**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

Игнатьев Артур Олегович

Проверили:

Рубцов Константин Анатольевич

Белгород 2025 г.

**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 [12].

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 A посылает узлу B специальный пакет SYN — приглашение к соединению; B отвечает пакетом SYN-ACK — согласием об установлении соединения; A 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 A это соединение называется исходящим, а для узла B - 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 порта [2, 4, 13, 14].

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.

Заголовок UDP 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. Потоковые 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.

Windows Sockets API (WSA) (сокр. Winsock) – техническая спецификация, которая определяет, как сетевое программное обеспечение Windows будет получать доступ к сетевым сервисам [3].

Winsock – это интерфейс сетевого программирования для Microsoft Windows.

Функция WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData) инициализирует библиотеку Winsock. В случае успеха возвращает 0. Дальше можно использовать любые остальные функции этой библиотеки, иначе возвращает код возникшей ошибки. WwVersionRequested – это необходимая минимальная версия библиотеки, при присутствии которой приложение будет корректно работать. Младший байт содержит номер версии, а старший – номер ревизии. LpWSAData – структура, в которую возвращается информация по инициализированной библиотеке (статус, версия и т.д.).

Функция WSAGetLastError (void) возвращает код ошибки, возникшей при выполнении последней операции. После работы с библиотекой, её необходимо выгрузить из памяти.

Функция WSACleanup (void) осуществляет очистку памяти, занимаемой библиотекой Winsock. Функция деинициализирует библиотеку Winsock и возвращает 0, если операция была выполнена успешно, иначе возвращает SOCKET\_ERROR. Расширенный код ошибки можно получить при помощи функции WSAGetLastError.

Функция socket (int af, int type, int protocol) возвращает либо дескриптор созданного сокета, либо ошибку INVALID\_SOCKET. Расширенный код ошибки можно получить при помощи функции WSAGetLastError.

Чтобы работать дальше с созданным сокетом его нужно привязать к какому-нибудь локальному адресу и порту. Этим занимается функция bind (SOCKET s, const struct sockaddr FAR\* name, int namelen). Здесь s – дескриптор сокета, который данная функция именует; name – указатель на структуру имени сокета; namelen – размер, в байтах, структуры name.

Функция listen (SOCKET s, int backlog) переводит сокет в состояние “прослушивания” (для протокола SPX). Здесь s – дескриптор сокета; backlog – это максимальный размер очереди входящих сообщений на соединение.

Функция connect (SOCKET s, const struct sockaddr FAR\* name, int namelen) используется процессом-клиентом для установления связи с сервером по протоколу SPX. 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) используется для принятия связи на сокет. Здесь s – дескриптор сокета; addr – указатель на структуру sockaddr; addrlen – размер структуры addr. 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 [8].

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). Здесь s — дескриптор сокета; name — структура sockaddr, в 24 которую система поместит данные; namelen — размер, в байтах, структуры name. Если операция выполнена успешно, возвращает 0, иначе возвращает SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

Передача данных по протоколу IPX осуществляется с помощью функции sendto (SOCKET s, const char FAR \* buf, int len, int flags, const struct sockaddr FAR \* to, int tolen). Здесь 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; to - указатель на структуру sockaddr, которая содержит адрес сокета-приёмника; tolen - размер структуры to. Если операция выполнена успешно, возвращает количество переданных байт, иначе возвращает SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

Передача данных по протоколу SPX осуществляется с помощью функции send (SOCKET s, const char FAR \* buf, int len, int flags). Здесь 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.

Прием данных по протоколу IPX осуществляется с помощью функции 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.

Прием данных по протоколу SPX осуществляется с помощью функции 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.

Функция closesocket(SOCKET s) служит для закрытия сокета. Возвращает 0, если операция была выполнена успешно, иначе возвращает SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

Анализ применяемых функций

Для использования библиотеки применяется WSAStartUp, WSACleanUp и WSAGetLastError для отслеживания ошибок и логирования.

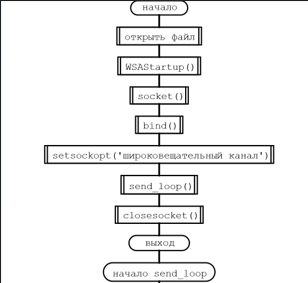
Для реализации серверной части использованы функции socket (у разных протоколов своя настройка) и bind. В части TCP сервера, применена listen для прослушивания сокета, после запускается поток на подключение пользователей с помощью функции accept.

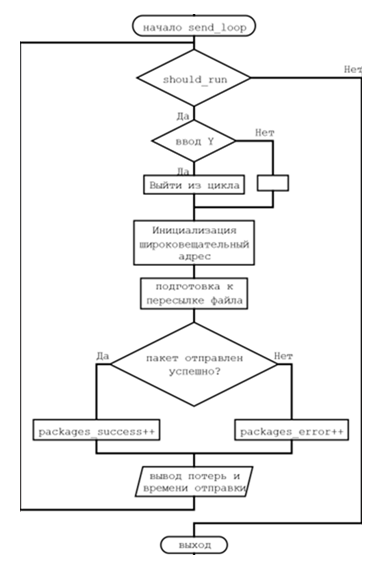
Для пересылки в UDP используется sendto в цикле с указанием получателя, а в TCP send в цикле. Для получения в UDP recvfrom с занулённым отправителем, в TCP используется recv – обе функции работают в цикле.

Пользователь также подключается через socket, в UDP выполняется bind, а TCP применяет connect. В дескрипторе реализации клиента адрес указывается в зависимости от выбранного протокола (UDP - собственный, TCP - сервера).

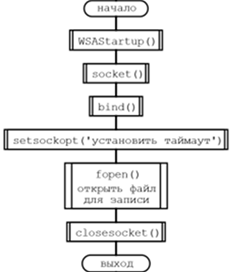
**Разработка программы. Блок-схемы программы**

