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 №5

на тему

**ЭЛЕМЕНТЫ СЕТЕВОГО ПРОГРАММИРОВАНИЯ**

Студент Царук В.А.

Преподаватель Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc184918422)

[2 Краткие теоретические сведения 4](#_Toc184918423)

[3 Описание функций программы 5](#_Toc184918424)

[3.1 Сервер 5](#_Toc184918425)

[3.2 Клиент 5](#_Toc184918426)

[Заключение 7](#_Toc184918427)

[Список использованных источников 8](#_Toc184918428)

[Приложение А (обязательное) Исходный код программы 9](#_Toc184918429)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Целью выполнения лабораторной работы является разбор основ построения и функционирования сетей, стеков протоколов, программных интерфейсов. Рассмотреть организацию взаимодействия между процессами в сетях.

В качестве задачи необходимо реализовать программу или несколько программ, демонстрирующих взаимодействие между собой или с иным (внешним) ПО посредством сетевых протоколов и программного интерфейса сокетов. Реализовать известные сетевые протоколы прикладного уровня или разработать собственные.

Нужно реализовать упрощенный чат для нескольких пользователей с использованием сетевых сокетов.

Архитектура: централизованная (выделенный процесс-сервер и процессы-клиенты) или децентрализованная (процессы-клиенты с «серверными» функциями).

Сервер: создание сокета для приема соединений или отдельных сообщений; прием и временное хранение сообщений; передача сообщений адресно одному или нескольким клиентам; поддержание списка актуальных клиентов.

Клиент: обнаружение сервера и соединение с ним; ввод пользовательских сообщений и передача их серверу либо напрямую соответствующему клиенту; прием и отображение сообщений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Протокол – набор правил и процедур взаимодействия между одноименными уровнями различных систем, обеспечивают корректную связь участников взаимодействия в сети.

Интерфейс – набор правил и средств их реализации для взаимодействия между соседними уровнями одной системы, обеспечивают возможность модульного построения системы.

Стек протоколов в сети – набор протоколов, обслуживающих различные уровни взаимодействия. Протоколы в стеке проектируются с расчетом на совместную согласованную работу, но остаются достаточно независимыми для возможности замены на альтернативные с сохранением интерфейсов.

Сокет – программный объект, обычно системный, абстрагирующий точку доступа к транспортной системе. В типичном случае сокет сопоставляется одному из портов одного из транспортных протоколов и служит удобной для прикладного программирования унифицированной надстройкой над ним. Реже используются сокеты, связанные с другими протоколами, либо «сырые» (неспецифированные).

Универсальность интерфейса сокетов требует достаточно сложной его организации с многочисленными настройками. Характеристики сокета:

Тип («семейство») протоколов, «домен» (domain) – выбор стека протоколов (например, TCP/IP).

Тип («семейство») адресации – выбор способа идентификации участников взаимодействия; обычно определяется стеком протоколов, но потенциально может различаться.

Адрес сокета – идентификатор сокета как точки взаимодействия, идентификатор использующего его процесса; формат адреса зависит от типа адресации.

Тип сокета – неявно выбирает протокол взаимодействия в рамках семейства протоколов [1].

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Программа реализует упрощенный чат с использованием сетевых сокетов. Система состоит из двух компонентов: сервер и клиент. Они взаимодействуют друг с другом по протоколу TCP. Архитектура построена на основе централизованного сервера, который управляет подключениями клиентов и маршрутизацией сообщений.

3.1 Сервер

Используем Winsock для создания сокетов и работы с сетевыми соединениями. Сервер запускается на порту 27015 и прослушивает входящие соединения. Принимает подключения клиентов через функцию *accept* и каждое подключение обрабатывается в отдельном потоке для обеспечения многопользовательского взаимодействия. Сервер получает сообщения от клиентов, проверяет их содержимое и определяет тип отправки. Также для каждого клиента сохраняется информация о сокете, уникальном идентификаторе и список его сообщений. При отключении клиента сервер удаляет его из списка активных подключений (Рисунок 1).

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 1 – Работа сервера

3.2 Клиент

Клиент подключается к серверу, используя указанный IP-адрес и порт. При подключении клиент генерирует уникальный идентификатор длинной 20 символов, который отправляется серверу. Пользователь может отправлять сообщения широковещательно и адресно. Если пользователь вводит пустую строку, клиент завершает соединение и завершает выполнение (Рисунок 2).

