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 Миненков Я.А.

Проверил ассистент кафедры информатики Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВКА ЗАДАЧИ

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

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Протокол управления передачей (*TCP*) – это надежный, но сложный протокол транспортного уровня. *TCP* добавляет в *IP* надежность функции поддержки соединения.

TCP – это надежная служба доставки потока, гарантирующая доставку потока данных из одного хоста в другой без дублирования и потерь данных. Так как передача пакетов не является надежной, то применяется метод подтверждения приема с повторной передачей, гарантирующий надежную доставку пакетов. Для этого базового метода требуется, чтобы приемник отправлял сообщение с подтверждением при приеме данных.

Отправитель хранит запись каждого отправленного пакета и ожидает подтверждение перед отправкой следующего пакета. Также отправитель хранит таймер для каждого отправленного пакета и отправляет пакет повторно по истечении заданного времени. Таймер необходим в случае потери или повреждения пакета [1].

UDP (*User Datagram Protocol*), с другой стороны, представляет собой протокол без установления соединения, не гарантирующий надежную доставку данных. Он более быстрый и более легковесный, так как не включает механизмы обеспечения надежности, контроля ошибок и повторной отправки данных. *UDP* может быть полезен для приложений, где небольшие задержки важнее надежности, например, в потоковых медиа-приложениях, играх или в ситуациях, где потеря части данных не критична [2].

Для клиент-серверных приложений обмена текстовыми сообщениями, где важно, чтобы сообщения приходили целиком и в правильной последовательности, *TCP* предпочтительнее. Он обеспечивает надежную доставку данных и контролирует их порядок, что важно для обмена текстовыми сообщениями, чтобы избежать искажения информации или ее потери.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Для обмена сообщения используется приложение сервера и приложения клиентов с общим чатом(Рисунок 1).