Сервер UDP

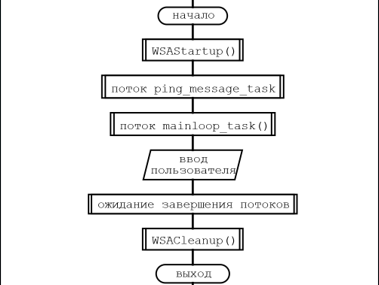


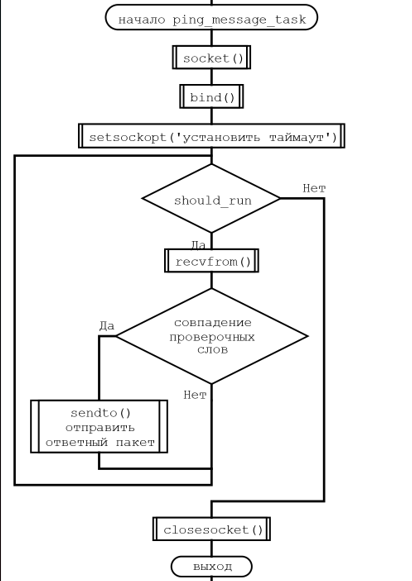


Клиент UDP

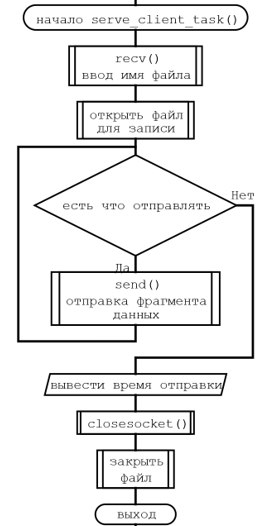


Сервер TCP

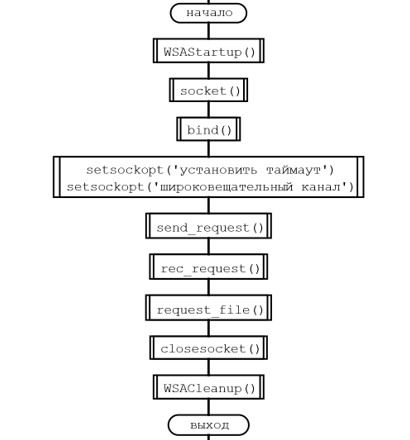


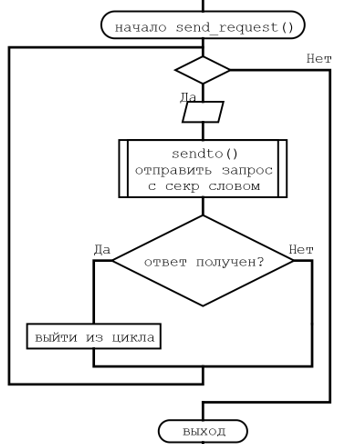


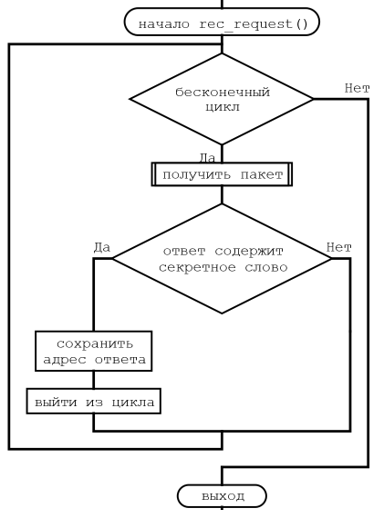


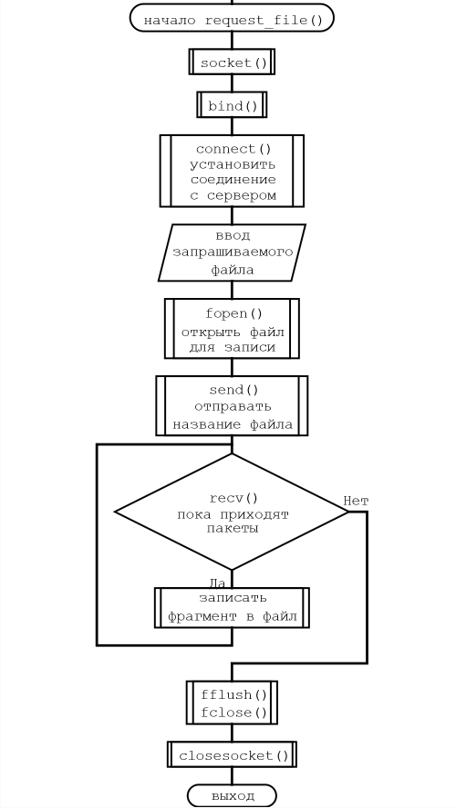


Клиент TCP









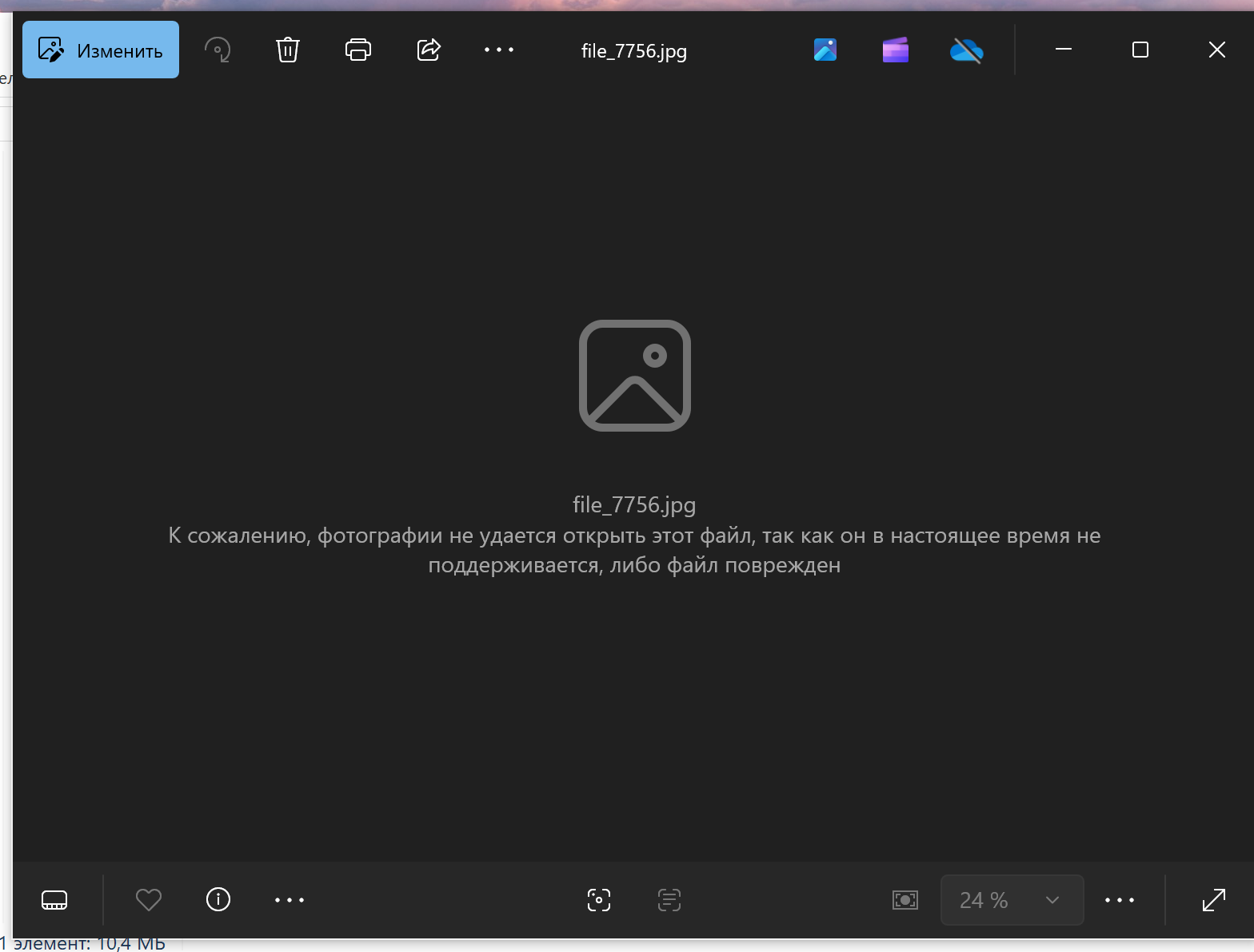
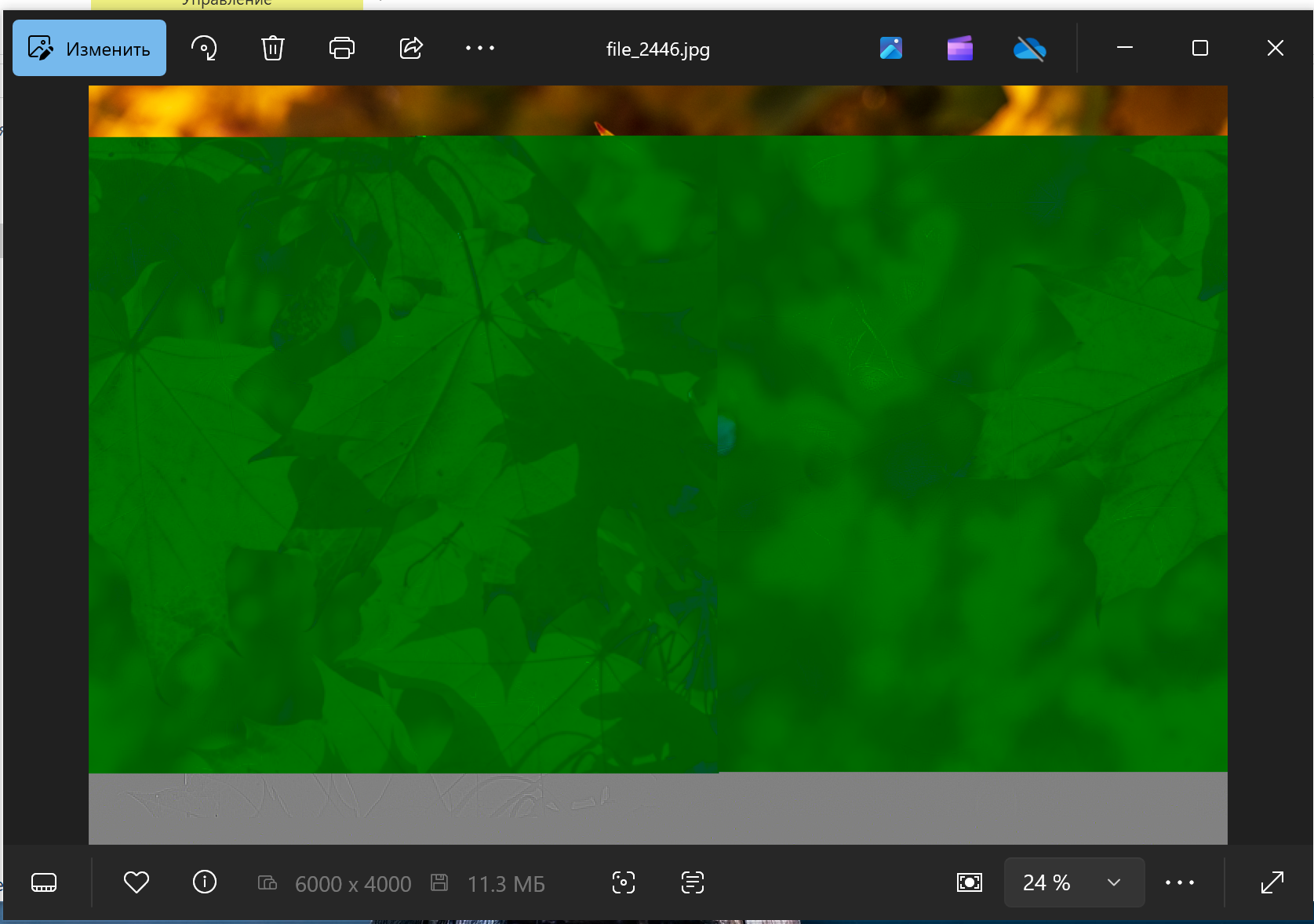
**Анализ функционирования программы**

