

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ  
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»  
(БГТУ им. В.Г. Шухова)**



**ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ**

**Лабораторная работа №4**

по дисциплине: Компьютерные сети

тема: «Программирование протоколов TCP/UDP с использованием библиотеки Winsock»

Выполнил: ст. группы ПВ-223  
Пахомов Владислав Андреевич

Проверили:  
Рубцов Константин Анатольевич

Белгород 2025 г.

## Лабораторная работа №4

### Программирование протоколов TCP/UDP с использованием библиотеки Winsock

#### Вариант 6

**Цель работы:** изучить протоколы TCP/UDP, основные функции библиотеки Winsock и составить программу для приема/передачи пакетов.

#### Краткие теоретические сведения

##### Протокол TCP

Transmission Control Protocol (TCP) (протокол управления передачей) - один из основных сетевых протоколов Интернета, предназначенный для управления передачей данных в сетях и подсетях TCP/IP. Выполняет функции протокола транспортного уровня модели OSI.

TCP — это транспортный механизм, предоставляющий поток данных, с предварительной установкой соединения, за счёт этого дающий уверенность в достоверности получаемых данных, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета. В отличие от UDP гарантирует целостность передаваемых данных и уведомление отправителя о результатах передачи. Реализация TCP, как правило, встроена в ядро ОС, хотя есть и реализации TCP в контексте приложения.

Когда осуществляется передача от компьютера к компьютеру через Интернет, TCP работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. Также TCP осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере. Программы для электронной почты и обмена файлами используют TCP. TCP контролирует длину сообщения, скорость обмена сообщениями, сетевой трафик.

TCP протокол базируется на IP для доставки пакетов, но добавляются две важные вещи: во-первых, устанавливается соединение — это позволяет ему, в отличие от IP, гарантировать доставку пакетов, во-вторых - используются порты для обмена пакетами между приложениями, а не просто узлами.

Протокол TCP предназначен для обмена данными — это «надежный» протокол, потому что он во-первых обеспечивает надежную доставку данных, так как предусматривает установления логического соединения; во-вторых, нумерует пакеты и подтверждает их прием квитанцией, а в случае потери организует повторную передачу; в-третьих, делит передаваемый поток байтов на части — сегменты - и передает их нижнему уровню, на приемной стороне снова собирает их в непрерывный поток байтов.

TCP соединение начинается с т.н. «рукопожатия»: узел А посылает узлу В специальный пакет SYN — приглашение к соединению; В отвечает пакетом SYN-ACK — согласием об установлении соединения; А посылает пакет ACK — подтверждение, что согласие получено.

После этого TCP соединение считается установленным, и приложения, работающие в этих узлах, могут посылать друг другу пакеты с данными.

«Соединение» означает, что узлы помнят друг о друге, нумеруют все пакеты, идущие в обе стороны, посылают подтверждения о получении каждого пакета и перепосылают потерявшиеся по дороге пакеты. Для узла А это соединение называется исходящим, а для узла В - входящим. Любое установленное TCP соединение симметрично, и пакеты с данными по нему всегда идут в обе стороны.

В отличие от традиционной альтернативы - UDP, который может сразу же начать передачу пакетов, TCP устанавливает соединения, которые должны быть созданы перед передачей данных. TCP соединение можно разделить на 3 стадии:

1. Установка соединения.
2. Передача данных.
3. Завершение соединения.

## Протокол UDP

Протокол UDP (User Datagram Protocol) является одним из основных протоколов, расположенных непосредственно над IP. Он предоставляет прикладным процессам транспортные услуги, немногим отличающиеся от услуг протокола IP. Протокол UDP обеспечивает доставку дейтограмм, но не требует подтверждения их получения. Протокол UDP не требует соединения с удаленным модулем UDP ("бессвязный" протокол). К заголовку IP-пакета UDP добавляет поля порт отправителя и порт получателя, которые обеспечивают мультиплексирование информации между различными прикладными процессами, а также поля длина UDP-дейтограммы и контрольная сумма, позволяющие поддерживать целостность данных. Таким образом, если на уровне IP для определения места доставки пакета используется адрес, на уровне UDP - номер порт.

Протокол UDP ориентирован на транзакции, получение датаграмм и защита от дублирования не гарантированы. Приложения, требующие гарантированного получения потоков данных, должны использовать протокол управления пересылкой (Transmission Control Protocol - TCP).

UDP - минимальный ориентированный на обработку сообщений протокол транспортного уровня, задокументированный в RFC 768. UDP не предоставляет никаких гарантий доставки сообщения для протокола верхнего уровня и не сохраняет состояния отправленных сообщений.

UDP обеспечивает многоканальную передачу (с помощью номеров портов) и проверку целостности (с помощью контрольных сумм) заголовка и существенных данных. Надежная передача в случае необходимости должна реализовываться пользовательским приложением.

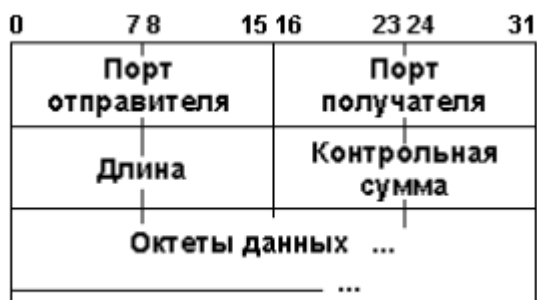


Рис. 4.1. Формат заголовка для датаграмм клиента

Заголовок UDP (рис. 4.1) состоит из четырех полей, каждое по 2 байта (16 бит). Два из них необязательны к использованию в IPv4, в то время как в IPv6 необязателен только порт отправителя.

В поле **Порт отправителя** указывается номер порта отправителя. Предполагается, что это значение задает порт, на который при необходимости будет посылаться ответ. В противном же случае, значение должно быть равным 0.

Поле **Порт получателя** обязательно и содержит порт получателя.

Поле **Длина датаграммы** задает длину всей датаграммы (заголовка и данных) в байтах. Минимальная длина равна длине заголовка – 8 байт. Теоретически, максимальный размер поля – 65535 байт для UDP-датаграммы (8 байт на заголовок и 65527 на данные). Фактический предел для длины данных при использовании IPv4 – 65507 (помимо 8 байт на UDP-заголовок требуется еще 20 на IP-заголовок).

Поле контрольной суммы используется для проверки заголовка и данных на ошибки. Если сумма не сгенерирована передатчиком, то поле заполняется нулями. Поле является обязательным для IPv6.

Из-за недостатка надежности, приложения UDP должны быть готовыми к некоторым потерям, ошибкам и дублированиям. Некоторые из них (например, TFTP) могут при необходимости добавить элементарные механизмы обеспечения надежности на прикладном уровне.

Но чаще такие механизмы не используются UDP-приложениями и даже мешают им. Потокковые медиа, многопользовательские игры в реальном времени и VoIP - примеры приложений, часто использующих протокол UDP. В этих конкретных приложениях потеря пакетов обычно не является большой проблемой. Если приложению необходим высокий уровень надежности, то можно использовать другой протокол.

Более серьезной потенциальной проблемой является то, что в отличие от TCP, основанные на UDP приложения не обязательно имеют хорошие механизмы контроля и избежания перегрузок. Чувствительные к перегрузкам UDP-приложения, которые потребляют значительную часть доступной пропускной способности, могут поставить под угрозу стабильность в Интернете.

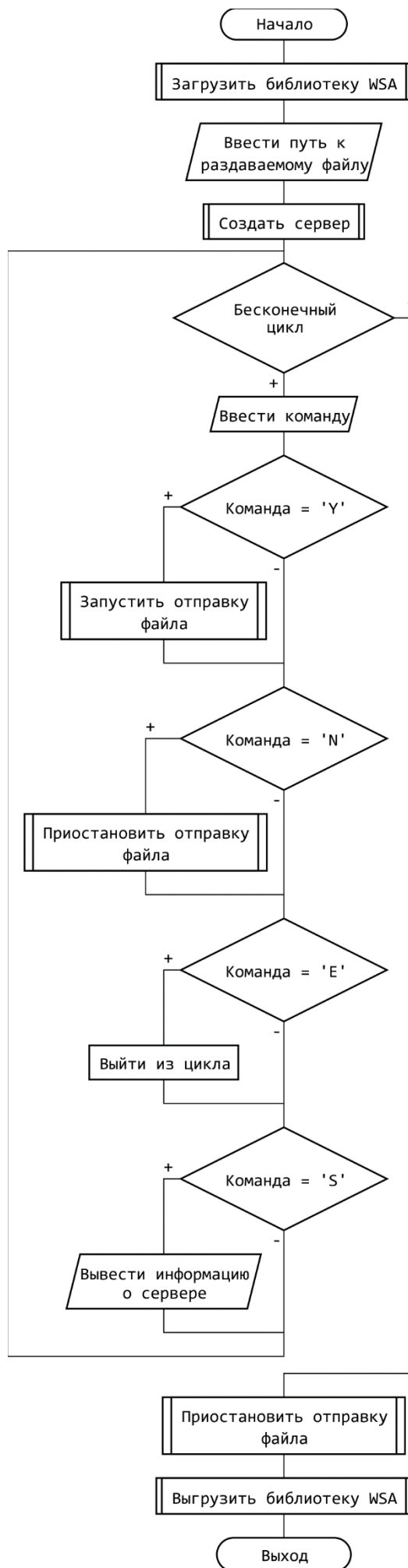
## **Основные функции API для работы с протоколом IPX**

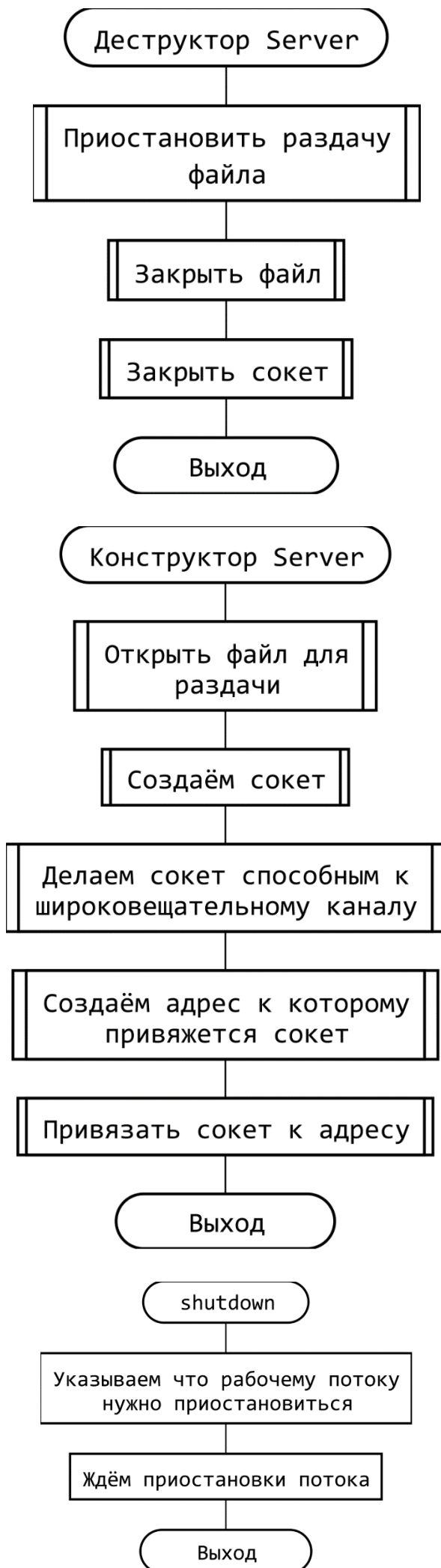
Инициализация Winsock, получение кодов ошибок, очистка памяти, создание и инициализация сокетов и т.д. происходит аналогично протоколу IP. Более подробно рассмотрим функции, необходимые для работы с TCP и UDP.