Изображение выглядит как текст, снимок экрана, Шрифт, черный

Автоматически созданное описание

Рисунок 2 – Работа клиента

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы были изучены основы построения сетей, функционирования стеков протоколов, а также программных интерфейсов для взаимодействия между процессами в сетевой среде. Была разработана программа, демонстрирующая взаимодействие клиентов через централизованный сервер. Сервер реализовывал функционал приема соединений, хранения списка подключенных клиентов и маршрутизации сообщений. Клиенты выполняли обнаружение сервера, обмен сообщениями и отображение полученной информации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Протокол, интерфейс, стек протоколов. Модель ISO/OSi [Электронный ресурс]. – Электронный ресурс. – Режим доступа: <https://citforum.ru/nets/tpns/glava_17.shtml>. – Дата доступа: 10.12.2024.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

Листинг 1 – *client.cpp*

#define WIN32\_LEAN\_AND\_MEAN

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdlib.h>

#include <stdio.h>

#include <ctime>

#include <iostream>

#include <thread>

#include <string>

#include <atomic>

#include <stdlib.h>

// Need to link with Ws2\_32.lib, Mswsock.lib, and Advapi32.lib

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

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

#pragma comment (lib, "AdvApi32.lib")

#define DEFAULT\_BUFLEN 512

#define DEFAULT\_PORT "27015"

std::atomic<bool> isRunning(true); // flag for ending sending

SOCKET ConnectSocket = INVALID\_SOCKET;

std::string static GenRandom(const int len) {

srand(time(0));

static const char alphanum[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

std::string tmp\_s;

tmp\_s.reserve(len);

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

tmp\_s += alphanum[rand() % (sizeof(alphanum) - 1)];

}

return tmp\_s;

}

void static ReceiveMessages() {

int iResult;

char recvbuf[DEFAULT\_BUFLEN];

int recvbuflen = DEFAULT\_BUFLEN;

while (isRunning) {

iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);

if (iResult > 0) {

recvbuf[iResult] = '\0';

printf("Server message: %s\n", recvbuf);

}

else if (iResult == 0) {

printf("Connection closed by server.\n");

break;

}

else {

printf("Receive failed: %d\n", WSAGetLastError());

break;

}

}

}

void static SendMessages() {

int iResult;

char sendbuf[DEFAULT\_BUFLEN];

while (true) {

printf(" -> Enter msg: ");

fgets(sendbuf, sizeof(sendbuf), stdin);

size\_t len = strlen(sendbuf);

if (len > 0 && sendbuf[len - 1] == '\n') {

sendbuf[len - 1] = '\0';

}

if (strlen(sendbuf) == 0) {

sendbuf[0] = '\0';

iResult = send(ConnectSocket, sendbuf, 1, 0);

if (iResult == SOCKET\_ERROR) {

printf("Send failed: %d\n", WSAGetLastError());

isRunning = false;

break;

}

isRunning = false;

break;

}

else {

iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);

if (iResult == SOCKET\_ERROR) {

printf("Send failed: %d\n", WSAGetLastError());

break;

}

}

}

}

int main()

{

WSADATA wsaData;

int iResult;

// Initialize Winsock

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

if (iResult != 0) {

printf("WSAStartup failed with error: %d\n", iResult);

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;

// Resolve the server address and port

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

if (iResult != 0) {

printf("getaddrinfo failed: %d\n", iResult);

WSACleanup();

return 1;

}

// Create a SOCKET for connecting to server

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

// Check for errors to ensure that the socket is a valid socket

if (ConnectSocket == INVALID\_SOCKET) {

printf("Error at socket(): %ld\n", WSAGetLastError());

freeaddrinfo(result);

WSACleanup();

return 1;

}

// Connect to server.

iResult = connect(ConnectSocket, result->ai\_addr, (int)result->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

closesocket(ConnectSocket);

ConnectSocket = INVALID\_SOCKET;

}

// Free the resources returned by getaddrinfo and print an error message

freeaddrinfo(result);

if (ConnectSocket == INVALID\_SOCKET) {

printf("Unable to connect to server!\n");

WSACleanup();

return 1;