Пример переданного изображения

Исходное изображение

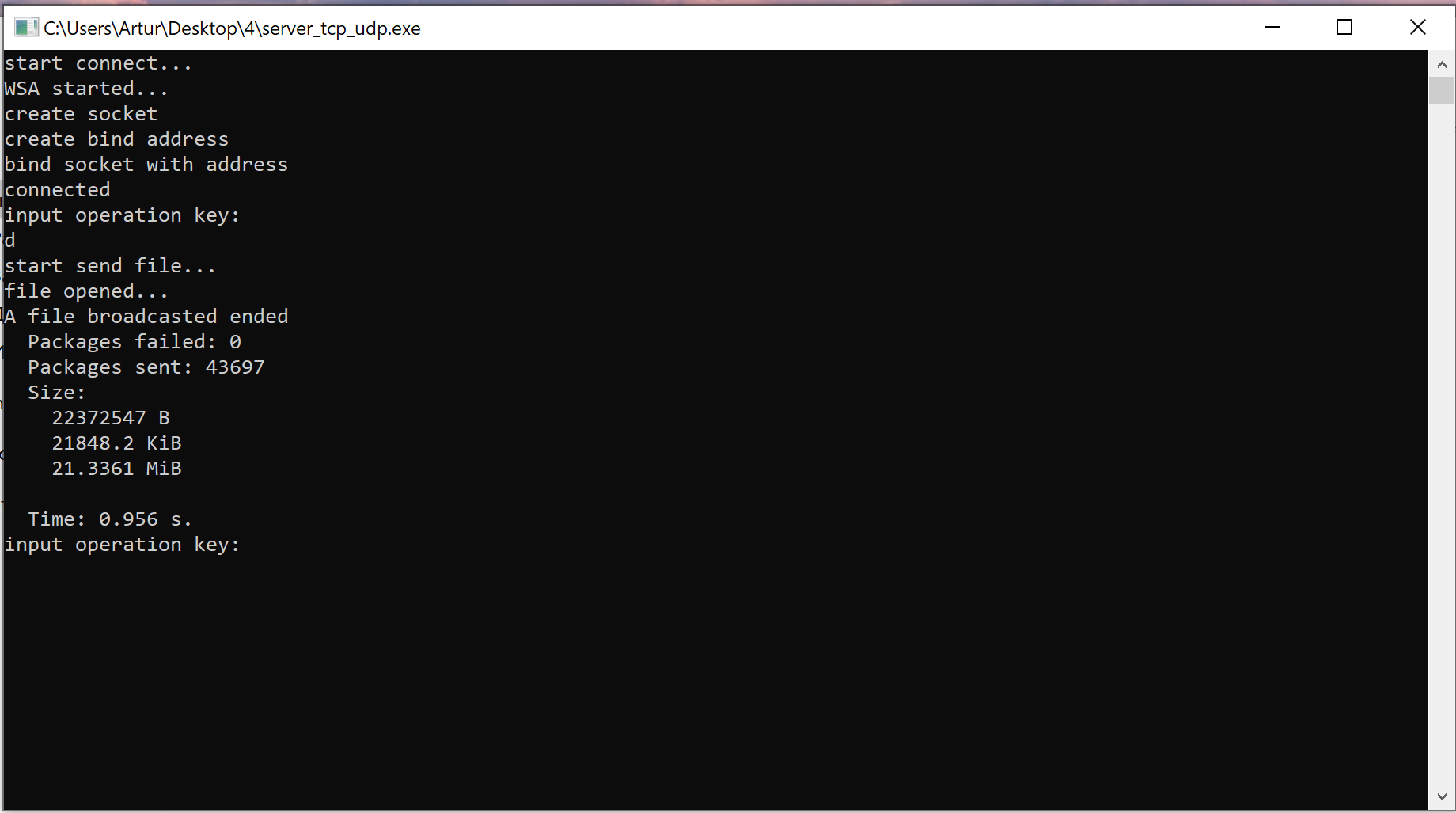


Переданные изображения UDP

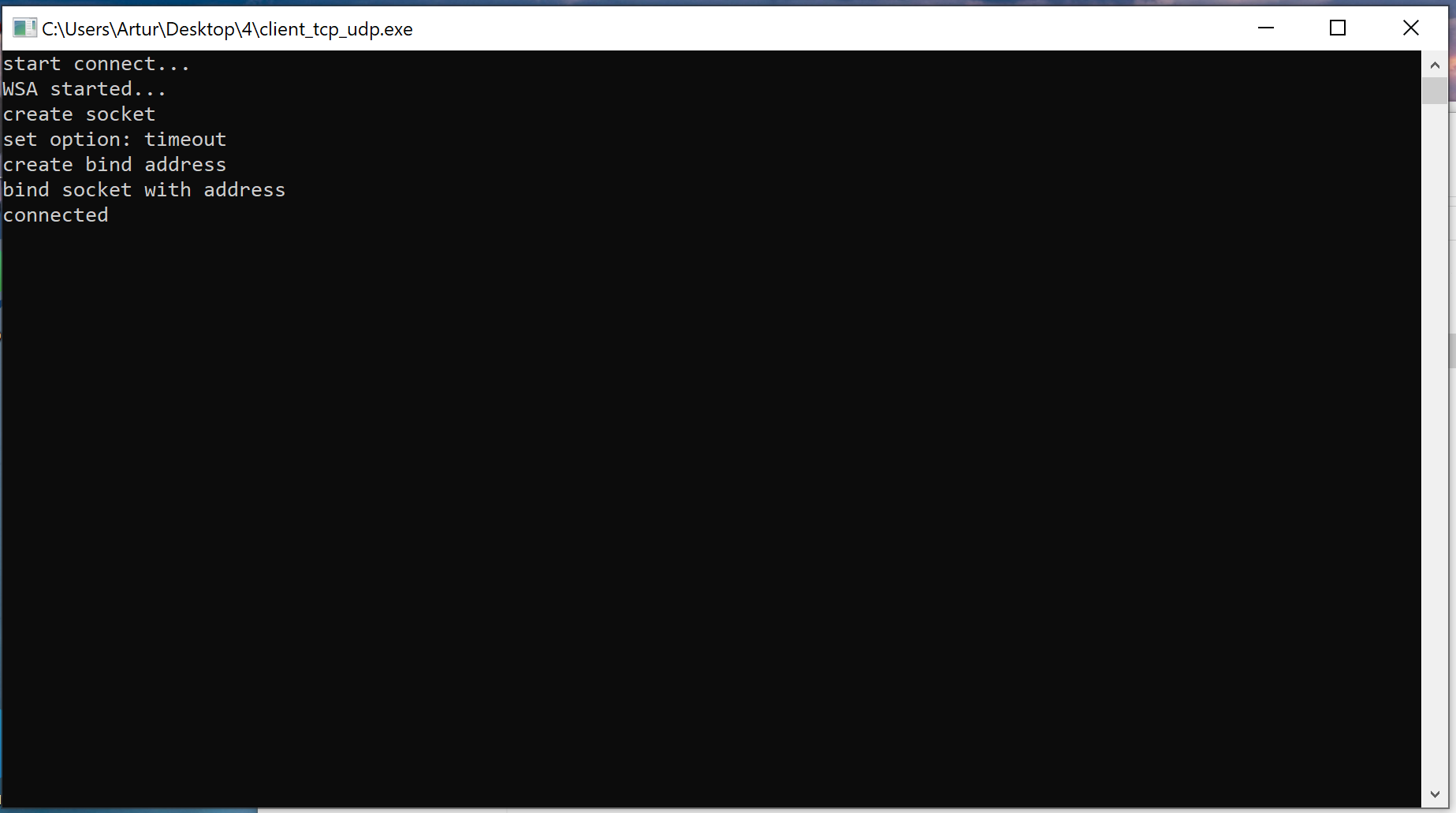
 

Работа программы:

Сервер



Клиент



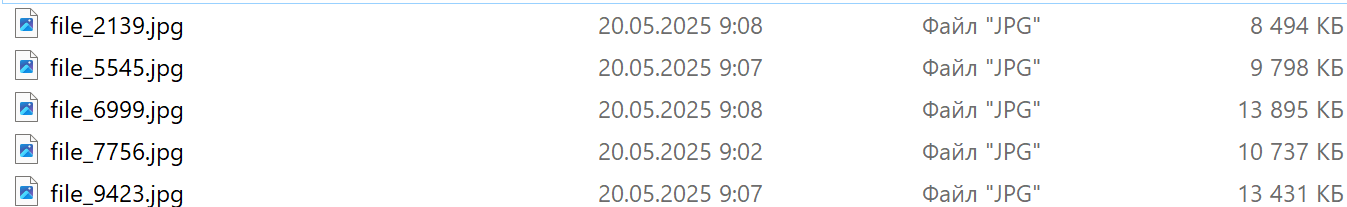
Передача осуществлялась 2м клиентам

|  |  |
| --- | --- |
|  | Время, сек |
| Передача №1 | 0.956 |
| Передача №2 | 0.938 |