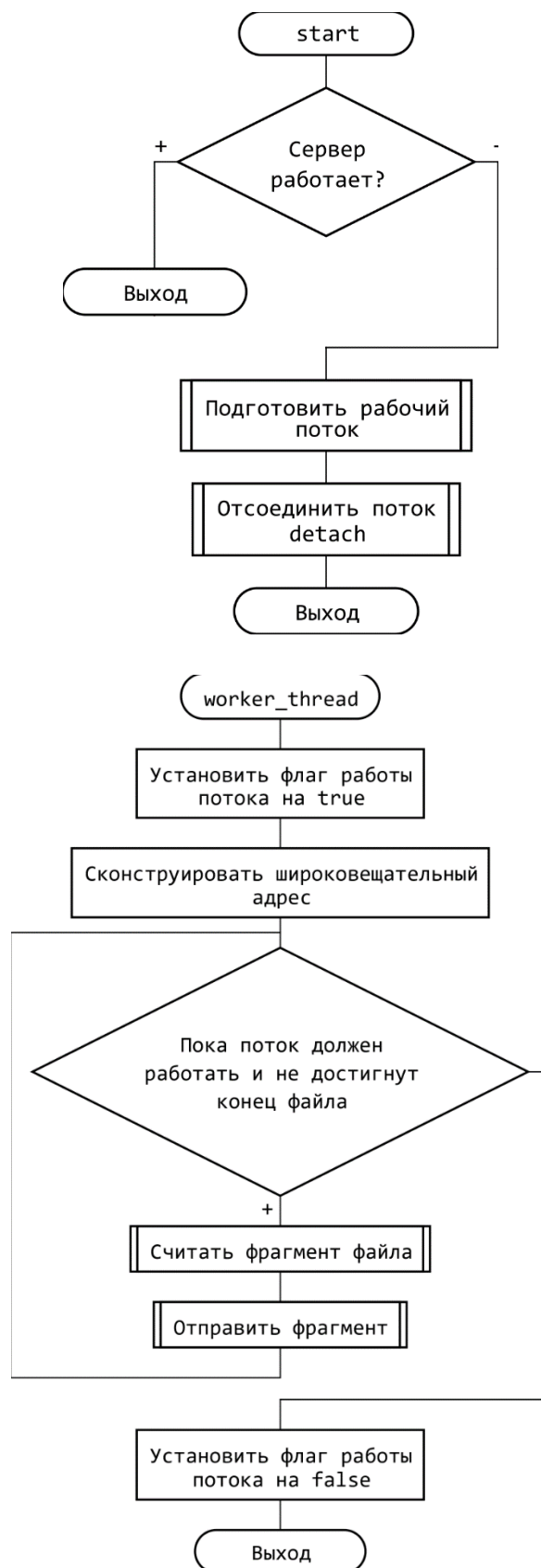
- Перевод сокета в состояние “прослушивания” (для TCP) осуществляется функцией `listen (SOCKET s, int backlog)`, где `s` – дескриптор сокета; `backlog` – максимальный размер очереди входящих сообщений на соединение. Используется сервером, чтобы информировать ОС, что он ожидает (“слушает”) запросы связи на данном соке. Без этой функции всякое требование связи с сокетом будет отвергнуто.
- Функция `connect (SOCKET s, const struct sockaddr FAR* name, int namelen)` нужна для соединения с сокетом, находящимся в состоянии “прослушивания” (для TCP). Она используется процессом-клиентом для установления связи с сервером. В случае успешного установления соединения `connect` возвращает 0, иначе `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.
- Функция `accept (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)` служит для подтверждения запроса на соединение (для TCP). Функция используется для принятия связи на сокет. Сокет должен быть уже слушающим в момент вызова функции. Если сервер устанавливает связь с клиентом, то данная функция возвращает новый сокет-дескриптор, через который и производит общение клиента с сервером. Пока устанавливается связь клиента с сервером, функция блокирует другие запросы связи с данным сервером, а после установления связи “прослушивание” запросов возобновляется.

- В случае автоматического распределения адресов и портов узнать какой адрес и порт присвоен сокету можно при помощи функции `getsockname (SOCKET s, struct sockaddr FAR* name, int FAR* namelen)`. Если операция выполнена успешно, возвращает 0, иначе возвращает `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.
- Для передачи данных по протоколу UDP используется функция `sendto (SOCKET s, const char FAR * buf, int len, int flags, const struct sockaddr FAR * to, int tolen)`. Если операция выполнена успешно, возвращает количество переданных байт, иначе возвращает `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.
- Для передачи данных по протоколу TCP используется функция `send (SOCKET s, const char FAR * buf, int len, int flags)`, где `s` - дескриптор сокета; `buf` - указатель на буфер с данными, которые необходимо переслать; `len` - размер (в байтах) данных, которые содержатся по указателю `buf`; `flags` - совокупность флагов, определяющих, каким образом будет произведена передача данных. Если операция выполнена успешно, возвращает количество переданных байт, иначе возвращает `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.
- Для приема данных по протоколу UDP используется функция `recvfrom (SOCKET s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen)`. Если операция выполнена успешно, возвращает количество полученных байт, иначе возвращает `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.
- Для приема данных по протоколу TCP используется функция `recv (SOCKET s, char FAR* buf, int len, int flags)`. Если операция выполнена успешно, возвращает количество полученных байт, иначе возвращает `SOCKET_ERROR` и номер ошибки можно получить при помощи функции `WSAGetLastError`.

