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

ОТЧЁТ

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.

Выполнил студент гр.153502 Толстой Д.В.

Проверил ассистент кафедры информатики Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Формулировка задачи………..……………………………………………….](#_gjdgxs)... [3](#_gjdgxs)

[2 Теоретические сведения……………………………………………………….. 4](#_30j0zll)

[3 Описание функций программы………………………………………………... 5](#_1fob9te)

[Список использованных источников…………………………………………….](#_2et92p0) 6

[Приложение А (обязательное) Листинг кода…………………………………..](#_tyjcwt) 7

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

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

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*TCP* (англ. *Transmission* *Control* *Protocol* — протокол управления передачей) — один из основных протоколов передачи данных интернета. Предназначен для управления передачей данных интернета. Пакеты в *TCP* называются сегментами.

В стеке протоколов *TCP*/*IP* выполняет функции транспортного уровня модели *OSI*.

Механизм *TCP* предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым (в отличие от *UDP*) целостность передаваемых данных и уведомление отправителя о результатах передачи.

Реализации *TCP* обычно встроены в ядра ОС. Существуют реализации *TCP*, работающие в пространстве пользователя.

Когда осуществляется передача от компьютера к компьютеру через Интернет, *TCP* работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. *TCP* осуществляет надёжную передачу потока байтов от одного процесса к другому [1].

# **3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ**

Сначала запускается серверное приложение, которое в отдельном потоке начинает прослушивание сообщений, и при получении таковых, происходит отправка этого сообщения всем подключенным пользователям (рисунок 1).



Рисунок 1 – Консоль северного приложения

Все запускаемые клиентские приложения подключаются к серверу и начинают прослушивание сервера, и при получении сообщения выводит в специальное окно для вывода сообщений (рисунок 2).



Рисунок 2 – Интерфейс клиентского приложения

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] TCP [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://ru.wikipedia.org/wiki/TCP>

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

Листинг 1 – Файл server.cpp

#include <iostream>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <string>

#pragma comment(lib, "Ws2\_32.lib")

#define DEFAULT\_PORT "27015"

#define MAX\_CLIENTS 10

SOCKET clientSockets[MAX\_CLIENTS];

int numClients = 0;

void SendToAllClients(const char\* message, int senderIndex) {

for (int i = 0; i < numClients; ++i) {

if (i == senderIndex) {

std::string new\_message("User ");

new\_message += std::to\_string(senderIndex) + " (You): ";

new\_message += message;

send(clientSockets[i], new\_message.c\_str(), strlen(new\_message.c\_str()), 0);

}

else

{

std::string new\_message("User ");

new\_message += std::to\_string(senderIndex) + ": ";

new\_message += message;

send(clientSockets[i], new\_message.c\_str(), strlen(new\_message.c\_str()), 0);

}

}

}

DWORD WINAPI ClientThread(LPVOID param) {

int clientIndex = (int)param;

char recvbuf[1024];

while (true) {

int recvResult = recv(clientSockets[clientIndex], recvbuf, sizeof(recvbuf), 0);

if (recvResult > 0) {

recvbuf[recvResult] = '\0';

std::cout << "Received: " << recvbuf << std::endl;

SendToAllClients(recvbuf, clientIndex);

}

else if (recvResult == 0) {

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

closesocket(clientSockets[clientIndex]);

break;

}

else {

std::cout << "recv failed with error: " << WSAGetLastError() << std::endl;

closesocket(clientSockets[clientIndex]);

break;

}

}

return 0;

}

int main() {

WSADATA wsaData;

int iResult;

SOCKET listenSocket = INVALID\_SOCKET;

addrinfo\* result = NULL;

addrinfo hints;

iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

std::cout << "WSAStartup failed: " << iResult << std::endl;

return 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;

iResult = getaddrinfo(NULL, DEFAULT\_PORT, &hints, &result);

if (iResult != 0) {

std::cout << "getaddrinfo failed: " << iResult << std::endl;

WSACleanup();

return 1;

}

listenSocket = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

if (listenSocket == INVALID\_SOCKET) {

std::cout << "Error at socket(): " << WSAGetLastError() << std::endl;

freeaddrinfo(result);

WSACleanup();

return 1;

}

iResult = bind(listenSocket, result->ai\_addr, (int)result->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

std::cout << "bind failed: " << WSAGetLastError() << std::endl;

freeaddrinfo(result);

closesocket(listenSocket);

WSACleanup();

return 1;

}

freeaddrinfo(result);

iResult = listen(listenSocket, SOMAXCONN);

if (iResult == SOCKET\_ERROR) {

std::cout << "listen failed: " << WSAGetLastError() << std::endl;

closesocket(listenSocket);

WSACleanup();

return 1;

}

std::cout << "Server started, waiting for incoming connections..." << std::endl;

while (true) {

SOCKET clientSocket = accept(listenSocket, NULL, NULL);

if (clientSocket == INVALID\_SOCKET) {

std::cout << "accept failed: " << WSAGetLastError() << std::endl;

closesocket(listenSocket);

WSACleanup();

return 1;

}

if (numClients < MAX\_CLIENTS) {

clientSockets[numClients++] = clientSocket;

CreateThread(NULL, 0, ClientThread, (LPVOID)(numClients - 1), 0, NULL);

}

else {

std::cout << "Too many clients. Connection rejected." << std::endl;

closesocket(clientSocket);

}

}

closesocket(listenSocket);

WSACleanup();

return 0;

}

Листинг 2 – Файл client.cpp

#include <iostream>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#include <string>

#pragma comment(lib, "Ws2\_32.lib")

#define DEFAULT\_PORT "27015"