}

printf("Connected to server.\n");

std::string sendbufID = GenRandom(20);

printf("Generated id: %s\n", sendbufID.c\_str());

iResult = send(ConnectSocket, sendbufID.c\_str(), 20, 0);

if (iResult == SOCKET\_ERROR) {

printf("Send failed: %d\n", WSAGetLastError());

closesocket(ConnectSocket);

WSACleanup();

return 1;

}

printf("Bytes sent: %d\n", iResult);

printf("INSTRUCTIONS:\nEnter message (50 symbols) and '->name' in the end if you want the specific client to be sent the msg.\n");

std::thread recvThread(ReceiveMessages);

std::thread sendThread(SendMessages);

sendThread.join();

recvThread.join();

// Cleanup

closesocket(ConnectSocket);

WSACleanup();

return 0;

}

Листинг 2 – server.cpp

*﻿#include <winsock2.h>*

*#include <ws2tcpip.h>*

*#include <stdio.h>*

*#include <iostream>*

*#include <thread>*

*#include <vector>*

*#include <mutex>*

*#include <string>*

*#include <unordered\_map>*

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

*#define DEFAULT\_PORT "27015"*

*#define DEFAULT\_BUFLEN 512*

*// Struct to store client information*

*struct ClientInfo {*

*SOCKET socket;*

*char name[21];*

*std::vector<std::string> msgs;*

*// Конструктор для удобной инициализации*

*ClientInfo(SOCKET s, const char\* n)*

*: socket(s), msgs() {*

*strncpy\_s(name, n, sizeof(name) - 1);*

*name[sizeof(name) - 1] = '\0'; // Ensure null termination*

*}*

*};*

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

*std::vector<ClientInfo> clientSockets; // List of connected client sockets*

*std::mutex clientMutex; // Mutex to synchronize access to the clientSockets vector*

*// Function to broadcast a message to all clients*

*void static BroadcastMessage(const std::string& message, SOCKET senderSocket) {*

*std::lock\_guard<std::mutex> lock(clientMutex);*

*for (const auto& client : clientSockets) {*

*if (client.socket != senderSocket) { // Don't send the message back to the sender*

*send(client.socket, message.c\_str(), int(message.size()), 0);*

*}*

*}*

*}*

*void HandleClient(SOCKET ClientSocket) {*

*char recvbuf[DEFAULT\_BUFLEN];*

*int iResult, iSendResult;*

*int recvbuflen = DEFAULT\_BUFLEN;*

*iResult = recv(ClientSocket, recvbuf, DEFAULT\_BUFLEN, 0);*

*if (iResult <= 0) {*

*closesocket(ClientSocket);*

*return;*

*}*

*recvbuf[iResult] = '\0'; // Null-terminate the received name*

*char clientName[21];*

*// Safely copy the received name to clientName*

*strncpy\_s(clientName, recvbuf, sizeof(clientName) - 1);*

*clientName[sizeof(clientName) - 1] = '\0'; // Ensure null termination*

*// Add client to the list*

*{*

*std::lock\_guard<std::mutex> lock(clientMutex);*

*clientSockets.push\_back(ClientInfo(ClientSocket, clientName));*

*}*

*std::cout << "Client connected: " << recvbuf << std::endl;*

*// Receive until the peer shuts down the connection*

*do {*

*iResult = recv(ClientSocket, recvbuf, recvbuflen, 0); // iResult - number of received bytes*

*printf("Bytes received: %d\n", iResult);*

*if (iResult > 0) {*

*recvbuf[iResult] = '\0';*

*std::string message = recvbuf;*

*// Check for empty msg*

*if (strlen(recvbuf) == 0) {*

*printf("Client %s disconnected.\n", clientName);*

*break;*

*}*

*else {*

*printf("Message received: %s\n", recvbuf);*

*}*

*// Check if the message contains "->" for directed messaging*

*size\_t arrowPos = message.find("->");*

*if (arrowPos != std::string::npos) {*

*std::string actualMessage = message.substr(0, arrowPos); // Extract the actual message*

*std::string targetID = message.substr(arrowPos + 2); // Extract the target client ID*

*// Save the message in the client's message list*

*{*

*std::lock\_guard<std::mutex> lock(clientMutex);*

*for (auto& client : clientSockets) {*

*if (client.socket == ClientSocket) {*

*client.msgs.push\_back(actualMessage); // Add the message to the client's message list*