### **Разработка программы. Блок-схемы программы. UDP-сервер**

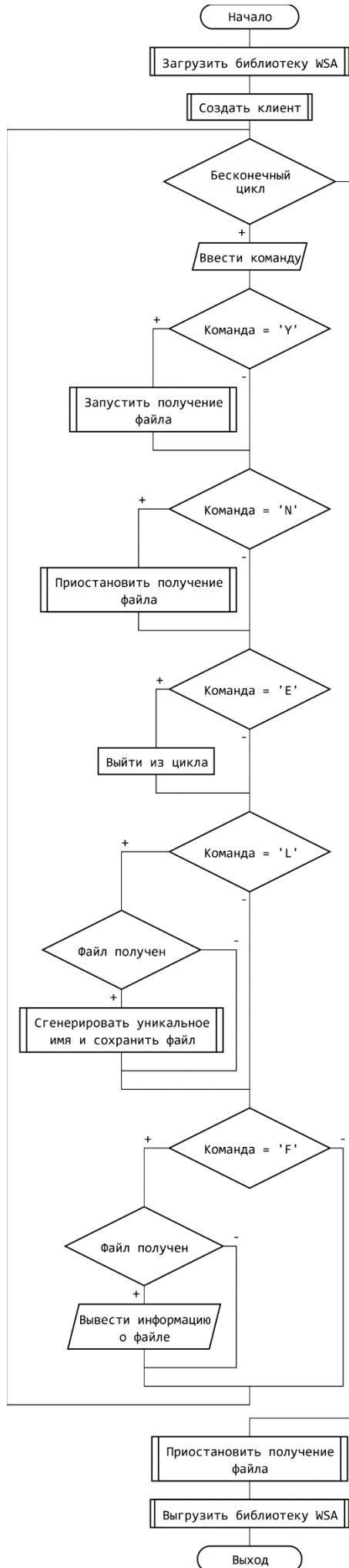


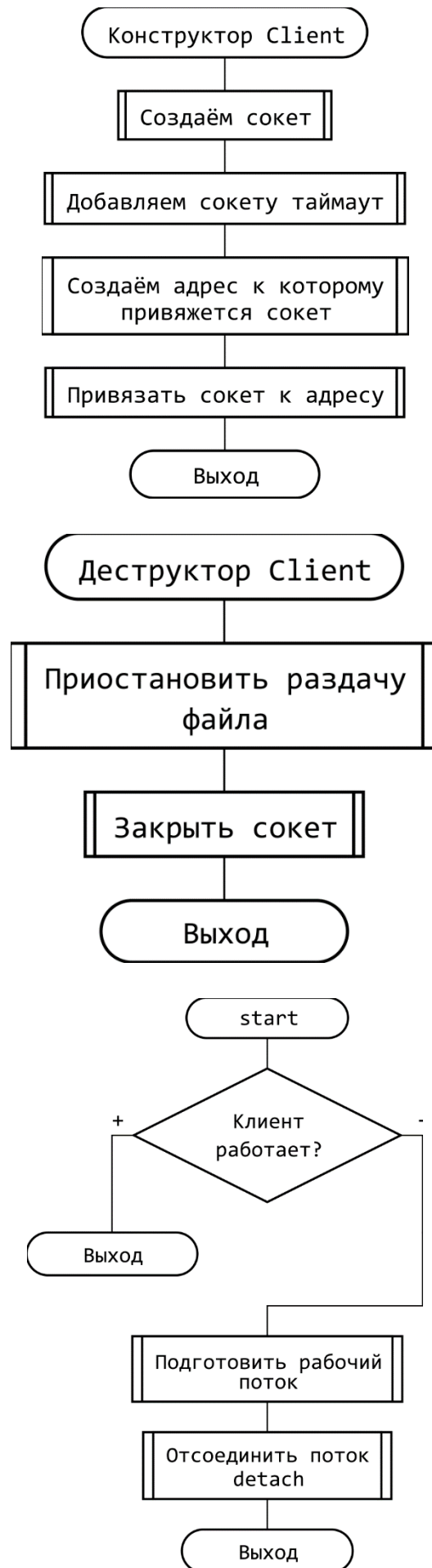


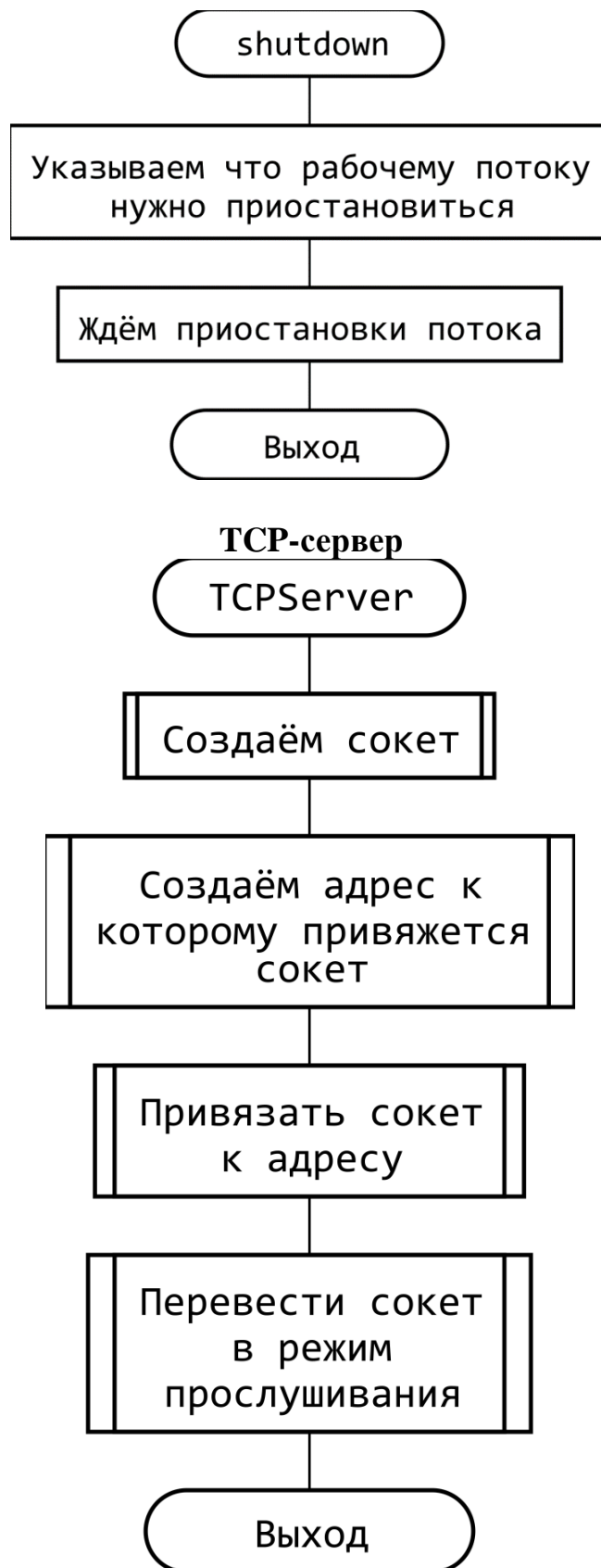


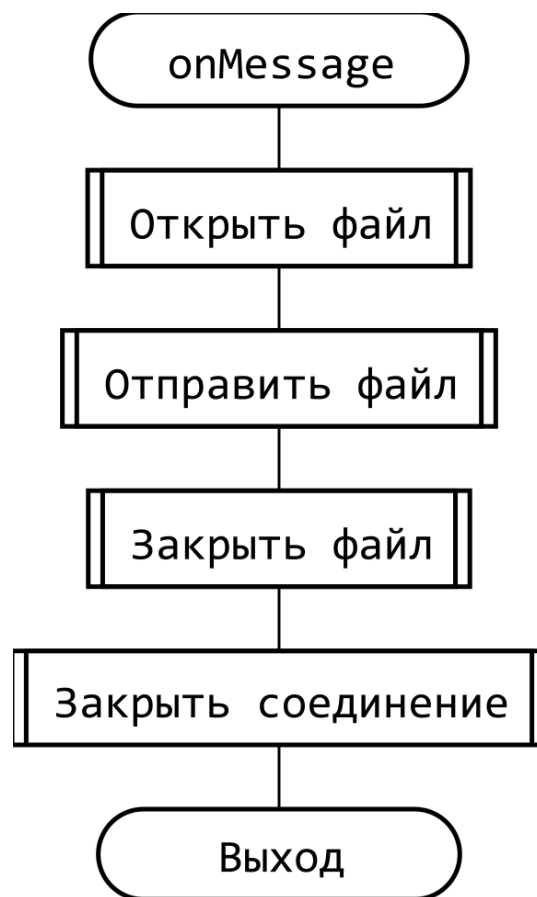


# UDP-клиент

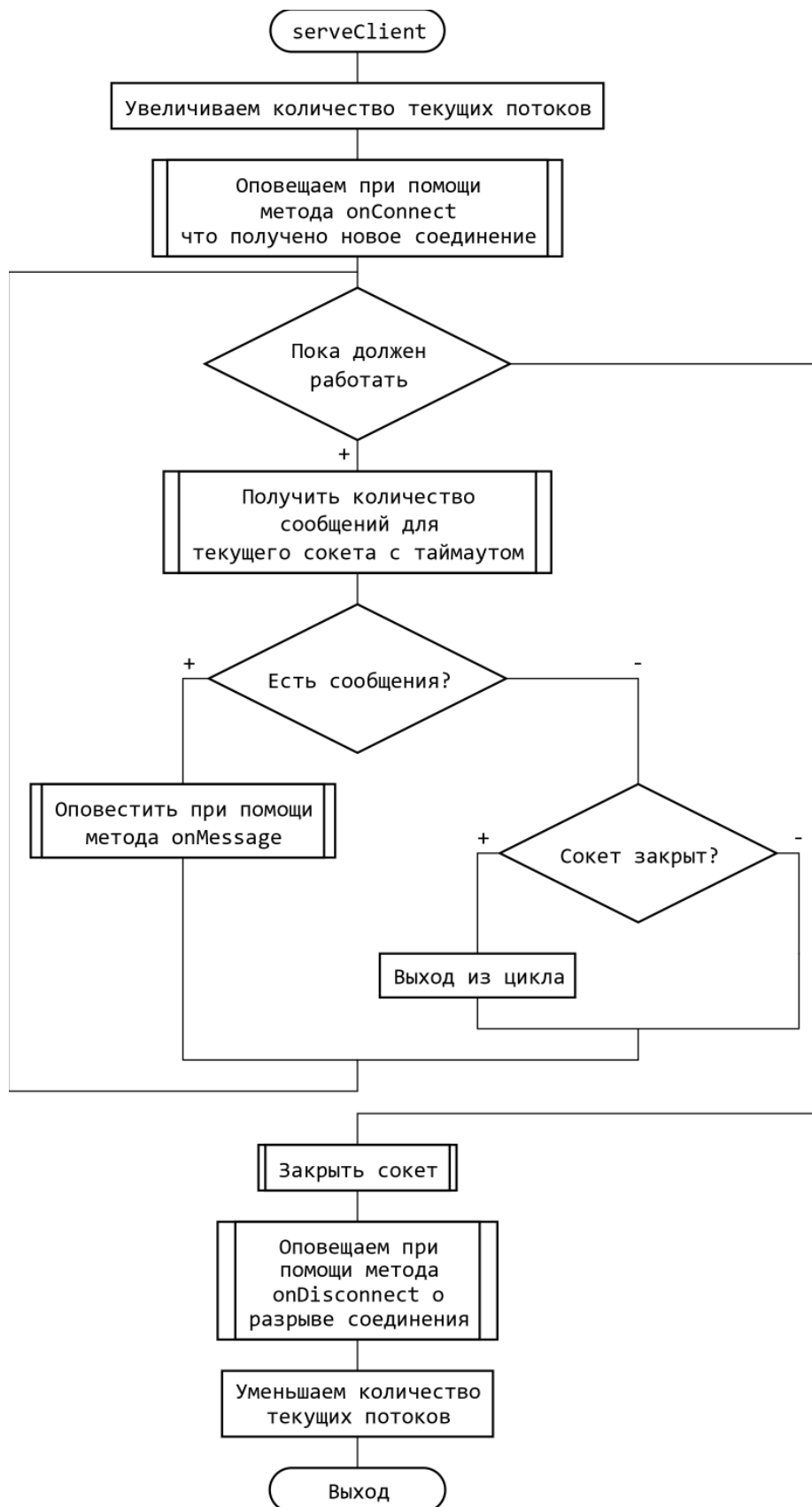


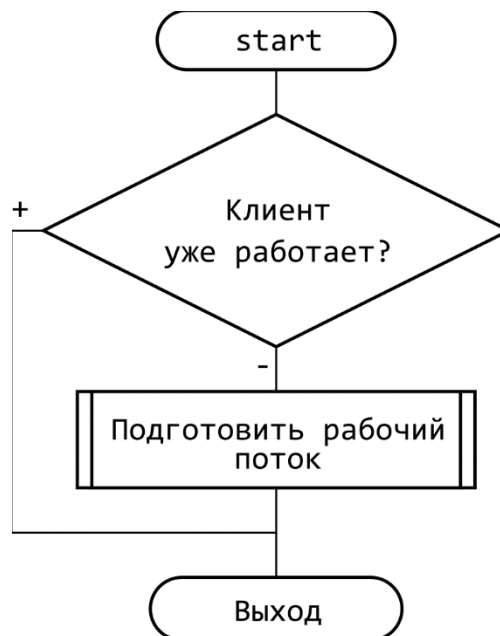




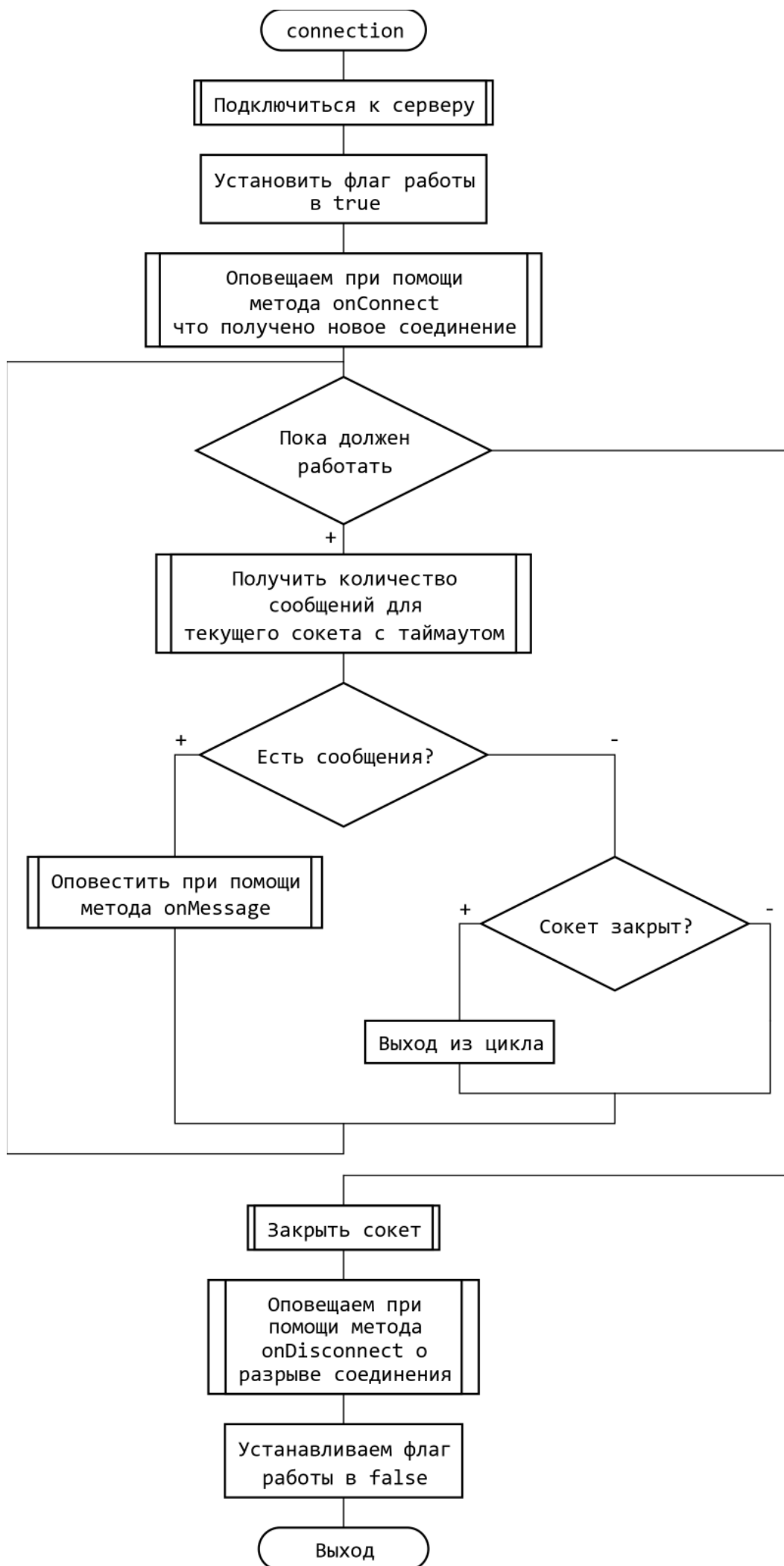




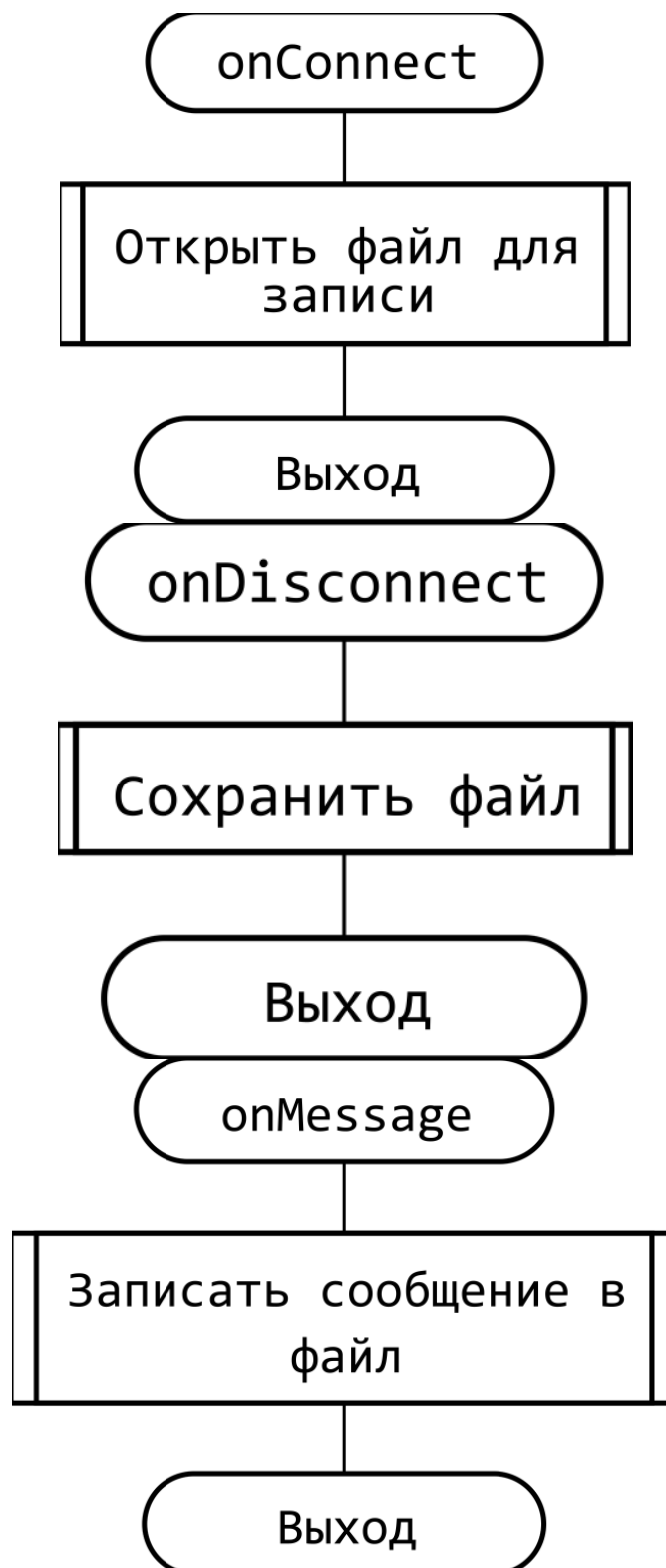


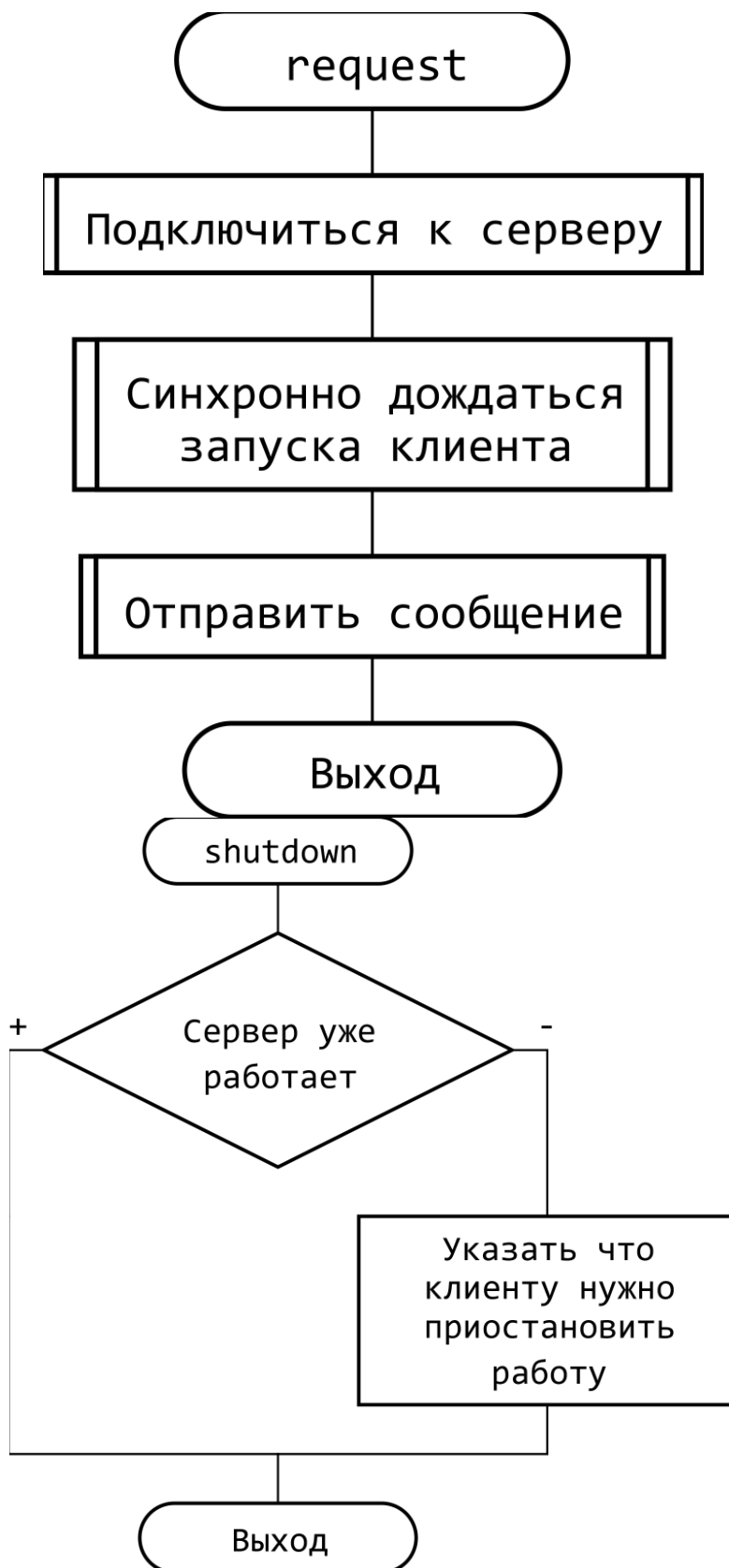


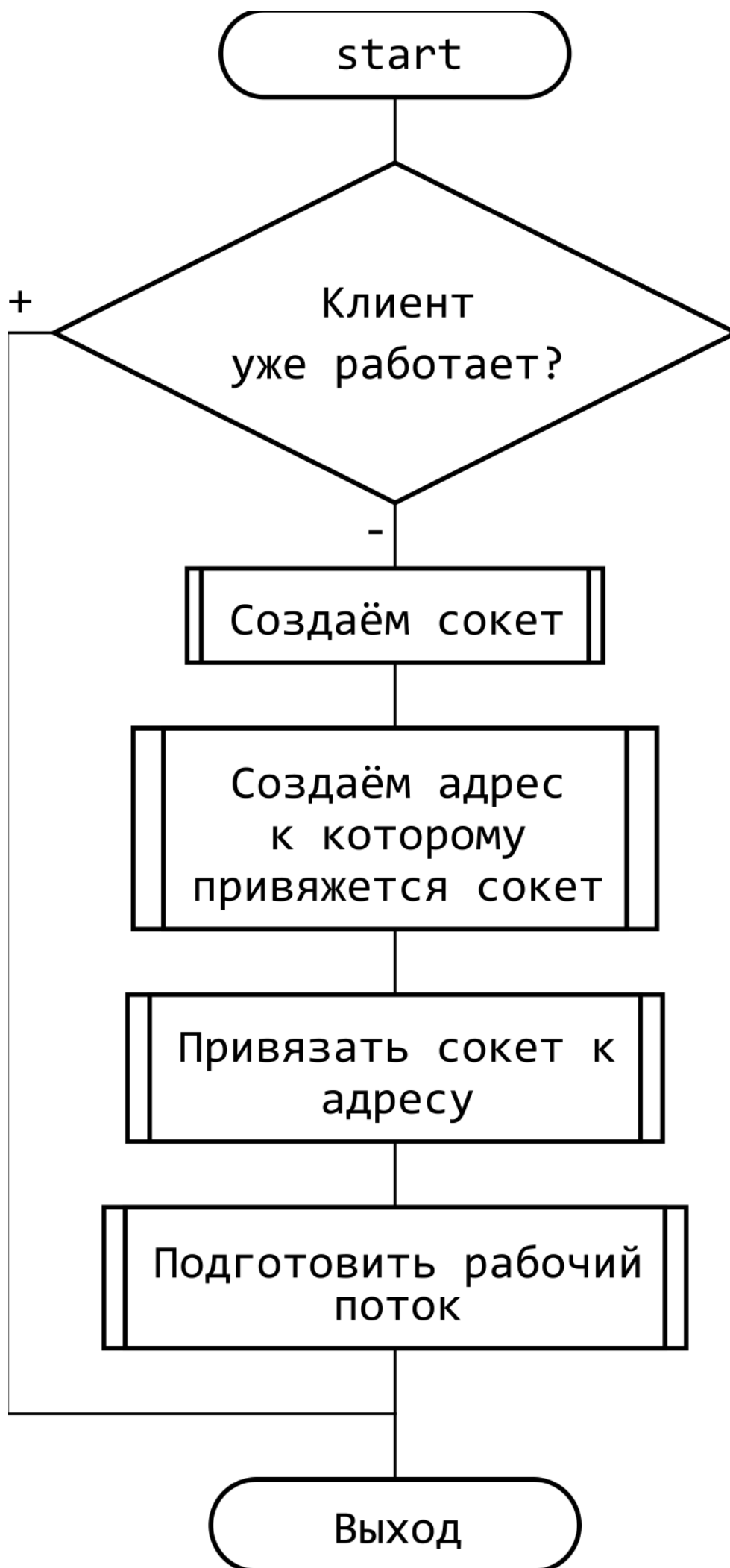
**ТСР-клиент**











## Анализ функционирования программ

### UDP

	Время, сек.
№1	9,860
№2	9,776
№3	10,006
№4	10,114
№5	9,802
Среднее	9,912
Дисперсия	0,021

Размер файла, МиБ
593,04101562500000

Скорость передачи, Мбит/с
478,66420364379800

	Объём (Virtual Box), МиБ		Объём (ноутбук), МиБ
№1	144,77943801879800		6,33197784423828
№2	124,54181671142500		5,26857376098632
№3	148,39307403564400		5,85187149047851
№4	123,90086746215800		6,45842361450195
№5	153,90678787231400		6,29322052001953
№6	139,83787918090800		
№7	141,66141128540000		
№8	150,49905014038000		