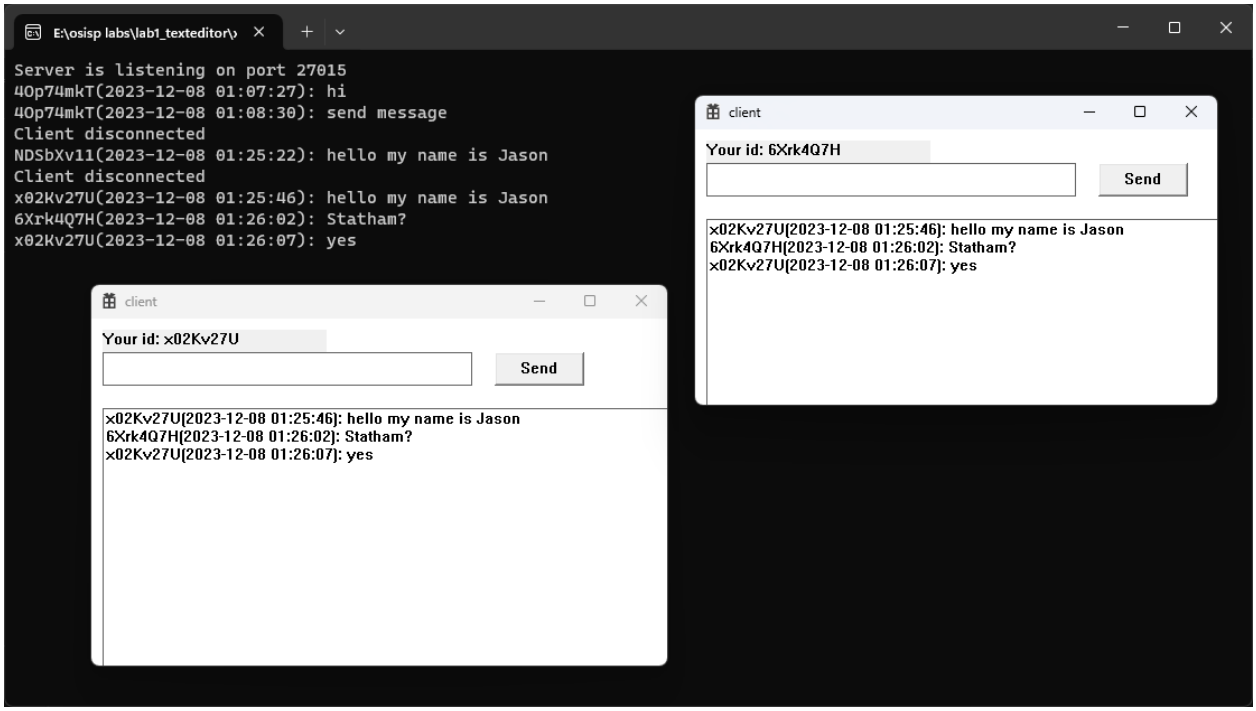

Рисунок 1 – Передача сообщений между пользователями

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Обзор сокетов – IBM Documentation [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.ibm.com/docs/ru/rtw/9.0.0?topic=transports-sockets-overview

[2] UDP — Википедия [Электронный ресурс]. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/UDP

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

Листинг 1 – Файл *server.cpp*

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <windows.h>
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

#pragma comment(lib, "ws2_32.lib")

const int DEFAULT_PORT = 27015;
const int BUFFER_SIZE = 1024;

std::vector<SOCKET> clients;
std::mutex clientsMutex;

void HandleClient(SOCKET clientSocket);

int main() {
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "Failed to initialize Winsock" << std::endl;
        return 1;
    }

    SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == INVALID_SOCKET) {
        std::cerr << "Failed to create socket" << std::endl;
        WSACleanup();
        return 1;
    }

    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(DEFAULT_PORT);
    serverAddress.sin_addr.s_addr = INADDR_ANY;

    if (bind(serverSocket, reinterpret_cast<sockaddr*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
        std::cerr << "Failed to bind socket" << std::endl;
        closesocket(serverSocket);
        WSACleanup();
        return 1;
    }

    if (listen(serverSocket, SOMAXCONN) == SOCKET_ERROR) {
        std::cerr << "Failed to listen on socket" << std::endl;
        closesocket(serverSocket);
        WSACleanup();
    }

    std::cout << "Server is listening on port " << DEFAULT_PORT << std::endl;

    while (true) {
        SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);
        if (clientSocket == INVALID_SOCKET) {
```

```
            std::cerr << "Failed to accept client connection" << std::endl;
            closesocket(serverSocket);
            WSACleanup();
            return 1;
        }

        std::lock_guard<std::mutex> lock(clientsMutex);
        clients.push_back(clientSocket);

        std::thread clientThread(HandleClient, clientSocket);
        clientThread.detach();
    }

    closesocket(serverSocket);
    WSACleanup();

    return 0;
}

void HandleClient(SOCKET clientSocket) {
    char buffer[BUFFER_SIZE];
    int bytesRead;

    do {
        bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
        if (bytesRead > 0) {
            buffer[bytesRead - 1] = '\0';
            std::cout << buffer << std::endl;

            // Instead of sending the response to the client who sent this message
            // we send it to all connected clients.
            std::lock_guard<std::mutex> lock(clientsMutex);
            for (SOCKET otherClient : clients) {
                send(otherClient, buffer, strlen(buffer), 0);
            }
        }
    } while (bytesRead > 0);

    std::cout << "Client disconnected" << std::endl;
    std::lock_guard<std::mutex> lock(clientsMutex);
    clients.erase(std::remove(clients.begin(), clients.end(), clientSocket),
clients.end());

    closesocket(clientSocket);
}
```

Листинг 2 – *client.exe*

```
#define _CRT_SECURE_NO_WARNINGS
#include "framework.h"
#include "client.h"
#include <winsock2.h>
#include <ws2tcpip.h>
#include <windows.h>
#include <iostream>
#include <thread>
#include <string>
#include <mutex>
#include <queue>
#include <locale>
#include <codecvt>
#include <random>
#include <locale.h>
```

