Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Институт информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"

профиль "Программное обеспечение средств вычислительной техники и автоматизированных систем"

**Курсовая работа**по дисциплине «Сетевое программирование»

**Разработка сетевого приложения «Чат».   
Асинхронный сервер с установлением соединения с использованием функции select**

Выполнил:

студент гр. ИП 214 \_\_\_\_\_\_\_\_\_\_\_\_\_/ Номоконов Д. А. /

ФИО студента

« 20 » марта 2025 г.

Проверил

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Павский К. В. /

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2025 г.

Оглавление

[Задание 3](#_Toc193044041)

[Описание протокола 3](#_Toc193044042)

[Основные этапы работы TCP 3](#_Toc193044043)

[Преимущества и недостатки TCP 5](#_Toc193044044)

[Флаги TCP 5](#_Toc193044045)

[Описание реализации 6](#_Toc193044046)

[Работа программы 8](#_Toc193044047)

[Листинг 8](#_Toc193044048)

# **Задание**

В рамках курсовой работы необходимо написать асинхронный сервер с установлением соединения с использованием функции select. Реализация на языке C++ для Unix-подобных операционных систем.

# **Описание протокола**

TCP (Transmission Control Protocol) – это протокол управления передачей. Один из основных протоколов Интернета, который обеспечивает надежную передачу данных между узлами сети. Он работает на транспортном уровне модели OSI и обеспечивает установление, поддержку и завершение соединений, а также гарантирует надежность и порядок передачи данных.

## **Основные этапы работы TCP**

1. Установление соединения (Three-Way Handshake)

Перед тем как начать обмениваться данными, TCP устанавливает соединение между двумя узлами через трехстороннее рукопожатие (Three-Way Handshake):

* **Шаг 1:** Клиент отправляет серверу сегмент с флагом **SYN** (synchronize), который указывает на желание установить соединение. В этом сегменте указывается начальный номер последовательности (Sequence Number).
* **Шаг 2:** Сервер, получив запрос на соединение, отвечает клиенту сегментом с флагами **SYN** и **ACK** (acknowledge). Это подтверждение, что сервер готов к соединению, и также включает свой номер последовательности.
* **Шаг 3:** Клиент отправляет серверу подтверждение с флагом **ACK**, сообщая, что он получил ответ от сервера и готов к обмену данными.

После этого соединение считается установленным, и стороны могут начать обмен данными.

1. Передача данных

После установления соединения начинается передача данных. Этот процесс включает несколько важных этапов:

* **Нумерация байтов**: каждый байт данных, передаваемый по TCP, имеет уникальный номер. Этот номер используется для отслеживания, какие байты были отправлены и приняты.
* **Подтверждения (ACK)**: когда получатель получает данные, он отправляет подтверждение (ACK) отправителю, сообщая, что он принял данные. При этом в ACK содержится номер последнего принятого байта.
* **Окно скольжения (Sliding Window)**: этот механизм позволяет получателю указать отправителю, сколько данных он может передать без ожидания подтверждения. Это помогает предотвратить перегрузку сети.

1. Контроль потока и управление перегрузкой

* **Контроль потока:** с помощью окна скольжения TCP регулирует количество данных, которые могут быть отправлены без получения подтверждения. Это предотвращает переполнение приемной стороны.
* **Управление перегрузкой**: TCP использует различные алгоритмы для контроля за перегрузкой сети. Один из таких алгоритмов — это **медленный старт (Slow Start)**, который постепенно увеличивает скорость передачи данных до тех пор, пока не будет обнаружена перегрузка сети. Если это происходит, TCP уменьшает скорость передачи данных.

1. Коррекция ошибок

* **Проверка целостности**: Каждый сегмент данных TCP включает контрольную сумму (checksum), которая позволяет проверять целостность данных. Если получатель обнаруживает ошибку, он может запросить повторную передачу данных.
* **Повторная передача (Retransmission)**: если отправленный сегмент не был подтвержден в течение определенного времени (например, из-за потери пакета), то он будет автоматически отправлен повторно.