Среднее	140,94004058837800		6,04081344604492
Средние потери, %	76%		99%

### TCP

	Время, сек.
№1	10,646
№2	20,646
№3	10,026
№4	19,401
№5	17,353
Среднее	15,614
Дисперсия	24,648

Размер файла, МиБ
593,04101562500000

Скорость передачи, Мбит/с
303,84312629699700

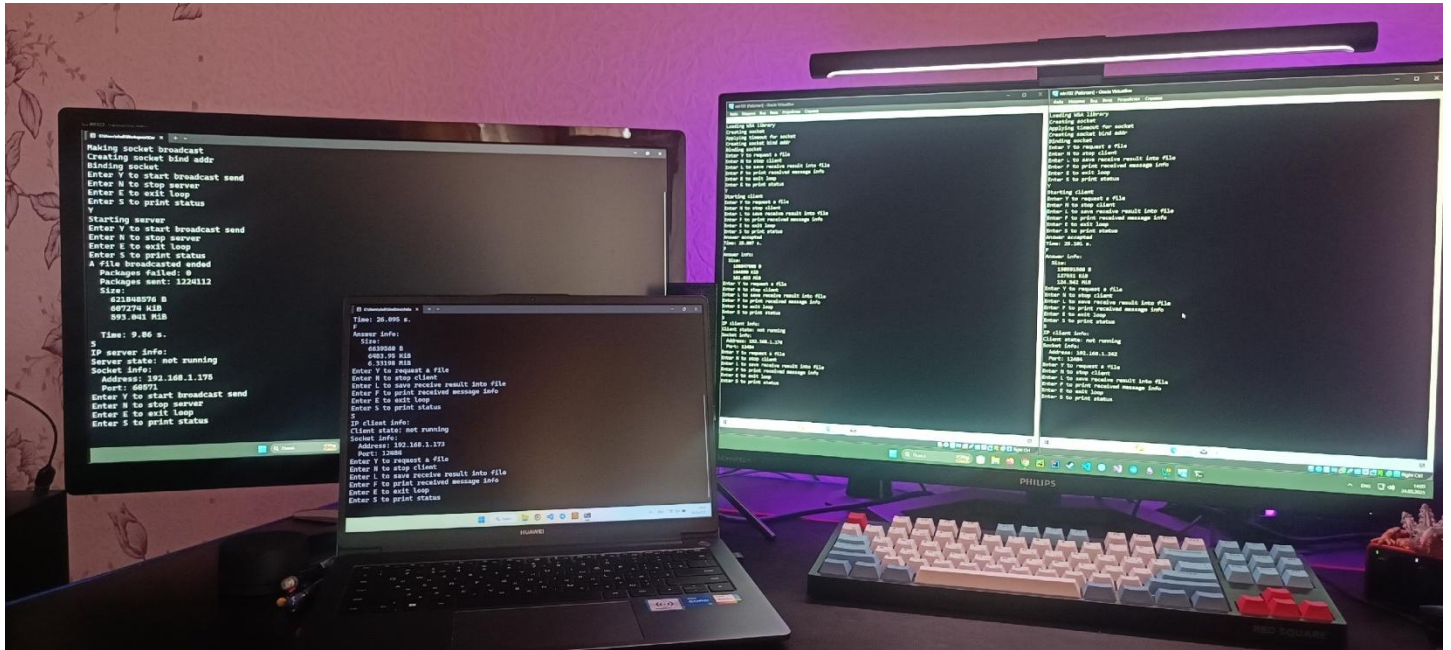
Применение протокола UDP позволяет добиться более высокой скорости (примерно на 58% выше) а также передача при помощи UDP более стабильна и имеет меньший разброс. Слабые стороны протокола UDP перекрывают плюсы TCP. Если мы посмотрим на то сколько данных было принято при помощи UD, то для устройств со стабильной сетью этот параметр составит примерно 24%, а для нестабильной – 1%. Данные же, переданные при помощи протокола TCP, были доставлены полностью и без ошибок, здесь этот параметр составляет 100%. Минусами же TCP становится его относительно меньшая скорость передачи, а также большой разброс во времени передачи файла – его трудно предсказать.

**Вывод:** в ходе лабораторной изучили протоколы TCP/UDP, основные функции библиотеки Winsock и составить программу для приема/передачи пакетов.

