_str(), \_TRUNCATE);

std::string str\_history(buffer);

delete[] buffer;

std::string str\_message;

str\_message.assign(c\_message);

std::string full\_message = "\nClient: " + str\_message;

std::string new\_history = str\_history + full\_message;

SetWindowTextA(hShowMessage, new\_history.c\_str());

}

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {

case WM\_SOCKET\_MESSAGE:

MessageBox(hwnd, L"Connection closed by the server.", L"Message", MB\_OK | MB\_ICONERROR);

break;

case WM\_COMMAND:

{

switch (LOWORD(wParam))

{

case ID\_BUTTON\_SEND:

SendMessages(clientSocket, hwnd);

break;

}

}

break;

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

}

return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow) {

WSADATA wsaData;

struct sockaddr\_in serverAddress;

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

MessageBox(NULL, L"Failed to initialize socket.", L"Error", MB\_OK | MB\_ICONERROR);

return 1;

}

const wchar\_t className[] = L"MyWindowClass";

const wchar\_t windowTitle[] = L"Chat Window";

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

wc.lpszClassName = className;

RegisterClass(&wc);

hwnd = CreateWindowEx(

0,

className,

windowTitle,

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, 500, 500,

NULL,

NULL,

hInstance,

NULL

);

if (hwnd == NULL) {

MessageBox(NULL, L"Failed to create the window.", L"Error", MB\_OK | MB\_ICONERROR);

WSACleanup();

return 1;

}

hShowMessage = CreateWindow(

L"STATIC", L"",

WS\_VISIBLE | WS\_CHILD | WS\_BORDER,

50, 50, 290, 300,

hwnd, NULL, hInstance, NULL);

hWriteMessage = CreateWindow(

L"EDIT", L"",

WS\_VISIBLE | WS\_CHILD | WS\_BORDER,

50, 400, 290, 25,

hwnd, reinterpret\_cast<HMENU>(EDIT\_MESSAGE), hInstance, NULL);

hSendButton = CreateWindow(L"BUTTON", L"Send", WS\_VISIBLE | WS\_CHILD, 350, 400, 75, 25, hwnd, (HMENU)ID\_BUTTON\_SEND, hInstance, NULL);

ShowWindow(hwnd, nCmdShow);

if ((clientSocket = socket(AF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET) {

MessageBox(hwnd, L"Failed to create socket.", L"Error", MB\_OK | MB\_ICONERROR);

WSACleanup();

return 1;

}

serverAddress.sin\_family = AF\_INET;

serverAddress.sin\_port = htons(12345);

serverAddress.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if (connect(clientSocket, (struct sockaddr\*)&serverAddress, sizeof(serverAddress)) == SOCKET\_ERROR) {

MessageBox(hwnd, L"Failed to connect to the server.", L"Error", MB\_OK | MB\_ICONERROR);

closesocket(clientSocket);

WSACleanup();

return 1;

}

SetWindowTextA(hShowMessage, "Connected to the server.");

std::thread receiveThread(ReceiveMessages, clientSocket);

//std::thread sendThread(SendMessages, clientSocket);

MSG msg;

while (GetMessage(&msg, NULL, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

receiveThread.join();

//sendThread.join();

closesocket(clientSocket);

WSACleanup();

return 0;

}