1. Закрытие соединения (Four-Way Handshake)

Когда одна из сторон завершает передачу данных, соединение должно быть закрыто. Для этого используется четырехсторонний процесс завершения соединения:

* **Шаг 1**: Сторона, которая инициирует завершение соединения, отправляет сегмент с флагом **FIN** (finish). Это сигнализирует о том, что эта сторона больше не будет отправлять данные.
* **Шаг 2**: Получатель подтверждает получение сегмента с флагом **ACK**, сообщая, что он принял запрос на завершение соединения.
* **Шаг 3**: Получатель может, в свою очередь, отправить свой сегмент с флагом **FIN**, если он также завершил передачу данных.
* **Шаг 4**: Инициатор завершения соединения подтверждает получение второго **FIN** сегмента, и соединение окончательно закрывается.

TCP гарантирует, что все данные будут доставлены в правильном порядке и без потерь:

* Если сегмент данных не был получен (например, из-за ошибки или потери в сети), он будет повторно отправлен.
* Если данные получены не в том порядке, в котором они были отправлены, TCP будет их переставлять.
* Все данные, передаваемые по TCP, сопровождаются контрольной суммой для проверки целостности.

## **Преимущества и недостатки TCP**

**Преимущества**:

* **Надежность**: гарантированная доставка данных и их целостность.
* **Порядок**: данные передаются в том порядке, в котором они были отправлены.
* **Контроль потока**: TCP предотвращает переполнение принимающей стороны.
* **Управление перегрузкой**: TCP может адаптировать скорость передачи данных, чтобы избежать перегрузки сети.

**Недостатки**:

* **Производительность**: из-за необходимости подтверждений и контроля за потоком, TCP может быть медленнее, чем без подтверждений (например, в случае с UDP).
* **Задержка**: время на установление соединения и подтверждения может добавить задержки при передаче данных.

## **Флаги TCP**

Каждый пакет TCP может содержать различные флаги, которые обозначают тип передаваемого пакета. Основные флаги:

* **SYN** — используется для установления соединения.
* **ACK** — подтверждение получения данных.
* **FIN** — сигнализирует о завершении соединения.
* **RST** — используется для сброса соединения.
* **PSH** — просит получателя сразу передать данные на приложение.
* **URG** — указывает на срочность данных.

# **Описание реализации**

Сервер чата реализован на языке C++ и использует сокеты для управления подключения клиентов. Он поддерживает многопользовательские комнаты, обмен сообщениями и передачу файлов.

1. Подключение клиентов

Сервер принимает входящие соединения через TCP-сокет:

* Создает серверный сокет.
* Устанавливается прослушивание порта.
* Добавляется в список отслеживаемых дескрипторов.
* В цикле select() ожидается активности на сокетах.
* Принимаются новые клиенты через accept().
* Отправляется запрос на ввод имени.

1. Аутентификация

При подключении клиент отправляет имя

* Сервер получает имя и проверяет на пустоту и уникальность
* Клиент получает приветственное сообщение.

1. Работа с select()

Для одновременной обработки нескольких клиентов используется select()

* Сервер создает множество отслеживаемых файловых дескрипторов fd\_set.
* В главный массив fd\_set добавляется серверный сокет.
* В основном цикле вызывается select(), который блокируется, пока не появится активность на каком-либо сокете.
* Когда новый клиент подключается, сервер вызывает accept() и добавляет новый сокет в fd\_set.
* Если активен клиентский сокет, сервер читает сообщения с помощью recv().
* Если клиент отключается, его сокет удаляется из множества fd\_set.

1. Работа с комнатами

* /create <name\_room> <max\_users> - сервер создает комнату, если ее нет, при этом устанавливая лимит участников. Создатель автоматически присоединяется к этой комнате.
* /join <name\_room> - команда для присоединения клиента к комнате. Сервер проверяет существование комнаты, количество мест. При соблюдении этих условий, клиент подключается к комнате.
* /leave – команда выхода из комнаты. Клиент удаляется из списка комнаты. Если комната пуста, она удаляется.