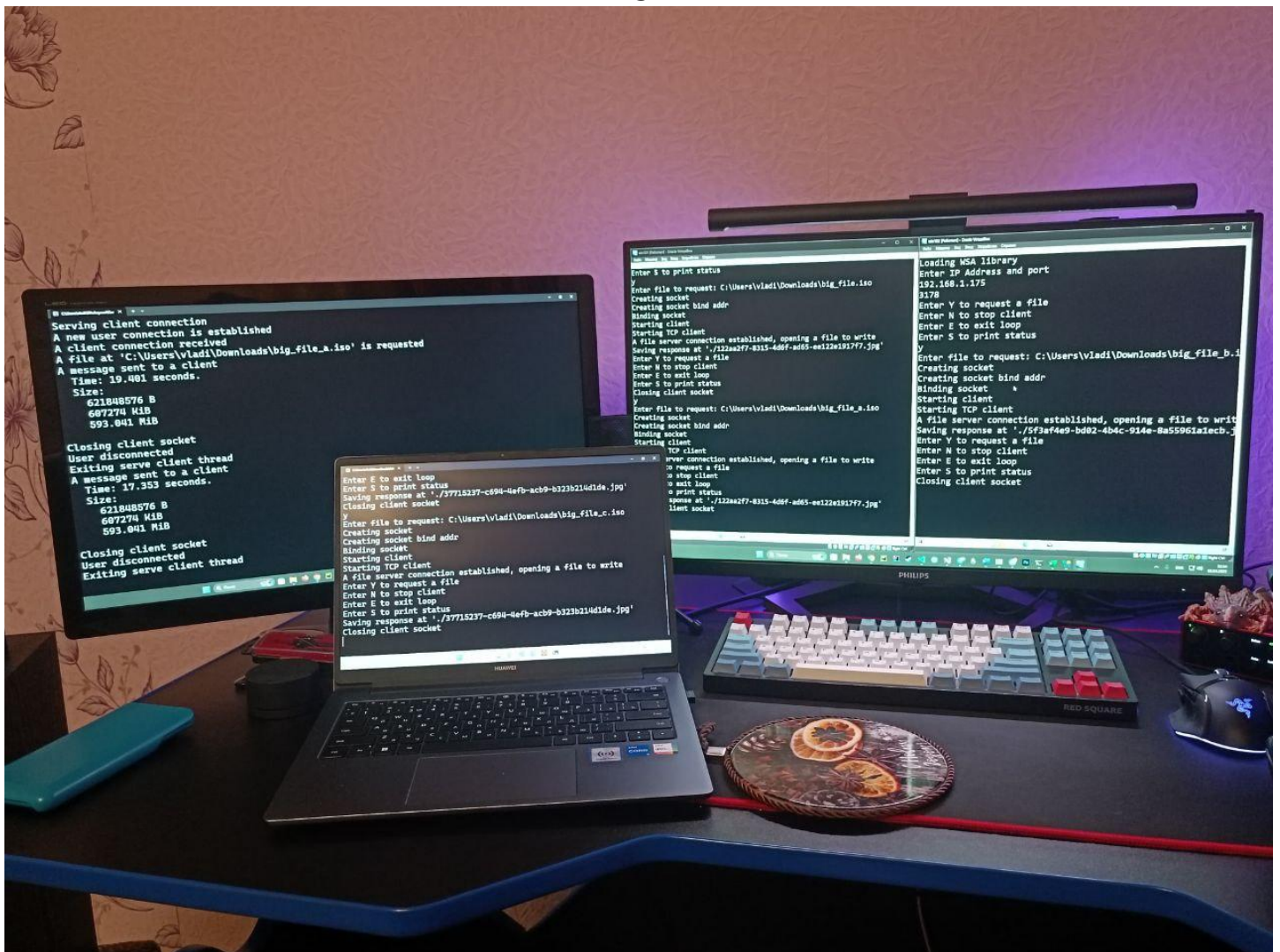
**Текст программ. Скриншоты программ.**

Ссылка на репозиторий с кодом: [https://github.com/IAmProgrammist/comp\\_net/tree/lab4](https://github.com/IAmProgrammist/comp_net/tree/lab4)








# UDP



# TCP





Исходное изображение	Получено на виртуальной машине (локальное подключение по Ethernet-кабелю)		Получено на ноутбуке (Wi-Fi подключение)	
				
				

```
#include <iostream>
#include <WinSock2.h>
#include <cstdio>
#include <fstream>
#include "client.h"
#include "../shared.h"

Client::Client() {
    std::clog << "Creating socket" << std::endl;
    // Создаём сокет
    this->socket_descriptor = socket(
        AF_INET,
        SOCK_DGRAM,
        IPPROTO_IP
    );

    std::clog << "Applying timeout for socket" << std::endl;
    // Добавляем сокету таймаут
    int timeout_ms = 10000;
    if (setsockopt(this->socket_descriptor, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout_ms,
        sizeof(timeout_ms)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to make socket
broadcast"));

    std::clog << "Creating socket bind addr" << std::endl;
    // Создаём адрес к которому привяжется сокет
    sockaddr_in bind_addr;
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr = getDeviceAddrInfo().sin_addr;
    bind_addr.sin_port = htons(CLIENT_DEFAULT_PORT);

    std::clog << "Binding socket" << std::endl;
    // Привязать сокет к адресу
    if (bind(this->socket_descriptor, (sockaddr*)&bind_addr, sizeof(bind_addr)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to bind socket
descriptor"));
}
```

```

Client::~~Client() {
    std::clog << "Stopping worker thread" << std::endl;
    // Приостанавливаем рабочий поток
    this->shutdown();

    std::clog << "Closing socket" << std::endl;
    // Закрываем сокет
    if (closesocket(this->socket_descriptor) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to close socket"));
}

void Client::request(char* payload, int payload_size) {
    // Если клиент уже работает, выходим из него
    if (this->running) {
        std::clog << "Client is already running" << std::endl;
        return;
    }

    std::clog << "Starting client" << std::endl;
    // Подготавливаем рабочий поток
    delete this->current_runner;
    this->should_run = true;
    this->temporary_data.clear();
    this->current_runner = new std::thread([this]() {
        // Устанавливаем флаг работы потока на true
        this->running = true;

        char buffer[IMAGE_FRAGMENT_SIZE];
        int bytes_received;

        auto a = std::chrono::high_resolution_clock::now();

        // Получаем сегменты и сохраняем их в буфер
        while (should_run && (bytes_received = recvfrom(
            this->socket_descriptor,
            buffer,
            sizeof(buffer),
            0,
            nullptr,
            nullptr)) != SOCKET_ERROR) {
            this->temporary_data.insert(this->temporary_data.end(), buffer, buffer +
sizeof(buffer));
        }

        auto b = std::chrono::high_resolution_clock::now();

        std::clog << "Answer accepted\n" <<
            "Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(b - a).count() /
1000.0 << " s." << std::endl;

        // Устанавливаем флаг работы потока на false
        this->running = false;
    });

    this->current_runner->detach();
}

bool Client::isReady() {
    return !this->running;
}

const std::vector<char>& Client::getAnswer() {
    return this->temporary_data;
}

void Client::wait_for_client_stop() {
    while (this->running) {

```

```

    }
}

void Client::shutdown() {
    if (!this->running) {
        std::clog << "Client is already stopped" << std::endl;
        return;
    }

    std::clog << "Stopping client" << std::endl;
    // Указываем что клиенту нужно приостановиться
    // и ждём пока он остановится
    this->should_run = false;
    wait_for_client_stop();
    delete this->current_runner;
}

std::ostream& Client::printClientInfo(std::ostream& out) {
    sockaddr_in client_address;
    int client_address_size = sizeof(client_address);
    int get_sock_name_res = getsockname(this->socket_descriptor, (sockaddr*)&client_address,
&client_address_size);

    out << "IP client info:\n" <<
        "Client state: " << (this->running ? "running" : "not running") << "\n";

    if (get_sock_name_res == SOCKET_ERROR) {
        out << "Unable to get socket info\n";
    }
    else {
        out << "Socket info:\n";
        printSockaddrInfo(out, client_address);
        out << "\n";
    }

    out.flush();

    return out;
}

std::ostream& Client::printAnswerInfo(std::ostream& out) {
    out << "Answer info:\n" << "  Size:\n";
    out << "    " << this->temporary_data.size() << " B\n";
    out << "    " << this->temporary_data.size() / 1024.0 << " KiB\n";
    out << "    " << this->temporary_data.size() / 1024.0 / 1024.0 << " MiB\n";

    out.flush();

    return out;
}

```

---

```
#pragma once
```

```

#include <vector>
#include <thread>
#include "iclient.h"

```

```

class Client : public IClient {
    void wait_for_client_stop();
protected:
    std::atomic<bool> running = false;
    std::atomic<bool> should_run = false;
    SOCKET socket_descriptor;
    std::vector<char> temporary_data;
    std::thread* current_runner = nullptr;

```



```
public:
    // Создаёт клиента
    Client();
    // Освобождает ресурсы клиента
    virtual ~Client();
    // Приостановить работу клиента
    void shutdown();
    // Запрашивает данные с клиента с дополнительной отправкой данных.
    // payload и payload_size игнорируется в данной реализации
    void request(char* payload, int payload_size);
    // Возвращает true если ответ принят
    bool isReady();
    // Возвращает true если ответ принят
    const std::vector<char>& getAnswer();
    // Отобразить информацию о клиенте
    std::ostream& printClientInfo(std::ostream& out);
    // Отобразить информацию о файле
    std::ostream& printAnswerInfo(std::ostream& out);
};
```

---

```
#pragma once

#include <sstream>
#include <WinSock2.h>
#include <iostream>
#include "iclient.h"
#include "../shared.h"

void IClient::init() {
    std::clog << "Loading WSA library" << std::endl;
    loadWSA();
}

void IClient::detach() {
    std::clog << "Unloading WSA library" << std::endl;
    unloadWSA();
}

IClient::IClient() {}

IClient::~IClient() {}

void IClient::request() {
    return this->request(nullptr, 0);
}
```

---

```
#pragma once

#include <string>
#include <vector>

class IClient {
public:
    // Создаёт клиента
    IClient();
    // Освобождает ресурсы клиента
    virtual ~IClient();
    // Приостановить работу клиента
    virtual void shutdown() = 0;
    // Запрашивает данные с клиента
    void request();
    // Запрашивает данные с клиента с дополнительной отправкой данных
```

```

virtual void request(char* payload, int payload_size) = 0;
// Возвращает true если ответ принят
virtual bool isReady() = 0;
// Возвращает true если ответ принят
virtual const std::vector<char>& getAnswer() = 0;
// Отобразить информацию о клиенте
virtual std::ostream& printClientInfo(std::ostream& out) = 0;
// Отобразить информацию о файле
virtual std::ostream& printAnswerInfo(std::ostream& out) = 0;

// Загружает библиотеку WinSock
static void init();
// Выгружает библиотеку WinSock
static void detach();
};

```

```

#include <iostream>
#include <WinSock2.h>
#include <algorithm>
#include "../shared.h"
#include "client.h"

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

int main() {
    try {
        // Инициализация библиотеки WSA
        IClient::init();

        // Создать клиент
        IClient* c = new Client();

        // Оставляем клиент работать, пока пользователь не решит его приостановить
        std::string input;
        while (true)
        {
            std::cout << "Enter Y to request a file\n"
                << "Enter N to stop client\n"
                << "Enter L to save receive result into file\n"
                << "Enter F to print received message info\n"
                << "Enter E to exit loop\n"
                << "Enter S to print status" << std::endl;

            std::cin >> input;
            std::transform(input.begin(), input.end(), input.begin(), toupper);
            if (input == "Y") {
                // Запустить клиент
                c->request();
            }
            else if (input == "N") {
                // Приостановить клиент
                c->shutdown();
            }
            else if (input == "E") {
                // Выход из цикла
                break;
            }
            else if (input == "L") {
                // Сохранить файл если он был успешно получен
                if (!c->isReady())
                    std::cout << "Client response is not ready yet" << std::endl;

                std::string save_path = getUniqueFilepath();
                saveByteArray(c->getAnswer(), save_path);
                std::cout << "A response was saved at '" << save_path << "'" << std::endl;
            }
        }
    }
}

```

```

    }
    else if (input == "F") {
        // Вывести информацию о файле если он был успешно получен
        if (!c->isReady())
            std::cout << "Client response is not ready yet" << std::endl;

        c->printAnswerInfo(std::cout);
    }
    else if (input == "S") {
        // Вывести информацию о клиенте
        c->printClientInfo(std::cout);
    }
}

// Приостановить сервер
c->shutdown();

delete c;
}
catch (const std::runtime_error& error) {
    std::cerr << "Failed while running server. Caused by: '" << error.what() << "'" <<
std::endl;

    return -1;
}

// Выгрузка библиотеки WSA
IClient::detach();

return 0;
}

```

```

#include <sstream>
#include <WinSock2.h>
#include <WS2tcpip.h>
#include <filesystem>
#include <random>
#include <fstream>
#include "shared.h"

namespace uuid {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_int_distribution<> dis(0, 15);
    static std::uniform_int_distribution<> dis2(8, 11);

    std::string generate_uuid_v4() {
        std::stringstream ss;
        int i;
        ss << std::hex;
        for (i = 0; i < 8; i++) {
            ss << dis(gen);
        }
        ss << "-";
        for (i = 0; i < 4; i++) {
            ss << dis(gen);
        }
        ss << "-4";
        for (i = 0; i < 3; i++) {
            ss << dis(gen);
        }
        ss << "-";
        ss << dis2(gen);
        for (i = 0; i < 3; i++) {
            ss << dis(gen);
        }
    }
}