*break;*

*}*

*}*

*}*

*// Find the target client*

*bool clientFound = false;*

*{*

*std::lock\_guard<std::mutex> lock(clientMutex);*

*for (const auto& client : clientSockets) {*

*if (client.name == targetID) {*

*clientFound = true;*

*std::string name(clientName);*

*std::string directedMessage = name + " (private): " + actualMessage;*

*send(client.socket, directedMessage.c\_str(), static\_cast<int>(directedMessage.size()), 0);*

*break;*

*}*

*}*

*}*

*if (!clientFound) {*

*std::string errorMsg = "Error: Client with ID '" + targetID + "' not found.\n";*

*send(ClientSocket, errorMsg.c\_str(), static\_cast<int>(errorMsg.size()), 0);*

*}*

*}*

*else {*

*// If no "->" found, treat as a broadcast*

*std::string name(clientName);*

*std::string broadcastMessage = name + ": " + message;*

*BroadcastMessage(broadcastMessage, ClientSocket);*

*}*

*}*

*else {*

*printf("Recv failed: %d\n", WSAGetLastError());*

*break;*

*}*

*} while (iResult > 0);*

*// Remove the client from the list after disconnection*

*{*

*std::lock\_guard<std::mutex> lock(clientMutex);*

*clientSockets.erase(std::remove\_if(clientSockets.begin(), clientSockets.end(),*

*[ClientSocket](const ClientInfo& client) {*

*return client.socket == ClientSocket;*

*}),*

*clientSockets.end());*

*}*

*printf("Client %s removed from list.\n", clientName);*

*closesocket(ClientSocket);*

*return;*

*}*

*int main() {*

*WSADATA wsaData; // The WSADATA structure contains information about the Windows Sockets implementation.*

*int iResult;*

*// Initialize Winsock (MAKEWORD - initializing version, )*

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

*if (iResult != 0) {*

*printf("WSAStartup failed: %d\n", iResult);*

*return 1;*

*}*

*// clear hints structure*

*ZeroMemory(&hints, sizeof(hints));*

*hints.ai\_family = AF\_INET; // specify the IPv4 address family*

*hints.ai\_socktype = SOCK\_STREAM; // specify a stream socket*

*hints.ai\_protocol = IPPROTO\_TCP; // specify the TCP protocol*

*hints.ai\_flags = AI\_PASSIVE; // caller intends to use the returned socket address structure in a call to the bind function*

*// Resolve the local address and port to be used by the server*

*iResult = getaddrinfo("0.0.0.0", DEFAULT\_PORT, &hints, &result);*

*if (iResult != 0) {*

*printf("getaddrinfo failed: %d\n", iResult);*

*WSACleanup();*

*return 1;*

*}*

*// Create a SOCKET*

*SOCKET ListenSocket = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);*

*if (ListenSocket == INVALID\_SOCKET) {*

*printf("Error at socket(): %ld\n", WSAGetLastError());*

*freeaddrinfo(result);*

*WSACleanup();*

*return 1;*

*}*

*// Bind exsisting socket to address*

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

*if (iResult == SOCKET\_ERROR) {*

*printf("bind failed with error: %d\n", WSAGetLastError());*

*freeaddrinfo(result); // sock adrr no longer needed*

*closesocket(ListenSocket);*

*WSACleanup();*

*return 1;*

*}*

*// Define listening mode for socket*

*// SOMAXCONN - maximum reasonable number of pending connections in the queue*

*if (listen(ListenSocket, SOMAXCONN) == SOCKET\_ERROR) {*

*printf("Listen failed with error: %ld\n", WSAGetLastError());*

*closesocket(ListenSocket);*

*WSACleanup();*

*return 1;*

*}*

*printf("Socket is listening\n");*

*while (true) {*

*SOCKET ClientSocket = INVALID\_SOCKET;*

*ClientSocket = accept(ListenSocket, NULL, NULL);*

*if (ClientSocket == INVALID\_SOCKET) {*

*printf("accept failed: %d\n", WSAGetLastError());*

*continue; // Don't terminate the server; just skip this iteration*

*}*

*printf("New client connected.\n");*

*// Add the new client*

*std::thread(HandleClient, ClientSocket).detach();*

*}*

*closesocket(ListenSocket);*

*WSACleanup();*

*return 0;*

*}*