1. Список комнат и пользователей

* /rooms – отображаемый список доступных комнат.
* /users – показывает всех участников.

1. Обмен сообщениями

Клиенты могут отправлять сообщения в комнату.

* Сервер пересылает их всем участникам комнаты.
* Сообщения отображаются в формате [имя]: текст.

1. Передача файлов (P2P-передача)

В системе реализована возможность передачи файлов напрямую между клиентами минуя основной сервер. Это снижает нагрузку на сервер и ускоряет процесс обмена данными.

* Инициализация передачи. Клиент, желающий отправить файл, отправляет команду /upload <filename> (при этом сервер ее интерпретирует как START\_UPLOAD <filename>) серверу.
* Сервер определяет количество клиентов, готовых принять файл, и сообщает отправителю.
* Создание P2P-соединения. Отправитель запускает прослушивание на временном P2P-сервере, создавая для этого новый сокет.
* Отправитель сообщает основному серверу о своем локальном P2P-порте, после чего сервер передает эту информацию клиентам-получателям.
* Подключение получателей. Получатели получают ip и port отправителя от основного сервера.
* Каждый клиент устанавливает прямое соединение с отправителем, вызывая connect().
* Передача файла. Отправитель читает данные из файла блоками (в бинарном виде) и отправляет их всем подключенным клиентам. Клиенты записывают полученные данные в локальный файл. После завершения отправки сервер передает END\_FILE, сигнализируя о завершении передачи.
* Подтверждение успешной передачи. Каждый клиент отправляет OK отправителю после успешного получения файла. Отправитель подсчитывает количество успешных подтверждений и сообщает об этом серверу.

1. Завершение работы

Клиент может отключится командой /quit.

* Сервер удаляет его из комнаты.
* Если клиент разрывает соединение, он автоматически удаляется.

# **Работа программы**

Запуск сервера:

Изображение выглядит как снимок экрана, текст, программное обеспечение, мультимедиа

Контент, сгенерированный ИИ, может содержать ошибки.

Присоединение клиента + основной функционал:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

# 

# **Листинг**

Client.cpp:

#include <iostream>

#include <thread>

#include <string>

#include <cstring>

#include <fstream>

#include <vector>

#include <arpa/inet.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/socket.h>

#include <netinet/in.h>

using namespace std;

#define BUFFER\_SIZE 1024

void sendFile(const string& filename, int server\_fd, int main\_sockfd, int num\_clients) {

ifstream file(filename, ios::binary);

if (!file || file.peek() == ifstream::traits\_type::eof()) {

cerr << "Error: Cannot open file or file is empty: " << filename << endl;

send(main\_sockfd, "FILE\_ERROR Empty or not found", 28, 0);

close(server\_fd);

return;

}

if (num\_clients == 0) {

send(main\_sockfd, "There are no clients to send files to", 38, 0);

close(server\_fd);

return;

}

struct sockaddr\_in p2p\_addr;

socklen\_t addr\_len = sizeof(p2p\_addr);

getsockname(server\_fd, (struct sockaddr\*)&p2p\_addr, &addr\_len);

int p2p\_port = ntohs(p2p\_addr.sin\_port);

string port\_msg = "P2P\_PORT " + to\_string(p2p\_port);

send(main\_sockfd, port\_msg.c\_str(), port\_msg.length(), 0);

vector<int> client\_fds;

struct timeval timeout;

timeout.tv\_sec = 10;

timeout.tv\_usec = 0;

setsockopt(server\_fd, SOL\_SOCKET, SO\_RCVTIMEO, &timeout, sizeof(timeout));

for (int i = 0; i < num\_clients; ++i) {

struct sockaddr\_in client\_addr;

socklen\_t client\_addr\_len = sizeof(client\_addr);

int client\_fd = accept(server\_fd, (struct sockaddr\*)&client\_addr, &client\_addr\_len);

if (client\_fd >= 0) {

client\_fds.push\_back(client\_fd);