```

```

    }
    ss << "-";
    for (i = 0; i < 12; i++) {
        ss << dis(gen);
    };
    return ss.str();
}
}

std::string getErrorTextWithWSAErrorCode(std::string errorText) {
    std::ostringstream resultError;
    resultError << errorText << " " << WSAGetLastError();

    return resultError.str();
}

std::ostream& printSockaddrInfo(std::ostream& out, sockaddr_in& sock) {
    char address[64] = {};
    inet_ntop(AF_INET, &sock.sin_addr, address, sizeof(address));

    out << "  Address: " << address << "\n" <<
        "  Port: " << htons(sock.sin_port);

    return out;
}

sockaddr_in getDeviceAddrInfo() {
    // Получить имя текущего устройства
    char host_name[256] = {};
    if (gethostname(host_name, sizeof(host_name)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to get host name"));

    addrinfo hints = {};

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    struct addrinfo* result = NULL;

    // Получить информацию об адресах на устройстве в сети
    DWORD dwRetVal = getaddrinfo(host_name, "", &hints, &result);
    if (dwRetVal != 0) {
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Getaddrinfo failed for device host
name"));
    }

    for (addrinfo* ptr = result; ptr != NULL; ptr = ptr->ai_next) {
        // Если адрес для устройства является IPv4 адресом, мы нашли нужный адрес, возвращаем его
        if (ptr->ai_family == AF_INET) {
            sockaddr_in device_sockaddr = {};
            memcpy(&device_sockaddr, ptr->ai_addr, sizeof(device_sockaddr));
            // Плохой костыль, нужно было использовать GetAdaptersInfo
            // Иначе не получится найти нужный айпишник с гейтвеем на роутер
            if (device_sockaddr.sin_addr.S_un.S_un_b.s_b3 > 10) {
                continue;
            }

            return device_sockaddr;
        }
    }

    throw std::runtime_error(getErrorTextWithWSAErrorCode("Getaddrinfo failed for device host
name"));
}

void loadWSA() {
    WORD wVersionRequested;

```

```

WSADATA wsaData;
wVersionRequested = MAKEWORD(2, 0);

// Загрузить библиотеку WinSock
if (WSAStartup(wVersionRequested, &wsaData) == SOCKET_ERROR)
    throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to load WSA library"));
}

void unloadWSA() {
    // Выгрузить библиотеку WinSock
    if (WSACleanup() == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to unload WSA library"));
}

std::string getUniqueFilepath() {
    return "." + uuid::generate_uuid_v4() + ".jpg";
}

void saveByteArray(const std::vector<char>& data, std::string path) {
    // Открываем файл
    std::ofstream save_file(path, std::ios::binary);

    if (!save_file.good())
        throw std::runtime_error("Unable to open file '" + path + "' for reading");

    // Пишем массив в файл
    for (int i = 0; i < data.size() / IMAGE_FRAGMENT_FILE_SIZE; i++)
        save_file.write(&data[i * IMAGE_FRAGMENT_FILE_SIZE], sizeof(char) *
IMAGE_FRAGMENT_FILE_SIZE);

    // Сохраняем и закрываем файл
    save_file.flush();
    save_file.close();
}

```

```

#pragma once

#include <string>
#include <WinSock2.h>
#include <vector>

#define SERVER_DEFAULT_PORT    0
#define CLIENT_DEFAULT_PORT    12484
#define IMAGE_FRAGMENT_SIZE    508
#define IMAGE_FRAGMENT_FILE_SIZE 512

// Формирует текст ошибки вместе с WSA кодом
std::string getErrorTextWithWSAErrorCode(std::string errorText);
// Выводит информацию о структуре sockaddr
std::ostream& printSockaddrInfo(std::ostream& out, sockaddr_in& sock);
// Возвращает IP адрес для текущего ПК
sockaddr_in getDeviceAddrInfo();
// Загружает WSA
void loadWSA();
// Выгружает WSA
void unloadWSA();
// Возвращает неконфликтующее имя для текущего файла
std::string getUniqueFilepath();
// Сохраняет массив байтов в файл
void saveByteArray(const std::vector<char>& data, std::string path);

```

```

#pragma once

#include <sstream>

```

```
#include <WinSock2.h>
#include <iostream>
#include "iserver.h"
#include "../shared.h"

void IServer::init() {
    std::clog << "Loading WSA library" << std::endl;
    loadWSA();
}

void IServer::detach() {
    std::clog << "Unloading WSA library" << std::endl;
    unloadWSA();
}

IServer::IServer() {
}

IServer::~IServer() {
}
```

---

```
#pragma once

#include <string>

// Абстрактный класс для работы с WinSock библиотекой
class IServer {
public:
    // Создаёт сервер
    IServer();
    // Освобождает ресурсы сервера
    virtual ~IServer() = 0;
    // Возобновить работу сервера
    virtual void start() = 0;
    // Приостановить работу сервера
    virtual void shutdown() = 0;
    // Отобразить информацию о сервере
    virtual std::ostream& printServerInfo(std::ostream& out) = 0;

    // Загружает библиотеку WinSock
    static void init();
    // Выгружает библиотеку WinSock
    static void detach();
};
```

---

```
#include <iostream>
#include <algorithm>
#include "server.h"

int main() {
    try {
        // Инициализация библиотеки WSA
        IServer::init();

        // Ввести путь к раздаваемому файлу
        std::cout << "Input path to file to send: ";
        std::cout.flush();
        std::string input;
        std::cin >> input;

        // Создать сервер
        IServer* s = new Server(input);

        // Оставляем сервер работать, пока пользователь не решит его приостановить
        while (true)
```

```

{
    std::cout << "Enter Y to start broadcast send\n"
        << "Enter N to stop server\n"
        << "Enter E to exit loop\n"
        << "Enter S to print status" << std::endl;

    std::cin >> input;
    std::transform(input.begin(), input.end(), input.begin(), toupper);
    if (input == "Y") {
        // Запустить сервер
        s->start();
    }
    else if (input == "N") {
        // Приостановить сервер
        s->shutdown();
    }
    else if (input == "E") {
        // Выход из цикла
        break;
    }
    else if (input == "S") {
        // Вывести информацию о сервере
        s->printServerInfo(std::cout);
    }
}

// Приостановить сервер
s->shutdown();

delete s;
}
catch (const std::runtime_error& error) {
    std::cerr << "Failed while running server. Caused by: '" << error.what() << "'" <<
std::endl;

    return -1;
}

// Выгрузка библиотеки WSA
IServer::detach();

return 0;
}

```

```

#include <exception>
#include <WinSock2.h>
#include <sstream>
#include <iostream>
#include <chrono>
#include "server.h"
#include "../shared.h"

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

Server::Server(std::string file_path, int port) {
    std::clog << "Opening file " << file_path << std::endl;
    // Открываем файл
    this->file = new std::ifstream(file_path, std::ios::binary);
    if (!this->file->is_open())
        throw std::runtime_error("Unable to open file for read " + file_path);

    std::clog << "Creating socket" << std::endl;

    // Создаём сокет
    this->socket_descriptor = socket(

```

```

        AF_INET,
        SOCK_DGRAM,
        IPPROTO_IP
    );

    std::clog << "Making socket broadcast" << std::endl;
    // Делаем сокет способным к широковещательному каналу
    bool broadcast = true;
    if (setsockopt(this->socket_descriptor, SOL_SOCKET, SO_BROADCAST, (char*)&broadcast,
        sizeof(broadcast)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to make socket
broadcast"));

    std::clog << "Creating socket bind addr" << std::endl;
    // Создаём адрес к которому привяжется сокет
    sockaddr_in bind_addr;
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr = getDeviceAddrInfo().sin_addr;
    bind_addr.sin_port = htons(port);

    std::clog << "Binding socket" << std::endl;
    // Привязать сокет к адресу
    if (bind(this->socket_descriptor, (sockaddr*)&bind_addr, sizeof(bind_addr)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to bind socket
descriptor"));
}

Server::~Server() {
    std::clog << "Stopping worker thread" << std::endl;
    // Приостанавливаем рабочий поток
    this->shutdown();

    std::clog << "Closing file" << std::endl;
    // Закрываем файл
    this->file->close();
    delete this->file;

    std::clog << "Closing socket" << std::endl;
    // Закрываем сокет
    if (closesocket(this->socket_descriptor) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to close socket"));
}

std::ostream& Server::printServerInfo(std::ostream& out) {
    sockaddr_in server_address;
    int server_address_size = sizeof(server_address);
    int get_sock_name_res = getsockname(this->socket_descriptor, (sockaddr*)&server_address,
&server_address_size);

    std::cout << "IP server info:\n" <<
        "Server state: " << (this->running ? "running" : "not running") << "\n";

    if (get_sock_name_res == SOCKET_ERROR) {
        std::cout << "Unable to get socket info\n";
    }
    else {
        std::cout << "Socket info:\n";
        printSockaddrInfo(std::cout, server_address);
        std::cout << "\n";
    }

    std::cout.flush();

    return out;
}

void Server::start() {

```



```

// Если сервер уже работает, выходим из него
if (this->running) {
    std::clog << "Server is already running" << std::endl;
    return;
}

// Подготавливаем рабочий поток
delete this->current_runner;

std::clog << "Starting server" << std::endl;
this->should_run = true;

this->current_runner = new std::thread([this]() {
    // Устанавливаем флаг работы потока на true
    this->running = true;

    this->file->clear();
    this->file->seekg(0, std::ios::beg);
    char buffer[IMAGE_FRAGMENT_SIZE];
    int packages_success = 0, packages_failed = 0;

    // Конструируем широковещательный sockaddr_in
    sockaddr_in client_sockaddr;
    client_sockaddr.sin_family = AF_INET;
    client_sockaddr.sin_port = htons(CLIENT_DEFAULT_PORT);
    client_sockaddr.sin_addr = getDeviceAddrInfo().sin_addr;
    memset(((char*)&client_sockaddr.sin_addr) + 3, 0xff, 1);

    int total_bytes = 0;

    auto a = std::chrono::high_resolution_clock::now();

    // Пока поток должен работать и не достигнут конец файла
    while (this->should_run && !this->file->eof()) {
        // Читать файл
        this->file->read(buffer, sizeof(buffer));
        int bytes_read = this->file->gcount();
        total_bytes += bytes_read;

        // Отправить фрагмент
        if (sendto(
            this->socket_descriptor,
            buffer,
            bytes_read,
            0,
            (sockaddr*)&client_sockaddr,
            sizeof(client_sockaddr)) == SOCKET_ERROR) {
            packages_failed++;
        }
        else {
            packages_success++;
        }
    }

    auto b = std::chrono::high_resolution_clock::now();

    std::clog << "A file broadcasted ended\n Packages failed: "
        << packages_failed << "\n Packages sent: " << packages_success << "\n" <<
        " Size: " << "\n" <<
        " " << total_bytes << " B\n" <<
        " " << total_bytes / 1024.0 << " KiB\n" <<
        " " << total_bytes / 1024.0 / 1024.0 << " MiB\n" <<
        "\n Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(b - a).count()
/ 1000.0 << " s." << std::endl;

    // Устанавливаем флаг работы потока на false
    this->running = false;

```

```

});

this->current_runner->detach();
}

void Server::shutdown() {
    if (!this->running) {
        std::clog << "Server is already stopped" << std::endl;
        return;
    }

    std::clog << "Stopping server" << std::endl;
    // Указываем что серверу нужно приостановиться
    // и ждём пока он остановится
    this->should_run = false;
    wait_for_server_stop();
    delete this->current_runner;
}

void Server::wait_for_server_stop() {
    // Используем спинлок, так как пакеты относительно небольшие и сервер должен
    // быстро увидеть что пора заканчивать работу
    while (this->running) {
    }
}
}

```

```

#pragma once

#include <WinSock2.h>
#include <fstream>
#include <thread>
#include <mutex>
#include <atomic>
#include "iserver.h"
#include "../shared.h"

class Server : public IServer {
    void wait_for_server_stop();
protected:
    std::atomic<bool> running = false;
    std::atomic<bool> should_run = false;
    std::ifstream* file = nullptr;
    SOCKET socket_descriptor;
    std::thread* current_runner = nullptr;

public:
    // Создаёт сервер
    Server(std::string file_path, int port = SERVER_DEFAULT_PORT);
    // Освобождает ресурсы сервера
    ~Server();
    // Возобновить работу сервера
    void start();
    // Приостановить работу сервера
    void shutdown();
    // Отобразить информацию о сервере
    std::ostream& printServerInfo(std::ostream& out);
};