```
#pragma comment(lib, "ws2_32.lib")

#define MAX_LOADSTRING 100
#define DEFAULT_PORT 27015
#define BUFFER_SIZE 1024
#define ID_EDIT_TEXTBOX 201
#define ID_SEND_BUTTON 202
#define ID_MESSAGES_LIST 203

SOCKET clientSocket;
char buffer[BUFFER_SIZE];
std::queue<std::string> messages;
std::mutex messagesMutex;
std::string id;

HWND hWndTextBox;
HWND hWndSendButton;
HWND hWndList;

HINSTANCE hInst;
WCHAR szTitle[MAX_LOADSTRING];
WCHAR szWindowClass[MAX_LOADSTRING];

ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE, int);
BOOL InitUIComponents(HWND);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);

bool SetupWinsock();
bool SetupClient();
void CleanupClient();
void HandleServerMessages();
std::string generateRandomID(int length);
std::string getCurrentDateTime();

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPWSTR    lpCmdLine,
    _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_CLIENT, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    id = generateRandomID(8);

    if (!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }

    if (!SetupWinsock()) {
        std::cerr << "Failed to initialize Winsock\n";
        return FALSE;
    }
    if (!SetupClient()) {
        std::cerr << "Failed to setup client\n";
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_CLIENT));
```

```
    MSG msg;

    // Main message loop:
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    CleanupClient();
    return (int)msg.wParam;
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance;

    HWND hWnd = CreateWindowW(
        szWindowClass,
        szTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0,
        CW_USEDEFAULT, 0,
        nullptr, nullptr,
        hInstance, nullptr
    );

    if (!hWnd)
    {
        return FALSE;
    }
    HMENU hMenu = CreateMenu();
    SetMenu(hWnd, hMenu);
    SetWindowPos(hWnd, 0, 0, 0, 450, 450, SWP_NOMOVE | SWP_NOZORDER);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    SetTimer(hWnd, 1, 200, NULL);

    return TRUE;
}

BOOL InitUIComponents(HWND hWnd) {
    std::wstring wideId(id.begin(), id.end());

    std::wstring idString = L"Your id: " + wideId;

    HWND hWndStatic = CreateWindow(
        TEXT("STATIC"),
        idString.c_str(),
        WS_CHILD | WS_VISIBLE,
        10, 10, 200, 20,
        hWnd,
        NULL,
        NULL,
        NULL
    );

    hWndTextBox = CreateWindow(TEXT("EDIT"), TEXT(""),
        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL,
```

```
        10, 30, 300, 30, hWnd,
        (HMENU)ID_EDIT_TEXTBOX, NULL, NULL);

    hWndSendButton = CreateWindow(TEXT("BUTTON"), TEXT("Send"),
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
        330, 30, 80, 30, hWnd,
        (HMENU)ID_SEND_BUTTON, NULL, NULL);

    hWndList = CreateWindow(
        TEXT("LISTBOX"), TEXT(""),
        WS_CHILD | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_HSCROLL | LBS_HASSTRINGS
| LBS_NOINTEGRALHEIGHT | LBS_DISABLENOSCROLL,
        10, 80, 400, 300,
        hWnd, (HMENU)ID_MESSAGES_LIST,
        NULL, NULL
    );

    if (!hWndTextBox || !hWndSendButton || !hWndList) {
        return FALSE;
    }

    return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
    {
        int wmId = LOWORD(wParam);
        switch (wmId)
        {
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        case ID_SEND_BUTTON:
        {
            TCHAR textBuffer[BUFFER_SIZE];
            GetWindowText(hWndTextBox, textBuffer, BUFFER_SIZE);
            char buffer[BUFFER_SIZE];
            size_t convertedChars = 0;


            wcstombs_s(&convertedChars, buffer, BUFFER_SIZE, textBuffer, _TRUNCATE);

            char tempBuffer[BUFFER_SIZE];
            strcpy(tempBuffer, buffer);

            std::string prefix = id + "(" + getCurrentDateTime() + "): ";

            strcpy(buffer, prefix.c_str());

            strcat(buffer, tempBuffer);