} else {

break;

}

}

if (client\_fds.empty()) {

send(main\_sockfd, "FILE\_ERROR No clients connected", 31, 0);

file.close();

close(server\_fd);

return;

}

file.clear();

file.seekg(0);

char fileBuffer[BUFFER\_SIZE];

streamsize bytesRead;

while ((bytesRead = file.read(fileBuffer, BUFFER\_SIZE).gcount()) > 0) {

for (int client\_fd : client\_fds) {

send(client\_fd, fileBuffer, bytesRead, 0);

}

}

for (int client\_fd : client\_fds) {

send(client\_fd, "END\_FILE", 8, 0);

}

int successful\_receivers = 0;

for (int client\_fd : client\_fds) {

char buffer[BUFFER\_SIZE];

int recvBytes = recv(client\_fd, buffer, BUFFER\_SIZE, 0);

if (recvBytes > 0 && strncmp(buffer, "OK", 2) == 0) {

successful\_receivers++;

}

close(client\_fd);

}

file.close();

close(server\_fd);

string result\_msg = "FILE\_SENT " + filename + " " + to\_string(successful\_receivers) + "/" + to\_string(num\_clients);

send(main\_sockfd, result\_msg.c\_str(), result\_msg.length(), 0);

}

void receiveFile(const string& peer\_ip, int peer\_port, const string& filename, int main\_sockfd) {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0) {

perror("Socket creation error");

return;

}

struct sockaddr\_in peer\_addr;

peer\_addr.sin\_family = AF\_INET;

peer\_addr.sin\_port = htons(peer\_port);

inet\_pton(AF\_INET, peer\_ip.c\_str(), &peer\_addr.sin\_addr);

if (connect(sockfd, (struct sockaddr\*)&peer\_addr, sizeof(peer\_addr)) < 0) {

perror("Connection error");

close(sockfd);

return;

}

system("mkdir -p downloads");

ofstream file("downloads/" + filename, ios::binary);

if (!file) {

cout << "Error: Cannot save file " << filename << endl;

close(sockfd);

return;

}

char buffer[BUFFER\_SIZE];

int bytesRead;

while ((bytesRead = recv(sockfd, buffer, BUFFER\_SIZE, 0)) > 0) {

if (strncmp(buffer, "END\_FILE", 8) == 0) break;

file.write(buffer, bytesRead);

}

file.close();

send(sockfd, "OK", 2, 0);

close(sockfd);

string received\_msg = "FILE\_RECEIVED " + filename;

send(main\_sockfd, received\_msg.c\_str(), received\_msg.length(), 0);

}

void receiveMessages(int sockfd) {

char buffer[BUFFER\_SIZE];

string peer\_ip;

string filename;

while (true) {

memset(buffer, 0, BUFFER\_SIZE);

int bytes\_received = recv(sockfd, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received <= 0) {

cout << "\nDisconnecting from the server...\n";

close(sockfd);

exit(0);

}

string response = buffer;

if (response.find("START\_UPLOAD ") == 0) {

string rest = response.substr(13);

size\_t pos = rest.find(" ");

string filename = rest.substr(0, pos);

int num\_clients = stoi(rest.substr(pos + 1));

int p2p\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in p2p\_addr;

p2p\_addr.sin\_family = AF\_INET;

p2p\_addr.sin\_addr.s\_addr = INADDR\_ANY;

p2p\_addr.sin\_port = 0;

if (::bind(p2p\_fd, (struct sockaddr\*)&p2p\_addr, sizeof(p2p\_addr)) < 0) {

perror("P2P bind error");

close(p2p\_fd);

continue;

}

listen(p2p\_fd, num\_clients);

thread(sendFile, filename, p2p\_fd, sockfd, num\_clients).detach();

}

else if (response.find("PEER\_UPLOAD ") == 0) {

string rest = response.substr(12);

size\_t pos = rest.find(" ");

peer\_ip = rest.substr(0, pos);

filename = rest.substr(pos + 1);