```

```

#include <fstream>
#include <webstur/utls.h>
#include <webstur/ip/tcp/tcpclient.h>

class DLLEXPORT FileTCPClient : public TCPClient {

```

```

std::ofstream save_file;
std::string save_path;
public:
    FileTCPClient(std::string address,
                  int port = TCP_SERVER_DEFAULT_PORT,
                  std::string file_path = getUniqueFilepath());
protected:
    // Открывает файл для записи
    void onConnect();
    // Сохраняет полученный файл от сервера
    void onDisconnect();
    // Записывает данные от сервера в файл
    void onMessage(const std::vector<char>& message);
};

```

---

```

// TCP-сервер, принимающий путь от пользователя и отправляющий в ответ файл

```

```

#pragma once

```

```

#include <webstur/ip/tcp/tcpserver.h>

```

```

class DLLEXPORT FileTCPServer : public TCPServer {
protected:
    // Метод, вызываемый при установлении соединения с клиентом
    virtual void onConnect(SOCKET client);
    // Метод, вызываемый при разрыве соединения с клиентом
    virtual void onDisconnect(SOCKET client);
    // Принимает путь от клиента и отправляет файл по этому пути
    void onMessage(SOCKET client, const std::vector<char>& message);
};

```

---

```

#pragma once

```

```

#include <thread>

```

```

#include <webstur/utills.h>

```

```

#include <webstur/iclient.h>

```

```

class DLLEXPORT TCPClient : public IClient {
protected:
    std::atomic<bool>* running = nullptr;
    std::atomic<bool>* should_run = nullptr;
    SOCKET socket_descriptor;
    std::thread* current_runner = nullptr;
    std::string server_address;
    int server_port;

    // Метод, вызываемый при установлении соединения с сервером
    virtual void onConnect() = 0;
    // Метод, вызываемый при разрыве соединения с сервером
    // Переданный сокет уже не является действительным,
    // однако передаётся для отладочной информации
    virtual void onDisconnect() = 0;
    // Метод, вызываемый при отправке получения сообщения от сервера
    virtual void onMessage(const std::vector<char>& message) = 0;
    // Отправляет данные message TCP-серверу
    void sendMessage(std::istream& message);
public:
    // Создаёт сервер
    TCPClient(std::string address, int port = TCP_SERVER_DEFAULT_PORT);
    // Освобождает ресурсы клиента
    ~TCPClient();
    // Приостановить работу клиента
    void shutdown();
    // Устанавливает соединение с сервером

```

```

void start();
// Запрашивает данные с сервера
void request();
// Запрашивает данные с сервера с дополнительной отправкой данных
void request(char* payload, int payload_size);
// Отобразить информацию о клиенте
std::ostream& printClientInfo(std::ostream& out);

private:
// Синхронное ожидание остановки сервера
void waitForClientStop();
// Синхронное ожидание старта сервера
void waitForClientStart();
// Метод для работы с сервером
void connection();
};

```

---

```

// Класс для запуска TCP-сервера, отдаёт обычные файлы

```

```

#pragma once

```

```

#include <atomic>
#include <thread>
#include <semaphore>
#include <sstream>
#include <webstur/utils.h>
#include <webstur/iserver.h>

```

```

class DLLEXPORT TCPServer : public IServer {
protected:
    std::atomic<bool>* running = nullptr;
    std::atomic<bool>* should_run = nullptr;
    SOCKET socket_descriptor;
    std::thread* current_runner = nullptr;
    std::atomic<int>* current_threads_amount = nullptr;

    // Метод, вызываемый при установлении соединения с клиентом
    virtual void onConnect(SOCKET client) = 0;
    // Метод, вызываемый при разрыве соединения с клиентом
    // Переданный сокет уже не является действительным,
    // однако передаётся для отладочной информации
    virtual void onDisconnect(SOCKET client) = 0;
    // Метод, вызываемый при разрыве общения с клиентом
    virtual void onMessage(SOCKET client, const std::vector<char>& message) = 0;
    // Отправляет данные message TCP-клиенту client
    void sendMessage(SOCKET client, std::istream& message);
    // Закрывает соединение
    void disconnect(SOCKET client);
public:
    // Создаёт сервер
    TCPServer(int port = TCP_SERVER_DEFAULT_PORT);
    // Освобождает ресурсы сервера
    ~TCPServer();
    // Возобновить работу сервера
    void start();
    // Приостановить работу сервера
    void shutdown();
    // Отобразить информацию о сервере
    std::ostream& printServerInfo(std::ostream& out);

private:
    // Метод для принятия соединений от клиента
    void serve();
    // Метод для работы с конкретным клиентом
    void serveClient(SOCKET client);

```

```
// Синхронное ожидание остановки сервера
void waitForServerStop();
};
```

```
#include "pch.h"
#include <iostream>
#include <webstur/ip/tcp/file/filetcpclient.h>

FileTCPClient::FileTCPClient(std::string address, int port, std::string save_path):
    TCPClient(address, port), save_path(save_path) {
}

void FileTCPClient::onConnect() {
    std::clog << "A file server connection established, opening a file to write" << std::endl;

    // Открываем файл для записи
    this->save_file = std::ofstream(save_path, std::ios::binary);

    if (!this->save_file.good()) {
        this->shutdown();
        throw std::invalid_argument("Couldn't open file for saving at '" + save_path + "'");
    }
    std::clog << "Saving response at '" << this->save_path << "'" << std::endl;
}

void FileTCPClient::onDisconnect() {
    // Сохраняем файл
    this->save_file.flush();
    this->save_file.close();
}

void FileTCPClient::onMessage(const std::vector<char>& message) {
    // Пишем сообщение в файл
    this->save_file.write(&message[0], message.size());
}
```

```
#include "pch.h"
#include <iostream>
#include <fstream>
#include <filesystem>
#include <webstur/ip/tcp/file/filetcpserver.h>

void FileTCPServer::onMessage(SOCKET client, const std::vector<char>& message) {
    std::string file_path = std::string(message.begin(), message.end());

    std::clog << "A file at '" << file_path << "' is requested" << std::endl;

    // Открыть файл
    std::ifstream read_file(file_path, std::ios::binary|std::ios::ate);
    auto total_bytes = read_file.tellg();
    read_file.seekg(0, std::ios::beg);
    auto start_time = std::chrono::high_resolution_clock::now();
    // Отправить файл
    this->sendMessage(client, read_file);
    auto end_time = std::chrono::high_resolution_clock::now();
    std::clog << "A message sent to a client\n" <<
        "    Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
start_time).count() / 1000.0 << " seconds.\n" <<
        "    Size:\n" <<
        "        " << total_bytes << " B\n" <<
        "        " << total_bytes / 1024.0 << " KiB\n" <<
        "        " << total_bytes / 1024.0 / 1024.0 << " MiB\n" << std::endl;
    // Закрывать файл
    read_file.close();
}
```

```

        // Закрывать соединение
        this->disconnect(client);
    }

    void FileTCPServer::onConnect(SOCKET client) {
        std::clog << "A new user connection is established" << std::endl;
    }

    void FileTCPServer::onDisconnect(SOCKET client) {
        std::clog << "User disconnected" << std::endl;
    }
}

```

```

#include "pch.h"
#include <istream>
#include <iostream>
#include <sstream>
#include <chrono>
#include <WS2tcpip.h>
#include <webstur/ip/tcp/tcpclient.h>

void TCPClient::request(char* payload, int payload_size) {
    // Подключиться к серверу
    this->start();
    // Ждём запуска клиента
    this->waitForClientStart();
    // Отправить сообщение
    auto request_stream = std::stringstream(payload != nullptr ?
        std::string(payload, payload + payload_size) : "");
    this->sendMessage(request_stream);
}

void TCPClient::shutdown() {
    if (!(*this->running)) {
        std::clog << "UDPClient is already stopped" << std::endl;
        return;
    }

    std::clog << "Stopping client" << std::endl;
    // Указываем что клиенту нужно приостановиться
    *this->should_run = false;
}

TCPClient::TCPClient(std::string address, int port) : server_address(address), server_port(port)
{
    this->should_run = new std::atomic<bool>(false);
    this->running = new std::atomic<bool>(false);
}

TCPClient::~~TCPClient() {
    std::clog << "Stopping worker thread" << std::endl;
    // Приостанавливаем рабочий поток и ждём пока он закончит работу
    this->shutdown();
    waitForClientStop();
    delete this->current_runner;

    std::clog << "Closing socket" << std::endl;
    // Закрываем сокет
    if (closesocket(this->socket_descriptor) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to close socket"));

    delete this->should_run;
    delete this->running;
}

void TCPClient::waitForClientStop() {

```

```

    auto start = std::chrono::high_resolution_clock::now();
    while (*this->running) {
        auto new_time = std::chrono::high_resolution_clock::now();

        if (std::chrono::duration_cast<std::chrono::seconds>(new_time - start).count() > 3 *
TCP_SERVER_TIMEOUT_S)
            throw std::runtime_error("Wait timeout reached");
    }
}

void TCPClient::sendMessage(std::istream& message) {
    char buffer[TCP_MAX_MESSAGE_SIZE];

    // Отправить данные клиенту
    while (!message.eof()) {
        message.read(buffer, sizeof(buffer));
        auto bytes_read = message.gcount();

        if (bytes_read == 0) break;

        if (send(this->socket_descriptor, buffer, bytes_read, 0) == SOCKET_ERROR)
            throw std::runtime_error(getErrorTextWithWSAErrorCode("Couldn't send data over
TCP"));
    }
}

void TCPClient::start() {
    // Если клиент уже работает, выходим из него
    if (*this->running) {
        std::clog << "UDPCliient is already running" << std::endl;
        return;
    }

    std::clog << "Creating socket" << std::endl;

    // Создаём сокет
    this->socket_descriptor = socket(
        AF_INET,
        SOCK_STREAM,
        IPPROTO_TCP
    );

    std::clog << "Creating socket bind addr" << std::endl;
    // Создаём адрес к которому привяжется сокет
    sockaddr_in bind_addr;
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr = getDeviceAddrInfo().sin_addr;
    bind_addr.sin_port = 0;

    std::clog << "Binding socket" << std::endl;
    // Привязать сокет к адресу
    if (bind(this->socket_descriptor, (sockaddr*)&bind_addr, sizeof(bind_addr)) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to bind socket
descriptor"));

    std::clog << "Starting client" << std::endl;
    // Подготавливаем рабочий поток
    delete this->current_runner;
    *this->should_run = true;
    this->current_runner = new std::thread(&TCPClient::connection, this);
    this->current_runner->detach();
}

std::ostream& TCPClient::printClientInfo(std::ostream& out) {
    sockaddr_in client_address;
    int client_address_size = sizeof(client_address);

```

```

    int get_sock_name_res = getsockname(this->socket_descriptor, (sockaddr*)&client_address,
&client_address_size);

    out << "IP client info:\n" <<
        "TCPClient state: " << (*this->running ? "running" : "not running") << "\n";

    if (get_sock_name_res == SOCKET_ERROR) {
        out << "Unable to get socket info\n";
    }
    else {
        out << "Socket info:\n";
        printSockaddrInfo(out, client_address);
        out << "\n";
    }

    out.flush();

    return out;
}

void TCPClient::connection() {
    std::clog << "Starting TCP client" << std::endl;
    // Запуск TCP-сервера

    try {
        sockaddr_in bind_addr;
        inet_pton(AF_INET, &this->server_address[0], &(bind_addr.sin_addr));
        bind_addr.sin_family = AF_INET;
        bind_addr.sin_port = htons(this->server_port);

        // Подключаемся к серверу
        if (connect(this->socket_descriptor, (sockaddr*)&bind_addr, sizeof(bind_addr)) ==
SOCKET_ERROR)
            throw std::runtime_error(getErrorTextWithWSAErrorCode("Couldn't connect to server at
" +
                this->server_address + ":" + std::to_string(this->server_port)));

        // Устанавливаем флаг работы в true
        *this->running = true;

        // Оповещаем при помощи метода onConnect что получено новое соединение
        this->onConnect();

        std::vector<char> buffer(0, TCP_MAX_MESSAGE_SIZE);
        FD_SET readfds;
        timeval timeout;
        timeout.tv_sec = TCP_SERVER_TIMEOUT_S;
        timeout.tv_usec = 0;
        bool outer_close = false;
        while (*this->should_run) {
            FD_ZERO(&readfds);
            FD_SET(this->socket_descriptor, &readfds);

            // Получить количество соединений для текущего сокета с таймаутом
            int to_read = select(0, &readfds, nullptr, nullptr, &timeout);
            if (to_read == SOCKET_ERROR) {
                // Если сокет закрыт извне, можно выйти из цикла
                if (WSAGetLastError() == 10038) {
                    outer_close = true;
                    break;
                }

                std::cerr << getErrorTextWithWSAErrorCode("Couldn't select for socket") <<
std::endl;
            }
            else if (to_read > 0) {
                int bytes_read;