            OutputDebugString(LPCWSTR(buffer));

            int bytesSent = send(clientSocket, buffer, BUFFER_SIZE, 0);
            if (bytesSent == SOCKET_ERROR) {
                std::cerr << "Failed to send message" << std::endl;
                break;
            }

            SetWindowText(hWndTextBox, TEXT(""));
        }
```

```
            break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
            }
        }
        break;
        case WM_CREATE:
        {
            if (!InitUIComponents(hWnd)) {
                return -1;
            }
        }
        break;
        case WM_TIMER:
        {
            std::lock_guard<std::mutex> lock(messagesMutex);
            while (!messages.empty()) {
                std::string message = messages.front();
                std::wstring_convert<std::codecvt_utf8<wchar_t>> converter;
                std::wstring wide_str = converter.from_bytes(message);
                SendMessage(hWndList, LB_ADDSTRING, 0,
reinterpret_cast<LPARAM>(wide_str.c_str()));
                messages.pop();
            }
        }
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
        }


ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEXW wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance      = hInstance;
    wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_CLIENT));
    wcex.hCursor        = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = MAKEINTRESOURCEW(IDC_CLIENT);
    wcex.lpszClassName  = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassExW(&wcex);
}

bool SetupWinsock() {
    WSADATA wsaData;
    return WSAStartup(MAKEWORD(2, 2), &wsaData) == 0;
}

bool SetupClient() {

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

```
    if (clientSocket == INVALID_SOCKET) {
        MessageBox(NULL, L"Failed to create socket", L"Error", MB_ICONERROR);
        WSACleanup();
        return false;
    }

    // Установка адреса и порта
    sockaddr_in serverAddress{};
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(DEFAULT_PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &(serverAddress.sin_addr)) <= 0) {
        MessageBox(NULL, L"Invalid server address", L"Error", MB_ICONERROR);
        closesocket(clientSocket);
        WSACleanup();
        return false;
    }

    if (connect(clientSocket, reinterpret_cast<sockaddr*>(&serverAddress),
sizeof(serverAddress)) == SOCKET_ERROR) {
        MessageBox(NULL, L"Failed to connect to server", L"Error", MB_ICONERROR);
        closesocket(clientSocket);
        WSACleanup();
        return false;
    }

    std::thread serverMessagesThread(HandleServerMessages);
    serverMessagesThread.detach();

    return true;

}

void CleanupClient() {
    closesocket(clientSocket);
    WSACleanup();
}

void HandleServerMessages() {
    int bytesRead;
    do {
        bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
        if (bytesRead > 0) {
            buffer[bytesRead] = '\0';

            int size_needed = MultiByteToWideChar(CP_UTF8, 0, buffer, -1, NULL, 0);
            wchar_t* wide_buffer = new wchar_t[size_needed];
            MultiByteToWideChar(CP_UTF8, 0, buffer, -1, wide_buffer, size_needed);

            std::lock_guard<std::mutex> lock(messagesMutex);

            char* buffer = new char[size_needed];

            WideCharToMultiByte(CP_UTF8, 0, wide_buffer, -1, buffer, size_needed,
NULL, NULL);

            std::string narrow_str(buffer);

            messages.push(narrow_str);
        }
        else {
            std::cerr << "Failed to receive server response" << std::endl;
            break;
        }
    } while (bytesRead > 0);
}
```

```
std::string generateRandomID(int length) {
    std::string characters =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, characters.size() - 1);

    std::string randomID;
    for (int i = 0; i < length; ++i) {
        randomID += characters[dis(gen)];
    }
    return randomID;
}

std::string getCurrentDateTime() {
    std::time_t currentTime = std::time(nullptr);
    std::tm localTime = *std::localtime(&currentTime);

    char buffer[20];
    std::strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &localTime);

    return std::string(buffer);
}
```