#define DEFAULT\_BUFLEN 1024

HWND g\_hwnd;

static HWND hwndEditInput;

static HWND hwndListBox;

static HWND hwndButtonSend;

SOCKET g\_socket;

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

DWORD WINAPI ReceiveThread(LPVOID param) {

char recvbuf[DEFAULT\_BUFLEN];

while (true) {

int recvResult = recv(g\_socket, recvbuf, DEFAULT\_BUFLEN, 0);

if (recvResult > 0) {

recvbuf[recvResult] = '\0';

int len = MultiByteToWideChar(CP\_UTF8, 0, recvbuf, -1, NULL, 0);

wchar\_t\* w\_recvbuf = new wchar\_t[len];

MultiByteToWideChar(CP\_UTF8, 0, recvbuf, -1, w\_recvbuf, len);

// Получение текста из текстового поля

wchar\_t textBuffer[DEFAULT\_BUFLEN];

GetWindowText(hwndListBox, textBuffer, DEFAULT\_BUFLEN);

// Преобразование текста к объекту std::wstring

std::wstring currentText = textBuffer;

// Добавление строки к текущему тексту

std::wstring additionalText(w\_recvbuf);

currentText += additionalText;

SendMessage(hwndListBox, LB\_ADDSTRING, 0, reinterpret\_cast<LPARAM>(currentText.c\_str()));

delete[] w\_recvbuf;

}

else if (recvResult == 0) {

MessageBox(NULL, L"Connection closed", L"Error", MB\_OK | MB\_ICONERROR);

break;

}

else {

MessageBox(NULL, L"Receive failed", L"Error", MB\_OK | MB\_ICONERROR);

break;

}

}

return 0;

}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

WNDCLASSEX wc;

MSG msg;

HWND hwnd;

wc.cbSize = sizeof(WNDCLASSEX);

wc.style = 0;

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = hInstance;

wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW + 1);

wc.lpszMenuName = NULL;

wc.lpszClassName = L"WinSockClient";

wc.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

if (!RegisterClassEx(&wc)) {

MessageBox(NULL, L"Window Registration Failed!", L"Error!", MB\_ICONEXCLAMATION | MB\_OK);

return 0;

}

hwnd = CreateWindowExW(WS\_EX\_CLIENTEDGE, L"WinSockClient", L"WinSock Client", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT, 600, 600, NULL, NULL, hInstance, NULL);

if (hwnd == NULL) {

MessageBox(NULL, L"Window Creation Failed!", L"Error!", MB\_ICONEXCLAMATION | MB\_OK);

return 0;

}

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

WSADATA wsaData;

int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

MessageBox(NULL, L"WSAStartup failed", L"Error", MB\_OK | MB\_ICONERROR);

return 1;

}

struct addrinfo\* result = NULL, \* ptr = NULL, hints;

ZeroMemory(&hints, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

iResult = getaddrinfo("127.0.0.1", DEFAULT\_PORT, &hints, &result);

if (iResult != 0) {

MessageBox(NULL, L"getaddrinfo failed", L"Error", MB\_OK | MB\_ICONERROR);

WSACleanup();

return 1;

}

for (ptr = result; ptr != NULL; ptr = ptr->ai\_next) {

g\_socket = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);

if (g\_socket == INVALID\_SOCKET) {

MessageBox(NULL, L"Error creating socket", L"Error", MB\_OK | MB\_ICONERROR);

WSACleanup();

return 1;

}

iResult = connect(g\_socket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

closesocket(g\_socket);

g\_socket = INVALID\_SOCKET;

continue;

}

break;

}

freeaddrinfo(result);

if (g\_socket == INVALID\_SOCKET) {

MessageBox(NULL, L"Unable to connect to server!", L"Error", MB\_OK | MB\_ICONERROR);

WSACleanup();

return 1;

}

CreateThread(NULL, 0, ReceiveThread, NULL, 0, NULL);

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

closesocket(g\_socket);

WSACleanup();

return (int)msg.wParam;

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {

switch (msg) {

case WM\_CREATE:

hwndEditInput = CreateWindowEx(WS\_EX\_CLIENTEDGE, L"EDIT", L"", WS\_CHILD | WS\_VISIBLE | WS\_VSCROLL | ES\_AUTOVSCROLL | ES\_MULTILINE | ES\_AUTOHSCROLL, 50, 360, 300, 30, hwnd, NULL, NULL, NULL);

hwndListBox = CreateWindowEx( 0, L"LISTBOX", NULL, WS\_CHILD | WS\_VISIBLE | WS\_BORDER | WS\_VSCROLL | WS\_HSCROLL | LBS\_HASSTRINGS | LBS\_NOINTEGRALHEIGHT | LBS\_DISABLENOSCROLL, 50, 50, 400, 300, hwnd, NULL, NULL,NULL );

hwndButtonSend = CreateWindowEx(NULL, L"BUTTON", L"Send", WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON, 360, 360, 90, 30, hwnd, (HMENU)1, NULL, NULL);

break;

case WM\_COMMAND:

if (LOWORD(wParam) == 1) {

wchar\_t sendbuf[DEFAULT\_BUFLEN];

GetWindowText(hwndEditInput, sendbuf, DEFAULT\_BUFLEN);

int len = wcslen(sendbuf);

// Get the required length of the narrow string

int requiredSize = WideCharToMultiByte(CP\_UTF8, 0, sendbuf, -1, NULL, 0, NULL, NULL);

// Allocate memory for the narrow string

char\* str = new char[requiredSize];

// Convert the wide string to narrow string

WideCharToMultiByte(CP\_UTF8, 0, sendbuf, -1, str, requiredSize, NULL, NULL);

send(g\_socket, str, len, 0);

SetWindowText(hwndEditInput, L"");

}

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

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

}

return 0;

}