| Передача №3 | 0.884 |
| Передача №4 | 0.874 |
| Передача №5 | 0.909 |
| Среднее | 0.912 |

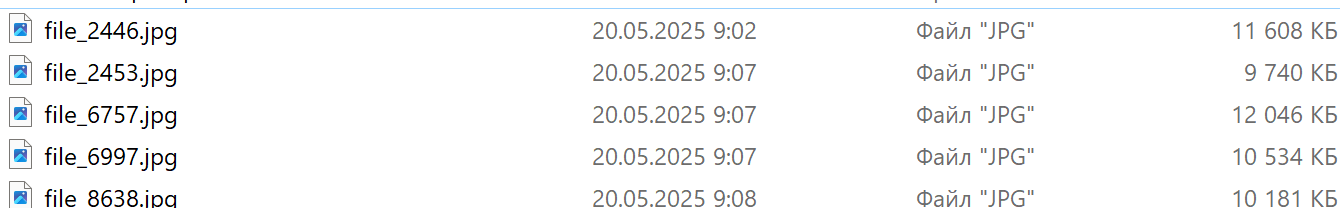
|  |  |
| --- | --- |
| Размер изображения Мбайт | Скорость передачи, Мбит/с |
| 21.3361 | 187,11 |

Размер полученных изображений:

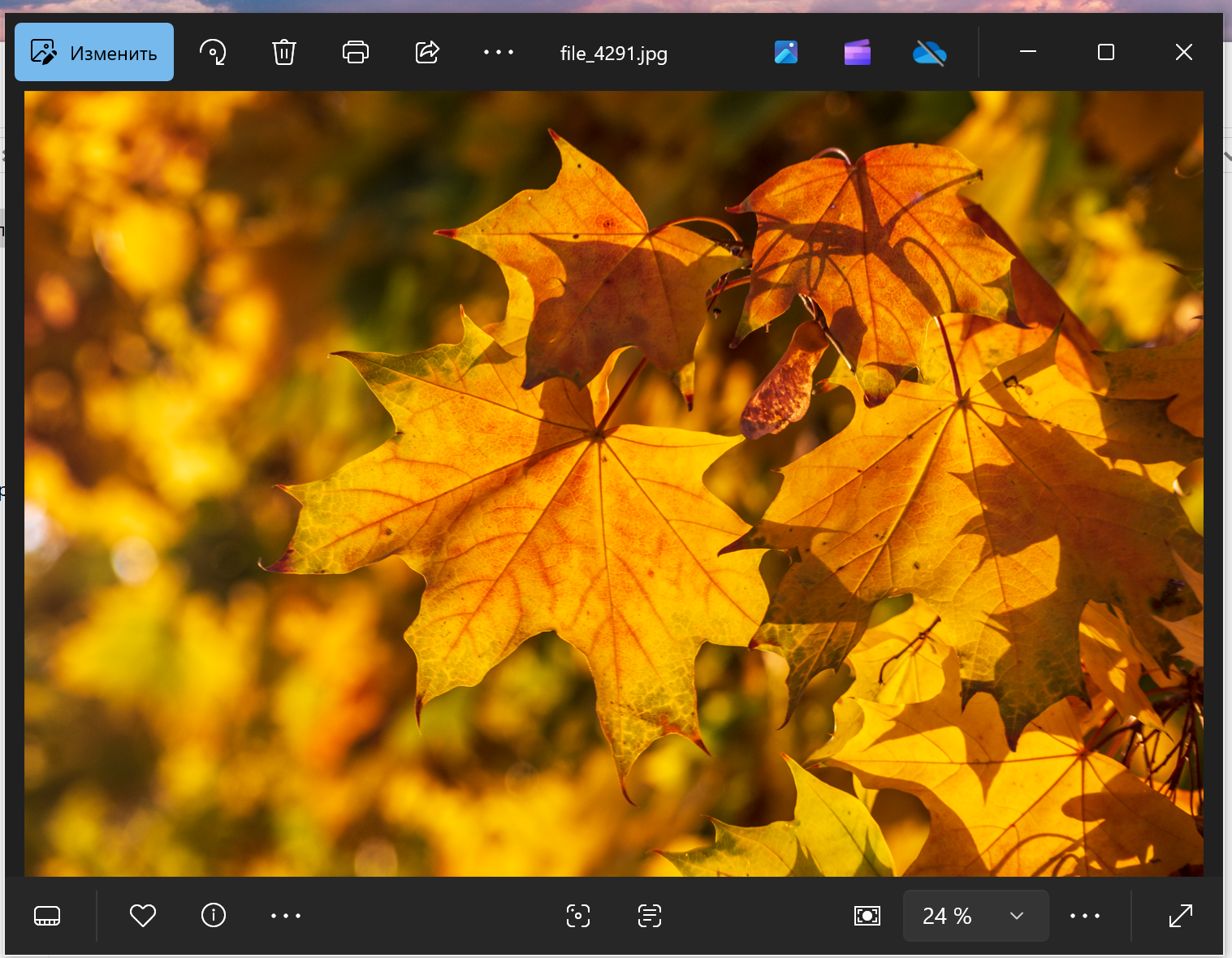
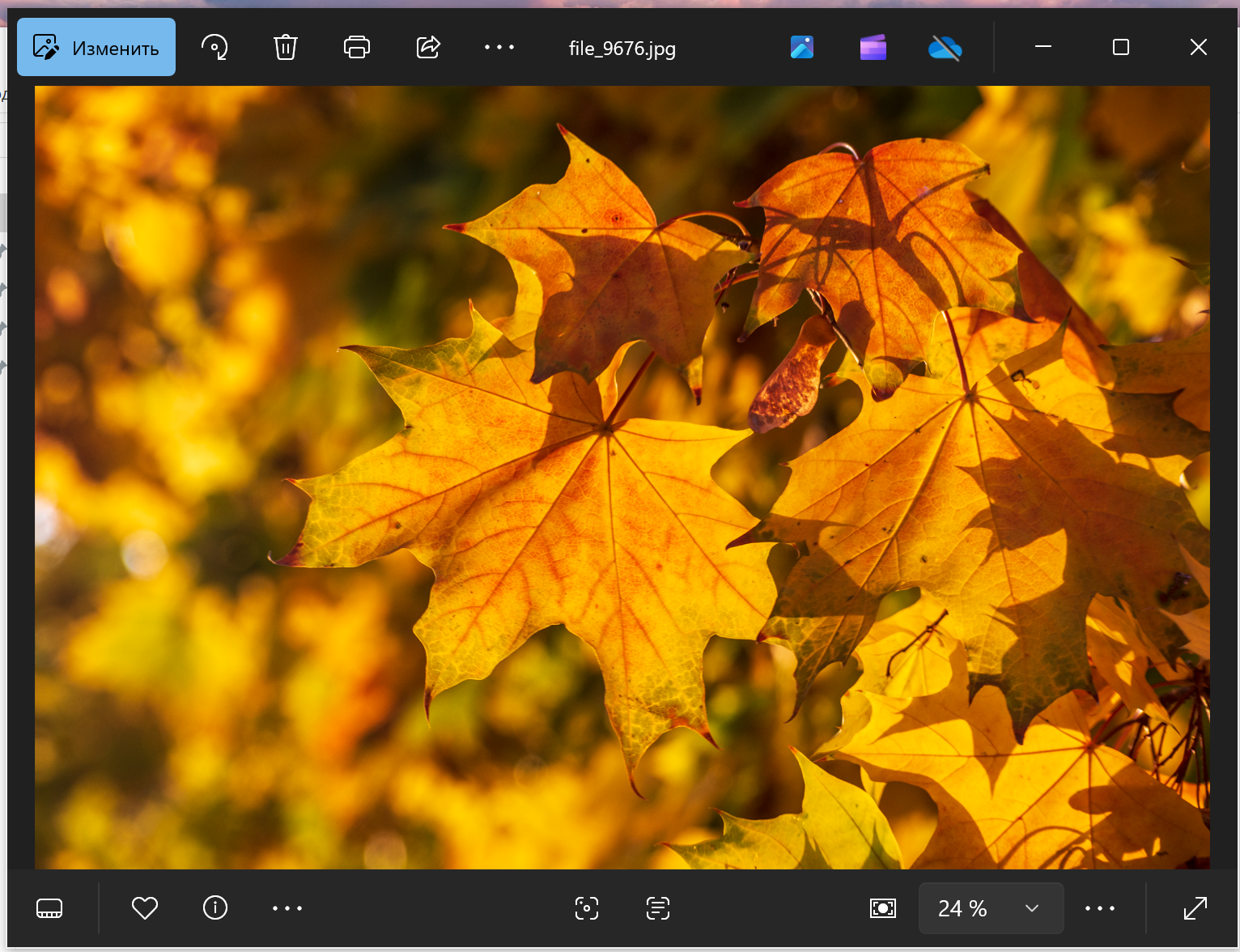
Клиент 1:



Клиент 2:

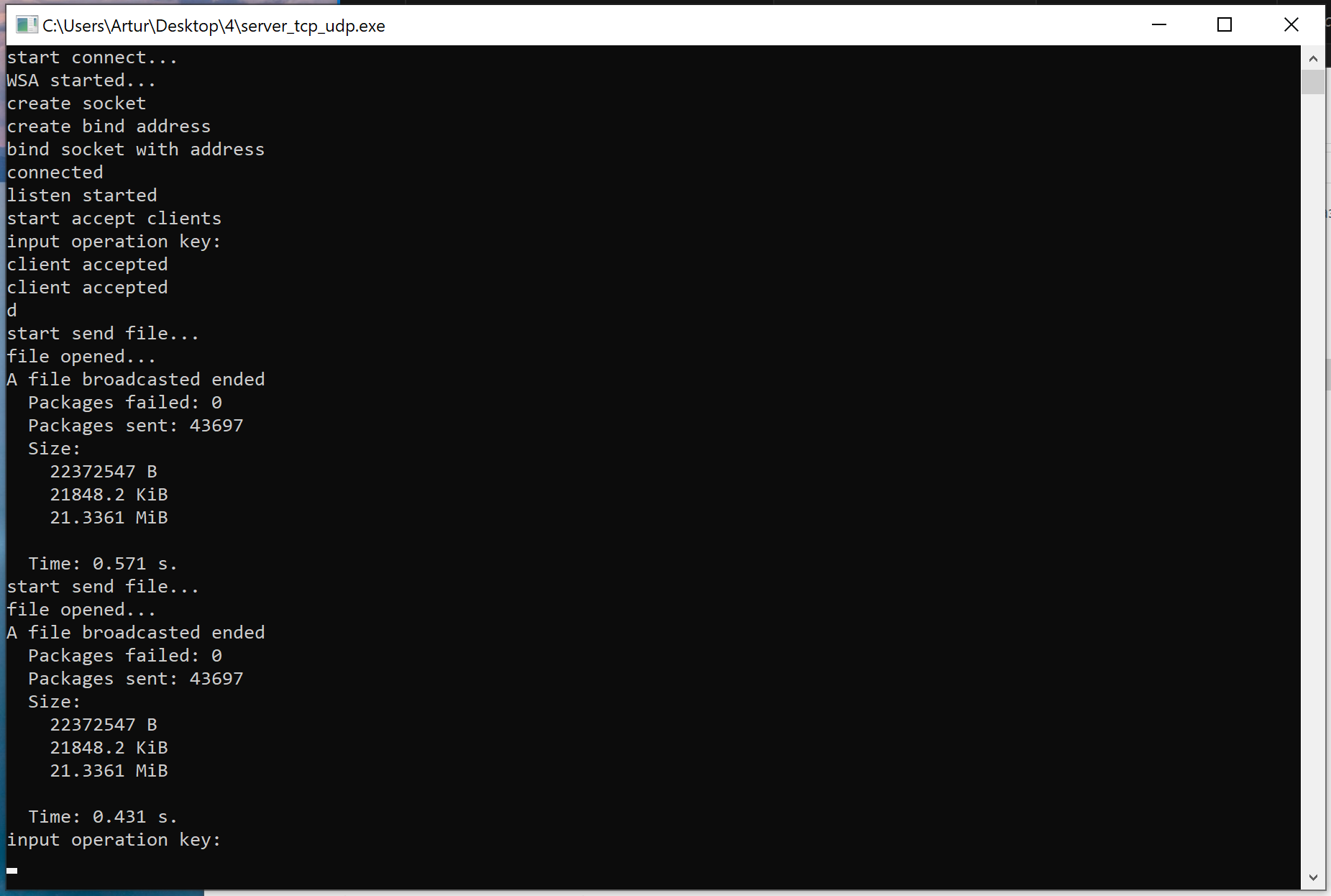


Переданные изображения TCP

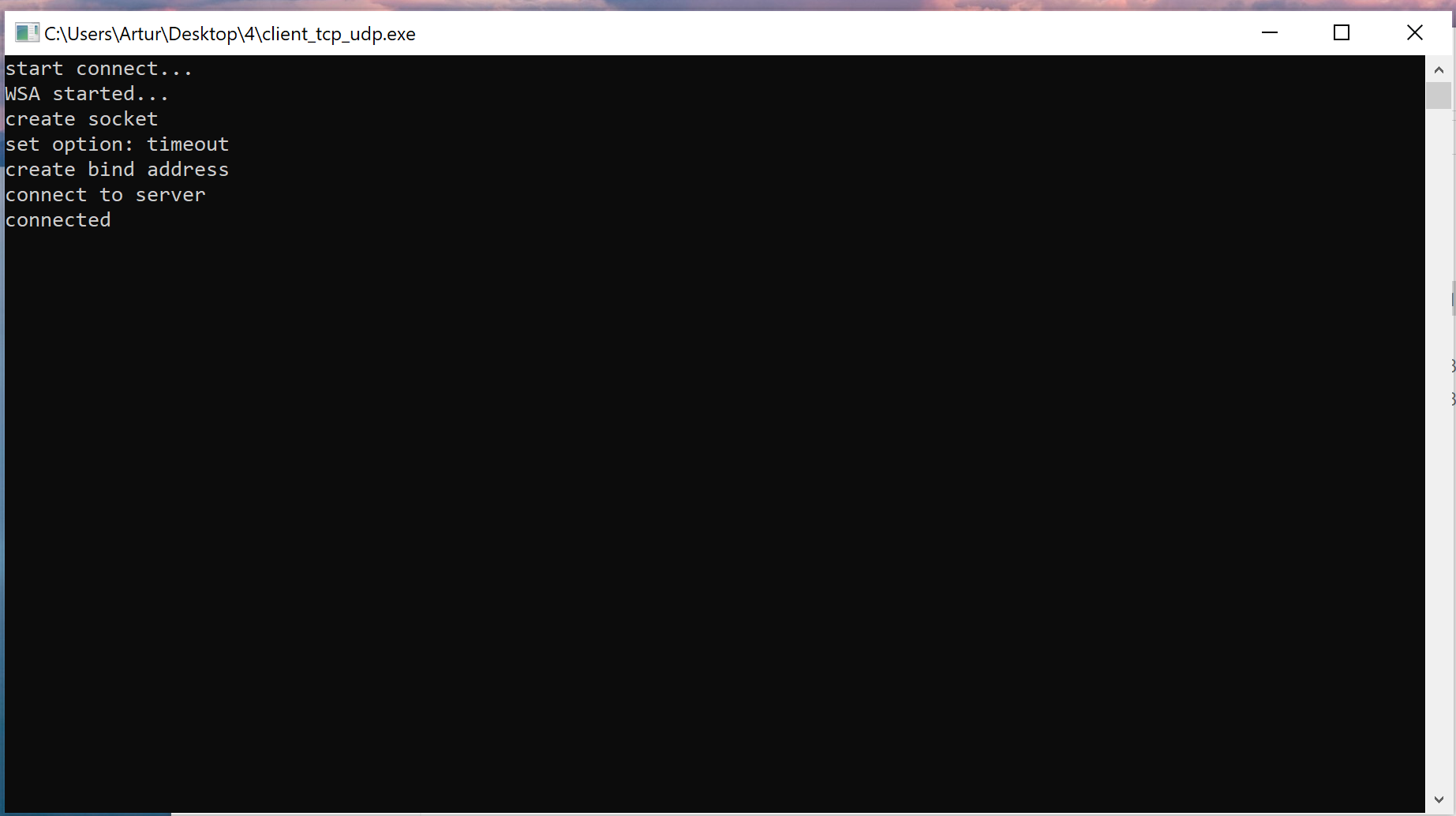
 

Работа программы:

Сервер



Клиент



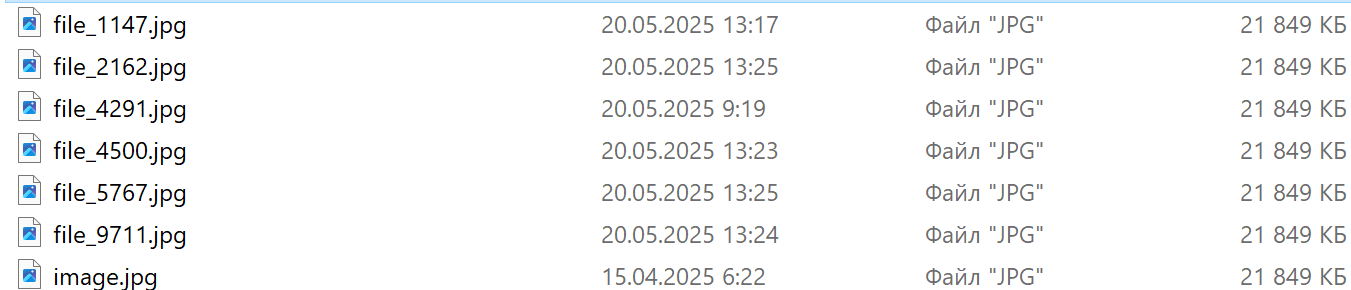
Передача осуществлялась 2м клиентам

|  |  |  |
| --- | --- | --- |
|  | Время, сек | |
| Клиент 1 | Клиент 2 |
| Передача №1 | 0.571 | 0.431 |
| Передача №2 | 0.628 | 0.892 |
| Передача №3 | 0.86 | 0.914 |
| Передача №4 | 0.452 | 0.439 |
| Передача №5 | 0.426 | 0.304 |
| Среднее | 1,1834 | |

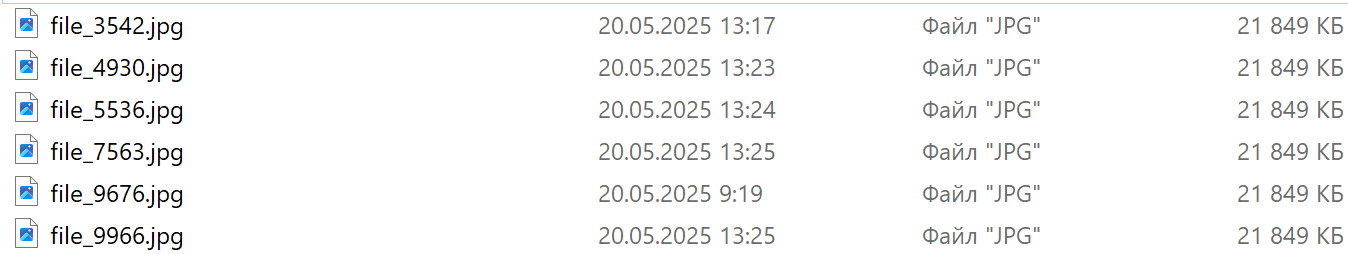
|  |  |
| --- | --- |
| Размер изображения Мбайт | Скорость передачи, Мбит/с |
| 21.3361 | 144,24 |

Размер полученных изображений:

Клиент 1:



Клиент 2:



**Вывод:** в ходе работы изучены протоколы TCP и UDP, применены основные функции библиотеки Winsock и составлены программы для приема/передачи пакетов. У UDP скорость передачи выше на 43 Мбит/с выше, чем у TCP. Но половина пакетов не доходит до клиентов, так как, если один клиент принял конкретный пакет, то другой этот пакет не получит. Значит теоретически, например при 10 клиентах, потери бы составили приблизительно 9/10. У TCP все пакеты доходят корректно, однако из-за последовательной передачи, пока один клиент не получит все пакеты, следующий клиент будет находится в ожидании, и следовательно, чем больше клиентом, тем выше задержка.

Код программы:

Файл Server\_TCP\_UDP

#include <WinSock2.h>

#include <winsock.h>

#include <ws2tcpip.h>

#include <iostream>

#include <exception>