```



```

        buffer.resize(TCP_MAX_MESSAGE_SIZE);
        // Если соединения есть, получаем его и запускаем поток для обработки
        if ((bytes_read = recv(
            this->socket_descriptor,
            &buffer[0],
            buffer.size(),
            0
        )) == SOCKET_ERROR) {
            std::cerr << getErrorTextWithWSAErrorCode("Couldnt get answer from a server")
<< std::endl;

            // Связь разорвана, выйти из цикла
            break;
        }
        else if (bytes_read != 0) {
            // Получено сообщение, оповестить при помощи метода onMessage
            buffer.resize(bytes_read);
            this->onMessage(buffer);
        }
        else break;
    }

    std::clog << "Closing client socket" << std::endl;
    // Закрываем сокет
    if (!outer_close && closesocket(this->socket_descriptor) == SOCKET_ERROR)
        std::cerr << getErrorTextWithWSAErrorCode("Unable to close client socket");
}
catch (...) {
    std::cerr << "An error occured while handling server connection" << std::endl;
}

// Оповещаем при помощи метода onDisconnect о разрыве соединения
this->onDisconnect();

// Устанавливаем флаг работы в false
*this->running = false;
*this->should_run = false;
}

void TCPClient::waitForClientStart() {
    auto start = std::chrono::high_resolution_clock::now();
    while (!(*this->running)) {
        auto new_time = std::chrono::high_resolution_clock::now();

        if (std::chrono::duration_cast<std::chrono::seconds>(new_time - start).count() > 3 *
TCP_SERVER_TIMEOUT_S)
            throw std::runtime_error("Wait timeout reached");
    }
}

void TCPClient::request() {
    return IClient::request();
}

```

```

#include "pch.h"
#include <iostream>
#include <sstream>
#include <string>
#include <webstur/ip/tcp/tcpserver.h>

TCPServer::TCPServer(int port) {
    this->should_run = new std::atomic<bool>(false);
    this->running = new std::atomic<bool>(false);

    std::clog << "Creating socket" << std::endl;

```

```

// Создаём сокет
this->socket_descriptor = socket(
    AF_INET,
    SOCK_STREAM,
    IPPROTO_TCP
);

std::clog << "Creating socket bind addr" << std::endl;
// Создаём адрес к которому привяжется сокет
sockaddr_in bind_addr;
bind_addr.sin_family = AF_INET;
bind_addr.sin_addr = getDeviceAddrInfo().sin_addr;
bind_addr.sin_port = htons(port);

std::clog << "Binding socket" << std::endl;
// Привязать сокет к адресу
if (bind(this->socket_descriptor, (sockaddr*)&bind_addr, sizeof(bind_addr)) == SOCKET_ERROR)
    throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to bind socket
descriptor"));

std::clog << "Making socket to listen for connections" << std::endl;
// Перевести сокет в режим прослушивания
if (listen(this->socket_descriptor, SOMAXCONN) == SOCKET_ERROR)
    throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to make socket to
listen"));
}

TCPServer::~TCPServer() {
    std::clog << "Stopping worker thread" << std::endl;
    // Приостанавливаем рабочий поток
    this->shutdown();
    this->waitForServerStop();
    delete this->current_runner;
    delete this->current_threads_amount;

    std::clog << "Closing socket" << std::endl;
    // Закрываем сокет
    if (closesocket(this->socket_descriptor) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Unable to close socket"));

    delete this->should_run;
    delete this->running;
}

std::ostream& TCPServer::printServerInfo(std::ostream& out) {
    sockaddr_in server_address;
    int server_address_size = sizeof(server_address);
    int get_sock_name_res = getsockname(this->socket_descriptor, (sockaddr*)&server_address,
&server_address_size);

    std::cout << "TCP server info:\n" <<
        "Server state: " << (*this->running ? "running" : "not running") << "\n";

    if (get_sock_name_res == SOCKET_ERROR) {
        std::cout << "Unable to get socket info\n";
    }
    else {
        std::cout << "Socket info:\n";
        printSockaddrInfo(std::cout, server_address);
        std::cout << "\n";
    }

    std::cout.flush();

    return out;
}

```

```

void TCPServer::start() {
    // Если сервер уже работает, выходим из него
    if (*this->running) {
        std::clog << "Server is already running" << std::endl;
        return;
    }

    // Подготавливаем рабочий поток
    delete this->current_runner;
    delete this->current_threads_amount;

    std::clog << "Starting server" << std::endl;
    *this->should_run = true;
    this->current_threads_amount = new std::atomic<int>(0);
    this->current_runner = new std::thread(&TCPServer::serve, this);

    this->current_runner->detach();
}

void TCPServer::shutdown() {
    if (!(*this->running)) {
        std::clog << "Server is already stopped" << std::endl;
        return;
    }

    std::clog << "Stopping server" << std::endl;
    // Указываем что серверу нужно приостановиться
    *this->should_run = false;
}

void TCPServer::waitForServerStop() {
    // Используем спинлок, так как пакеты относительно небольшие и сервер должен
    // быстро увидеть что пора заканчивать работу
    while (*this->running) {
    }
}

void TCPServer::serve() {
    std::clog << "Started TCPServer work" << std::endl;
    // Устанавливаем флаг работы потока на true
    *this->running = true;

    FD_SET readfds;
    timeval timeout;
    timeout.tv_sec = TCP_SERVER_TIMEOUT_S;
    timeout.tv_usec = 0;
    // Пока сервер должен работать
    while (*this->should_run) {
        FD_ZERO(&readfds);
        FD_SET(this->socket_descriptor, &readfds);

        // Получить количество соединений для текущего сокета с таймаутом
        int connections_amount = select(0, &readfds, nullptr, nullptr, &timeout);
        if (connections_amount == SOCKET_ERROR) {
            std::cerr << getErrorTextWithWSAErrorCode("Couldn't select for socket") << std::endl;
        }
        else if (connections_amount > 0) {
            std::clog << "A pending connection is detected" << std::endl;
            // Если соединения есть, получаем его и запускаем поток для обработки
            SOCKET client_descriptor = accept(
                socket_descriptor,
                nullptr,
                nullptr);
            std::thread client_thread(&TCPServer::serveClient, this, client_descriptor);
            client_thread.detach();
        }
    }
}

```

```

}

std::cout << "Waiting for children threads to stop" << std::endl;
// Ждём пока не закончат работу потоки для работы с клиентами
while (*this->current_threads_amount > 0) {
}

std::cout << "TCPServer stopped" << std::endl;
// Устанавливаем флаг работы потока на false
*this->running = false;
}

void TCPServer::serveClient(SOCKET client) {
    std::clog << "Serving client connection" << std::endl;
    // Увеличиваем количество текущих потоков
    (*this->current_threads_amount)++;

    try {
        // Оповещаем при помощи метода onConnect что получено новое соединение
        this->onConnect(client);

        std::vector<char> buffer(0, TCP_MAX_MESSAGE_SIZE);
        FD_SET readfds;
        timeval timeout;
        timeout.tv_sec = TCP_SERVER_TIMEOUT_S;
        timeout.tv_usec = 0;
        bool outer_close = false;
        while (*this->should_run) {
            FD_ZERO(&readfds);
            FD_SET(client, &readfds);

            // Получить количество соединений для текущего сокета с таймаутом
            int to_read = select(0, &readfds, nullptr, nullptr, &timeout);
            if (to_read == SOCKET_ERROR) {
                // Если сокет закрыт извне, можно выйти из цикла
                if (WSAGetLastError() == 10038) {
                    outer_close = true;
                    break;
                }

                std::cerr << getErrorTextWithWSAErrorCode("Couldn't select for socket") <<
std::endl;
            }
            else if (to_read > 0) {
                std::clog << "A client connection received" << std::endl;
                int bytes_read;
                buffer.resize(TCP_MAX_MESSAGE_SIZE);
                // Если соединения есть, получаем его и запускаем поток для обработки
                if ((bytes_read = recv(
                    client,
                    &buffer[0],
                    buffer.size(),
                    0
                )) == SOCKET_ERROR) {
                    std::cerr << getErrorTextWithWSAErrorCode("Couldnt get answer from a client")
<< std::endl;

                    // Связь разорвана, выйти из цикла
                    break;
                }
                else if (bytes_read != 0) {
                    // Получено сообщение, оповестить при помощи метода onMessage
                    buffer.resize(bytes_read);
                    this->onMessage(client, buffer);
                }
                else break;
            }
        }
    }
}

```

```

        std::clog << "Closing client socket" << std::endl;
        // Закрываем сокет
        if (!outer_close && closesocket(client) == SOCKET_ERROR)
            std::cerr << getErrorTextWithWSAErrorCode("Unable to close client socket");
    }
    catch (...) {
        std::cerr << "An error occurred while handling user connection";
    }

    // Оповещаем при помощи метода onDisconnect о разрыве соединения
    this->onDisconnect(client);

    std::clog << "Exiting serve client thread" << std::endl;
    // Уменьшаем количество текущих потоков
    (*this->current_threads_amount)--;
}

void TCPServer::sendMessage(SOCKET client, std::istream& message) {
    char buffer[TCP_MAX_MESSAGE_SIZE];

    // Отправить данные клиенту
    while (!message.eof()) {
        message.read(buffer, sizeof(buffer));
        auto bytes_read = message.gcount();

        if (bytes_read == 0) break;

        if (send(client, buffer, bytes_read, 0) == SOCKET_ERROR)
            throw std::runtime_error(getErrorTextWithWSAErrorCode("Couldn't send data over
TCP"));
    }
}

void TCPServer::disconnect(SOCKET client) {
    if (closesocket(client) == SOCKET_ERROR)
        throw std::runtime_error(getErrorTextWithWSAErrorCode("Couldn't close connection for
socket"));
}

```

```

#include <iostream>
#include <algorithm>
#include <webstur/ip/tcp/file/filetcpclient.h>

int main() {
    try {
        // Инициализация библиотеки WSA
        IClient::init();

        // Ввести ip и порт
        std::cout << "Enter IP Address and port" << std::endl;
        std::string address;
        int port;
        std::cin >> address >> port;

        // Создать клиент
        TCPClient* c = new FileTCPClient(std::string(address), port);

        // Оставляем клиент работать, пока пользователь не решит его приостановить
        std::string input;
        while (true)
        {
            std::cout << "Enter Y to request a file\n"
                << "Enter N to stop client\n"
                << "Enter E to exit loop\n"

```

```

        << "Enter S to print status" << std::endl;

std::cin >> input;
std::transform(input.begin(), input.end(), input.begin(), toupper);
if (input == "Y") {
    // Ввести запрашиваемый файл
    std::cout << "Enter file to request: ";
    std::cout.flush();
    std::cin >> input;
    // Запустить клиент
    c->request(&input[0], input.size());
}
else if (input == "N") {
    // Приостановить клиент
    c->shutdown();
}
else if (input == "E") {
    // Выход из цикла
    break;
}
else if (input == "S") {
    // Вывести информацию о клиенте
    c->printClientInfo(std::cout);
}
}

// Приостановить сервер
c->shutdown();

delete c;
}
catch (const std::runtime_error& error) {
    std::cerr << "Failed while running client. Caused by: '" << error.what() << "'" <<
std::endl;

    return -1;
}

// Выгрузка библиотеки WSA
IClient::detach();

return 0;
}

```

```

#include <iostream>
#include <algorithm>
#include <webstur/ip/tcp/file/filetcpserver.h>

int main() {
    try {
        // Инициализация библиотеки WSA
        IServer::init();

        // Создать сервер
        IServer* s = new FileTCPServer();

        std::string input;
        // Оставляем сервер работать, пока пользователь не решит его приостановить
        while (true)
        {
            std::cout << "Enter Y to start accepting user requests\n"
                << "Enter N to stop server\n"
                << "Enter E to exit loop\n"
                << "Enter S to print status" << std::endl;

```

```

std::cin >> input;
std::transform(input.begin(), input.end(), input.begin(), toupper);
if (input == "Y") {
    // Запустить сервер
    s->start();
}
else if (input == "N") {
    // Приостановить сервер
    s->shutdown();
}
else if (input == "E") {
    // Выход из цикла
    break;
}
else if (input == "S") {
    // Вывести информацию о сервере
    s->printServerInfo(std::cout);
}
}

// Приостановить сервер
s->shutdown();

delete s;
}
catch (const std::runtime_error& error) {
    std::cerr << "Failed while running server. Caused by: '" << error.what() << "' <<
std::endl;

    return -1;
}

// Выгрузка библиотеки WSA
IServer::detach();

return 0;
}

```