}

else if (response.find("PEER\_UPLOAD\_PORT ") == 0) {

int peer\_port = stoi(response.substr(17));

thread(receiveFile, peer\_ip, peer\_port, filename, sockfd).detach();

}

else {

cout << "\r" << response << endl;

cout << "> " << flush;

}

}

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

cerr << "Usage: " << argv[0] << " <server\_ip> <port>\n";

return 1;

}

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0) {

perror("Socket creation error");

return 1;

}

struct sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(atoi(argv[2]));

if (inet\_pton(AF\_INET, argv[1], &servaddr.sin\_addr) <= 0) {

perror("Invalid server address");

close(sockfd);

return 1;

}

if (connect(sockfd, (struct sockaddr\*)&servaddr, sizeof(servaddr)) < 0) {

perror("Connection error");

close(sockfd);

return 1;

}

cout << "Connected to the server!\n";

thread receiveThread(receiveMessages, sockfd);

string message;

cout << "> " << flush;

while (true) {

getline(cin, message);

if (message == "/exit") {

break;

}

if (send(sockfd, message.c\_str(), message.length(), 0) == -1) {

cerr << "Error sending the message!\n";

break;

}

cout << "> " << flush;

}

close(sockfd);

receiveThread.detach();

return 0;

}

Server.cpp:

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <set>

#include <string>

#include <cstring>

#include <sstream>

#include <fstream>

#include <algorithm>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <fcntl.h>

using namespace std;

// ==== Visual ====

#define RESET "\033[0m"

#define RED "\033[31m"

#define GREEN "\033[32m"

#define YELLOW "\033[33m"

#define BLUE "\033[34m"

#define CYAN "\033[36m"

// === Global ===

#define PORT 59851 // 8080 BUSY on mac

#define MAX\_CLIENT 10

#define BUFFER\_SIZE 1024

struct ClientInfo {

int socket;

string room;

string name;

};

struct RoomInfo {

int maxClients = MAX\_CLIENT;

vector<int> clients;

};

vector<ClientInfo> clients;

unordered\_map<string, RoomInfo> rooms;

void setNonblocking(int sock) {

int flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL, flags | O\_NONBLOCK);

}

void sendMessage(int clientFd, const string &message) {

send(clientFd, message.c\_str(), message.size(), 0);

}

void sendToRoom(const string &room, int senderFd, const string &message) {

auto it = rooms.find(room);

if(it == rooms.end()) return;

RoomInfo &roomInfo = it->second;

for (int clientFd : roomInfo.clients) {

if (clientFd != senderFd) sendMessage(clientFd, message);

}

}

void setNameClient(ClientInfo &client, string client\_name) {

client\_name.erase(remove(client\_name.begin(), client\_name.end(), '\n'), client\_name.end());

client\_name.erase(remove(client\_name.begin(), client\_name.end(), '\r'), client\_name.end());

if (!client\_name.empty()) {

client.name = client\_name;

sendMessage(client.socket, "Welcome, " + client.name + "!\nUse " YELLOW "/help" RESET " to get page of help.");

} else {

sendMessage(client.socket, "Error: Name cannot be empty! Try again.");

}

}

void commandClientMessage(ClientInfo &client, const string message) {

stringstream ss(message);

string command;

ss >> command;

if (command == "/create") {

string nameRoom;

string maxUsersRoom;

int maxClients = MAX\_CLIENT;

ss >> nameRoom >> maxUsersRoom;

if (nameRoom.empty()) {

sendMessage(client.socket, "Name room is empty! Reapet create room now");

return;

}

if (!maxUsersRoom.empty()) {

try {

maxClients = stoi(maxUsersRoom);

} catch (...) {

sendMessage(client.socket, "Invalid user limit! Must be a number between 2 and 10.");

return;

}

}

if (maxClients < 2 || maxClients > 10) {

sendMessage(client.socket, "User limit must be between 2 and 10.");

return;