#include <string>

#include <fstream>

#include <chrono>

#include <vector>

#include <pthread.h>

#define FILE\_FRAGMENT\_SIZE 512 // Размер фрагмента файла, отправляемого за один пакет

// Протоколы передачи

enum TransportProtocol

{

    TCP,

    UDP

};

// Структура аргументов для сервера

struct server\_args

{

    SOCKET server\_socket;

    sockaddr\_in socket\_address;

    TransportProtocol protocol;

};

// Флаг работы сервера и список клиентов (для TCP)

bool should\_run = false;

std::vector<SOCKET> clients;

// Генерация исключения с кодом ошибки Winsock

void throw\_error\_with\_code()

{

    std::string err\_msg = "error with code: ";

    err\_msg += std::to\_string(WSAGetLastError());

    throw std::runtime\_error(err\_msg);

}

// Объявления функций

void startup\_wsa();

SOCKET get\_socket\_descriptor\_tcp();

SOCKET get\_socket\_descriptor\_udp();

void set\_option\_broadcast(SOCKET socket\_descriptor);

sockaddr\_in get\_bind\_addr(const char \*address, unsigned short port);

void bind\_socket\_with\_address(SOCKET socket\_descriptor, sockaddr\_in bind\_addr);

void listen\_connections(SOCKET socket\_descriptor);

pthread\_t run\_accept\_clients(SOCKET connection);

// Подключение к серверу и настройка сокета

SOCKET connect(const char \*address, unsigned short port, TransportProtocol protocol)

{

    std::clog << "start connect..." << std::endl;

    startup\_wsa(); // Инициализация библиотеки Winsock

    std::clog << "WSA started..." << std::endl;