}

if (rooms.find(nameRoom) == rooms.end()) {

rooms[nameRoom] = { maxClients, {client.socket} };

client.room = nameRoom;

sendMessage(client.socket, "Room " + nameRoom + " was created successfully!");

} else {

sendMessage(client.socket, "A room with that name already exists.");

}

}

else if (command == "/join") {

string nameRoom;

ss >> nameRoom;

if (nameRoom.empty()) {

sendMessage(client.socket, "Name room is empty! Reapet create room now");

return;

}

auto it = rooms.find(nameRoom);

if (it != rooms.end()) {

RoomInfo &room = it->second;

if (room.clients.size() >= static\_cast<size\_t>(room.maxClients)) {

sendMessage(client.socket, "This room is full! Max users: " + to\_string(room.maxClients));

return;

}

client.room = nameRoom;

room.clients.push\_back(client.socket);

sendMessage(client.socket, "You joined the room '" + nameRoom);

sendToRoom(nameRoom, client.socket, client.name + " has joined the room!");

} else {

sendMessage(client.socket, "This room is not found!");

}

}

else if (command == "/users") {

string userList = "List users:\n";

for (const auto &c : clients) {

userList += c.name + " " + to\_string(c.socket);

if (!c.room.empty()) {

userList += " (Room: " + c.room + ")";

} else {

userList += " (Not in the room)";

}

userList += "\n";

}

sendMessage(client.socket, userList);

}

else if (command == "/rooms") {

string allRoom = "List of rooms:\n";

int count = 1;

for (const auto &[roomName, roomInfo] : rooms) {

allRoom += to\_string(count) + ". " + roomName + " (" +

to\_string(roomInfo.clients.size()) + "/" +

to\_string(roomInfo.maxClients) + " users)\n";

count++;

}

if (rooms.empty()) {

allRoom += "No active rooms.\n";

}

sendMessage(client.socket, allRoom);

}

else if (command == "/leave") {

if (client.room.empty()) {

sendMessage(client.socket, "You're not in a room!");

return;

}

string nameRoom = client.room;

auto it = rooms.find(nameRoom);

if (it != rooms.end()) {

RoomInfo &room = it->second;

room.clients.erase(remove(room.clients.begin(), room.clients.end(), client.socket), room.clients.end());

sendMessage(client.socket, "You left the room: " + nameRoom);

sendToRoom(nameRoom, client.socket, client.name + " has left the room!");

if (room.clients.empty()) {

rooms.erase(it);

}

client.room.clear();

}

}

else if (command == "/upload") {

if (client.room.empty()) {

sendMessage(client.socket, "You're not in a room!");

return;

}

string filename;

ss >> filename;

if (filename.empty()) {

sendMessage(client.socket, "Usage: /upload <filename>");

return;

}

struct sockaddr\_in client\_addr;

socklen\_t addr\_len = sizeof(client\_addr);

getpeername(client.socket, (struct sockaddr\*)&client\_addr, &addr\_len);

char client\_ip[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &client\_addr.sin\_addr, client\_ip, INET\_ADDRSTRLEN);

auto it = rooms.find(client.room);

int num\_clients = it->second.clients.size() - 1;

cout << "Sending START\_UPLOAD to " << client.name << ": " << filename << " " << num\_clients << endl;

sendMessage(client.socket, "START\_UPLOAD " + filename + " " + to\_string(num\_clients));

cout << "Sending PEER\_UPLOAD to room " << client.room << ": " << client\_ip << " " << filename << endl;

string upload\_msg = "PEER\_UPLOAD " + string(client\_ip) + " " + filename;

sendToRoom(client.room, client.socket, upload\_msg);

}

else if (command == "/help") {

string allCommand =

YELLOW "/create" RESET " <name room> <max users>\n"

YELLOW "/join" RESET " <name room>\n"

YELLOW "/users\n" RESET

YELLOW "/rooms\n" RESET

YELLOW "/leave\n" RESET

YELLOW "/upload" RESET " <filename>\n";

sendMessage(client.socket, allCommand);