    SOCKET socket\_descriptor;

    if (protocol == TCP)

        socket\_descriptor = get\_socket\_descriptor\_tcp(); // Создание TCP сокета

    else if (protocol == UDP)

        socket\_descriptor = get\_socket\_descriptor\_udp(); // Создание UDP сокета

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

    sockaddr\_in bind\_addr = get\_bind\_addr(address, port); // Формирование структуры адреса

    std::clog << "create bind address" << std::endl;

    bind\_socket\_with\_address(socket\_descriptor, bind\_addr); // Привязка сокета к адресу

    std::clog << "bind socket with address\nconnected" << std::endl;

    // Если TCP, слушаем подключения и запускаем поток для приема клиентов

    if (protocol == TCP)

    {

        listen\_connections(socket\_descriptor);

        std::clog << "listen started" << std::endl;

        std::clog << "start accept clients" << std::endl;

        run\_accept\_clients(socket\_descriptor);

    }

    return socket\_descriptor;

}

// Инициализация библиотеки Winsock

void startup\_wsa()

{

    WORD wVersionRequested = MAKEWORD(2, 0);

    WSADATA wsaData;

    if (WSAStartup(wVersionRequested, &wsaData) == SOCKET\_ERROR)

        throw\_error\_with\_code();

}

// Создание TCP сокета

SOCKET get\_socket\_descriptor\_tcp()

{

    SOCKET res = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

    if (res == INVALID\_SOCKET)

        throw\_error\_with\_code();

    return res;

}

// Создание UDP сокета

SOCKET get\_socket\_descriptor\_udp()

{

    SOCKET res = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_IP);

    if (res == INVALID\_SOCKET)

        throw\_error\_with\_code();

    return res;

}

// Установка опции широковещательной передачи (для UDP)

void set\_option\_broadcast(SOCKET socket\_descriptor)

{

    bool broadcast = true;

    if (setsockopt(socket\_descriptor, SOL\_SOCKET, SO\_BROADCAST, (char \*)&broadcast, sizeof(broadcast)) == SOCKET\_ERROR)

        throw\_error\_with\_code();

}

// Получение структуры адреса для привязки

sockaddr\_in get\_bind\_addr(const char \*address, unsigned short port)

{

    sockaddr\_in res;

    res.sin\_family = AF\_INET;

    res.sin\_addr.s\_addr = inet\_addr(address);

    res.sin\_port = htons(port);

    return res;

}

// Привязка сокета к адресу

void bind\_socket\_with\_address(SOCKET socket\_descriptor, sockaddr\_in bind\_addr)

{

    if (bind(socket\_descriptor, (sockaddr \*)&bind\_addr, sizeof(bind\_addr)) == SOCKET\_ERROR)

        throw\_error\_with\_code();

}

// Перевод сокета в режим прослушивания (для TCP)

void listen\_connections(SOCKET socket\_descriptor)

{

    if (listen(socket\_descriptor, SOMAXCONN) == SOCKET\_ERROR)

        throw\_error\_with\_code();

}

// Поток для приёма подключений клиентов

void \*start\_loop\_accept\_clients(void \*arg)

{

    SOCKET con = ((SOCKET)(intptr\_t)arg);

    should\_run = true;

    while (should\_run)

    {

        sockaddr\_in client\_addr;

        int client\_addr\_size = sizeof(client\_addr);

        SOCKET client\_socket = accept(con, (sockaddr \*)&client\_addr, &client\_addr\_size);

        clients.emplace\_back(client\_socket);

        if (client\_socket != INVALID\_SOCKET)

            std::clog << "client accepted" << std::endl;

        else

            throw\_error\_with\_code();

    }

    return nullptr;

}

// Запуск потока для приёма клиентов

pthread\_t run\_accept\_clients(SOCKET connection)

{

    pthread\_t thread;

    if (pthread\_create(&thread, nullptr, &start\_loop\_accept\_clients, (void \*)(intptr\_t)connection) != 0)

        throw "error when start thread\n";

    return thread;

}

// Завершение подключения и очистка Winsock

void disconnect(SOCKET connection)

{

    std::clog << "start disconnect..." << std::endl;

    if (closesocket(connection) == SOCKET\_ERROR)

        throw\_error\_with\_code();

    WSACleanup();

    std::clog << "disconnected" << std::endl;

}

// Функции для отправки файла

void send\_file\_by\_path(SOCKET con, sockaddr\_in client\_addr, std::string file\_path, TransportProtocol protocol);

// Основной цикл сервера

void \*start\_loop\_server(void \*args)

{

    struct server\_args typed\_args = \*((server\_args \*)args);

    SOCKET con = typed\_args.server\_socket;

    TransportProtocol protocol = typed\_args.protocol;

    should\_run = true;

    while (should\_run)

    {

        std::cout << "input operation key:" << std::endl;

        char operation\_key = '\0';

        std::cin >> operation\_key;

        switch (operation\_key)

        {

        case 'i':

        case 'I': // Ввод пути к файлу

        {

            std::cout << "input path to file from current dir:" << std::endl;

            std::string file\_path;

            std::cin >> file\_path;

            if (protocol == UDP)

                send\_file\_by\_path(con, typed\_args.socket\_address, file\_path, protocol);

            else if (protocol == TCP)

                for (SOCKET &client : clients)

                    send\_file\_by\_path(client, {}, file\_path, protocol);

        }

        break;

        case 'd':

        case 'D': // Отправка фиксированного файла (image.jpg)

            if (protocol == UDP)

                send\_file\_by\_path(con, typed\_args.socket\_address, std::string("./image.jpg"), protocol);

            else if (protocol == TCP)

                for (SOCKET &client : clients)

                    send\_file\_by\_path(client, {}, std::string("./image.jpg"), protocol);

            break;

        case 'c':

        case 'C': // Завершение работы

            should\_run = false;

            break;

        default:

            std::cout << "incorrect input, try again" << std::endl;

            break;

        }

        // Закрытие клиентских сокетов

        for (SOCKET &sock : clients)

            closesocket(sock);

        clients.clear();

    }

    return nullptr;

}

// Запуск потока сервера

pthread\_t run\_server(struct server\_args connection)

{

    pthread\_t thread;

    if (pthread\_create(&thread, nullptr, &start\_loop\_server, (void \*)&connection) != 0)

        throw "error when start thread\n";

    return thread;

}

// Получение указателя на открытый файл

std::ifstream \*new\_get\_file(std::string path)

{

    std::ifstream \*file = new std::ifstream(path, std::ios::binary);

    if (!file->is\_open())

        throw "Unable to open file for read: " + path;

    return file;

}

// Альтернативная функция (не используется в коде)

std::ifstream get\_file(std::string path)

{

    std::ifstream file = std::ifstream(path, std::ios::binary);

    if (!file.is\_open())

        throw "Unable to open file for read: " + path;

    return file;

}

// Отправка файла по протоколу TCP или UDP

void send\_file(SOCKET con, sockaddr\_in client\_addr, std::ifstream \*file, TransportProtocol protocol)

{

    char buffer[FILE\_FRAGMENT\_SIZE];

    int packages\_success = 0, packages\_failed = 0;

    int total\_bytes = 0;

    auto a = std::chrono::high\_resolution\_clock::now();

    while (should\_run && !file->eof())

    {

        file->read(buffer, sizeof(buffer));

        int bytes\_read = file->gcount();

        total\_bytes += bytes\_read;

        if (protocol == UDP &&

            sendto(con, buffer, bytes\_read, 0, (sockaddr \*)&client\_addr, sizeof(client\_addr)) != SOCKET\_ERROR)

            packages\_success++;

        else if (protocol == TCP &&

                 send(con, buffer, bytes\_read, 0) != SOCKET\_ERROR)

            packages\_success++;

        else

            packages\_failed++;

    }

    auto b = std::chrono::high\_resolution\_clock::now();

    // Статистика по отправке

    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;

}

// Отправка файла по пути

void send\_file\_by\_path(SOCKET con, sockaddr\_in client\_addr, std::string file\_path, TransportProtocol protocol)

{

    std::clog << "start send file..." << std::endl;

    std::ifstream \*file = new\_get\_file(file\_path);

    std::clog << "file opened..." << std::endl;

    send\_file(con, client\_addr, file, protocol);

    file->close();

    delete file;

}

// Точка входа в программу

int main()

{

    auto con = connect("192.168.41.96", 0x8080, TCP); // Установление соединения

    sockaddr\_in client\_addr; //= get\_bind\_addr("192.168.41.96", 0x8080); // Адрес клиента (для UDP)

    struct server\_args args = {con, client\_addr, TCP};

    auto server\_thread = run\_server(args); // Запуск сервера

    pthread\_join(server\_thread, NULL); // Ожидание завершения потока

    disconnect(con); // Отключение

    return 0;

}

Файл Client\_TCP\_UDP

#include <WinSock2.h>

#include <winsock.h>

#include <ws2tcpip.h>

#include <iostream>

#include <exception>

#include <string>

#include <fstream>

#include <chrono>

#include <vector>

#include <random>

// Размер одного фрагмента файла (512 байт)

#define FILE\_FRAGMENT\_SIZE 512

// Перечисление для указания протокола передачи данных

enum TransportProtocol

{

    TCP,

    UDP

};

// Глобальная переменная для управления работой клиента

bool should\_run = false;

// Генерация исключения с кодом ошибки из WinSock

void throw\_err\_with\_code()

{

    std::string err\_msg = "error with code: ";

    err\_msg += std::to\_string(WSAGetLastError());

    throw std::runtime\_error(err\_msg);

}

// Объявления вспомогательных функций

void startup\_wsa();

SOCKET get\_socket\_descriptor\_tcp();

SOCKET get\_socket\_descriptor\_udp();

void set\_option\_timeout(SOCKET socket\_descriptor, unsigned int timeout\_ms);

sockaddr\_in get\_bind\_addr(const char \*address, unsigned short port);

void bind\_socket\_with\_address(SOCKET socket\_descriptor, sockaddr\_in bind\_addr);

void connect\_to\_server(SOCKET socket\_descriptor, sockaddr\_in bind\_addr);

// Функция подключения клиента к серверу

SOCKET connect(const char \*address, unsigned short port, TransportProtocol protocol)

{

    std::clog << "start connect..." << std::endl;

    startup\_wsa(); // Инициализация библиотеки WinSock

    std::clog << "WSA started..." << std::endl;

    // Создание сокета в зависимости от протокола

    SOCKET socket\_descriptor;

    if (protocol == TCP)

        socket\_descriptor = get\_socket\_descriptor\_tcp();

    else if (protocol == UDP)

        socket\_descriptor = get\_socket\_descriptor\_udp();

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

    // Установка таймаута на приём

    set\_option\_timeout(socket\_descriptor, 10000);

    std::clog << "set option: timeout" << std::endl;

    // Получение структуры адреса

    sockaddr\_in bind\_addr = get\_bind\_addr(address, port);

    std::clog << "create bind address" << std::endl;

    // Подключение или привязка сокета к адресу

    if (protocol == UDP)

    {

        bind\_socket\_with\_address(socket\_descriptor, bind\_addr);

        std::clog << "bind socket with address\nconnected" << std::endl;

    }

    else if (protocol == TCP)

    {

        connect\_to\_server(socket\_descriptor, bind\_addr);

        std::clog << "connect to server\nconnected" << std::endl;

    }

    return socket\_descriptor;

}

// Инициализация WinSock

void startup\_wsa()

{

    WORD wVersionRequested = MAKEWORD(2, 0);

    WSADATA wsaData;

    if (WSAStartup(wVersionRequested, &wsaData) == SOCKET\_ERROR)

        throw\_err\_with\_code();

}

// Создание TCP-сокета

SOCKET get\_socket\_descriptor\_tcp()

{

    SOCKET res = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

    if (res == INVALID\_SOCKET)

        throw\_err\_with\_code();

    return res;

}

// Создание UDP-сокета

SOCKET get\_socket\_descriptor\_udp()

{

    SOCKET res = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_IP);

    if (res == INVALID\_SOCKET)

        throw\_err\_with\_code();

    return res;

}

// Установка таймаута на получение данных

void set\_option\_timeout(SOCKET socket\_descriptor, unsigned int timeout\_ms)

{

    if (setsockopt(socket\_descriptor, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&timeout\_ms, sizeof(timeout\_ms)) == SOCKET\_ERROR)

        throw\_err\_with\_code();

}

// Заполнение структуры sockaddr\_in для подключения

sockaddr\_in get\_bind\_addr(const char \*address, unsigned short port)

{

    sockaddr\_in res;

    res.sin\_family = AF\_INET;

    res.sin\_addr.s\_addr = inet\_addr(address);

    res.sin\_port = htons(port);

    return res;

}

// Привязка сокета к адресу (используется только для UDP)

void bind\_socket\_with\_address(SOCKET socket\_descriptor, sockaddr\_in bind\_addr)

{

    if (bind(socket\_descriptor, (sockaddr \*)&bind\_addr, sizeof(bind\_addr)) == SOCKET\_ERROR)

        throw\_err\_with\_code();

}

// Подключение сокета к серверу (используется только для TCP)

void connect\_to\_server(SOCKET socket\_descriptor, sockaddr\_in bind\_addr)

{

    if (connect(socket\_descriptor, (sockaddr \*)&bind\_addr, sizeof(bind\_addr)) == SOCKET\_ERROR)

        throw\_err\_with\_code();

}

// Завершение соединения

void disconnect(SOCKET connection)

{

    std::clog << "start disconnect..." << std::endl;

    if (closesocket(connection) == SOCKET\_ERROR)

        throw\_err\_with\_code();

    WSACleanup();

    std::clog << "disconnected" << std::endl;

}

// Обработка клиента (приём файла)

void recv\_file(SOCKET con, TransportProtocol protocol);

// Запуск приёма файла от сервера

void handle\_client(SOCKET con, TransportProtocol protocol)

{

    should\_run = true;

    recv\_file(con, protocol);

}

// Генерация случайного имени файла

std::string generate\_filename()

{

    std::random\_device rd;

    std::mt19937 gen(rd());

    std::uniform\_int\_distribution<> dis(1000, 9999);

    return "file\_" + std::to\_string(dis(gen)) + ".jpg";

}

// Создание файла для записи

std::ofstream create\_output\_file(std::string filename)

{

    std::ofstream file(filename, std::ios::binary);

    if (!file)

        throw std::runtime\_error("Failed to create file: " + filename);

    return file;

}

// Запись части файла в поток

void save\_file\_fragment(std::ofstream &file, const char \*data, size\_t size)

{

    file.write(data, size);

    if (!file.good())

        throw std::runtime\_error("File write error");

}

// Приём файла по сети

void recv\_file(SOCKET con, TransportProtocol protocol)

{

    char buffer[FILE\_FRAGMENT\_SIZE];

    std::string filename = generate\_filename(); // Случайное имя для сохранения файла

    std::ofstream out\_file = create\_output\_file(filename);

    int bytes\_received = 0;

    bool is\_recved = false;

    auto a = std::chrono::high\_resolution\_clock::now(); // Засекаем время начала

    while (should\_run)

    {

        if (is\_recved && !bytes\_received)

            break;

        // Приём данных от сервера (в зависимости от протокола)

        if (protocol == UDP && (bytes\_received = recvfrom(

                                    con,

                                    buffer,

                                    sizeof(buffer),

                                    0,

                                    nullptr,

                                    nullptr)) != SOCKET\_ERROR ||

            protocol == TCP && (bytes\_received = recv(

                                    con,

                                    buffer,

                                    sizeof(buffer),

                                    0)) != SOCKET\_ERROR)

        {

            is\_recved = true;

            save\_file\_fragment(out\_file, buffer, bytes\_received);

        }

        else

        {

            std::cerr << "not get answer from server: " << GetLastError << std::endl;

            break;

        }

    }

    auto b = std::chrono::high\_resolution\_clock::now(); // Засекаем время окончания

    std::clog << "Answer accepted\n"

              << "Time: " << std::chrono::duration\_cast<std::chrono::milliseconds>(b - a).count() / 1000.0 << " s." << std::endl;

}

// Главная функция — подключение, приём файла, завершение

int main()

{

    auto con = connect("192.168.41.96", 0x8080, TCP); // Подключение к серверу

    handle\_client(con, TCP); // Приём данных от сервера

    disconnect(con); // Завершение соединения

    return 0;

}