}

else {

if (client.room.empty()) sendMessage(client.socket, "You're not in the room!");

else sendToRoom(client.room, client.socket, client.name + ": " + message);

}

}

int main() {

int server\_fd, new\_socket, max\_sd, activity;

struct sockaddr\_in address;

fd\_set readfds;

server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

setNonblocking(server\_fd);

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

if (::bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

perror("Error bind()");

close(server\_fd);

return 1;

}

listen(server\_fd, MAX\_CLIENT);

cout << "Server starts on port " << PORT << endl;

while (true) {

FD\_ZERO(&readfds);

FD\_SET(server\_fd, &readfds);

max\_sd = server\_fd;

for (const auto &client : clients) {

FD\_SET(client.socket, &readfds);

if (client.socket > max\_sd) max\_sd = client.socket;

}

activity = select(max\_sd + 1, &readfds, nullptr, nullptr, nullptr);

if (activity < 0 && errno != EINTR) {

perror("Error select()");

}

if (FD\_ISSET(server\_fd, &readfds)) {

socklen\_t addr\_len = sizeof(address);

new\_socket = accept(server\_fd, (struct sockaddr \*)&address, &addr\_len);

setNonblocking(new\_socket);

sendMessage(new\_socket, "Input your name: ");

clients.push\_back({new\_socket, "", ""});

}

for (auto it = clients.begin(); it != clients.end(); ) {

if (FD\_ISSET(it->socket, &readfds)) {

char buffer[BUFFER\_SIZE] = {0};

int valread = read(it->socket, buffer, BUFFER\_SIZE);

if (valread == 0) {

close(it->socket);

cout << it->name << " disconnected" << endl;

it = clients.erase(it);

} else {

buffer[valread] = '\0';

string msg = buffer;

if (msg.find("P2P\_PORT ") == 0) {

string port = msg.substr(9);

string upload\_msg = "PEER\_UPLOAD\_PORT " + port + "\n";

sendToRoom(it->room, it->socket, upload\_msg);

++it;

}

else if (msg.find("FILE\_SENT ") == 0) {

string result = it->name + ": " + msg.substr(10);

sendToRoom(it->room, it->socket, result);

sendMessage(it->socket, result);

++it;

}

else if (msg.find("FILE\_RECEIVED ") == 0) {

string result = it->name + " received " + msg.substr(14);

sendToRoom(it->room, it->socket, result);

sendMessage(it->socket, result);

++it;

}

else if (msg.find("FILE\_ERROR") == 0) {

string result = it->name + ": File transfer failed - " + msg.substr(11);

sendToRoom(it->room, it->socket, result);

sendMessage(it->socket, result);

++it;

}

else if (it->name.empty()) {

setNameClient(\*it, msg);

++it;

} else {

commandClientMessage(\*it, msg);

++it;

}

}

} else {

++it;

}

}

}

close(server\_fd);

return 0;

}

# **Список источников**

1. Павский К. В., Ефимов А. В. Разработка сетевых приложений (протоколы TCP/IP, клиент-сервер, PCAP, Boost.ASIO): Учебное пособие / Сибирский государственный университет телекоммуникаций и информатики. – Новосибирск, 2018. – 80 с.
2. Протоколы TCP/IP и разработка сетевых приложений: учеб. пособие / К.В. Павский; Сиб. гос. ун-т телекоммуникаций и информатики. - Новосибирск: СибГУТИ, 2013. – 130c.
3. IBM : сайт. – URL: https://www.ibm.com/docs/en/zos/2.5.0?topic=calls-select (дата обращения: 05.03.2025).
4. Пролетарский, А. В. Технологии TCP/IP в современных компьютерных сетях / А. В. Пролетарский, Е. А. Ромашкина, Е. В. Смирнова. – Москва : Издательство МГТУ им. Н.Э. Баумана, 2022. – 640 с. – ISBN 978-5-7038-5166-1.