

**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
Игнатъев 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 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 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. Поточковые 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. Далее можно использовать любые остальные функции этой библиотеки, иначе возвращает код возникшей ошибки. `wVersionRequested` – 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 именуется; `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.

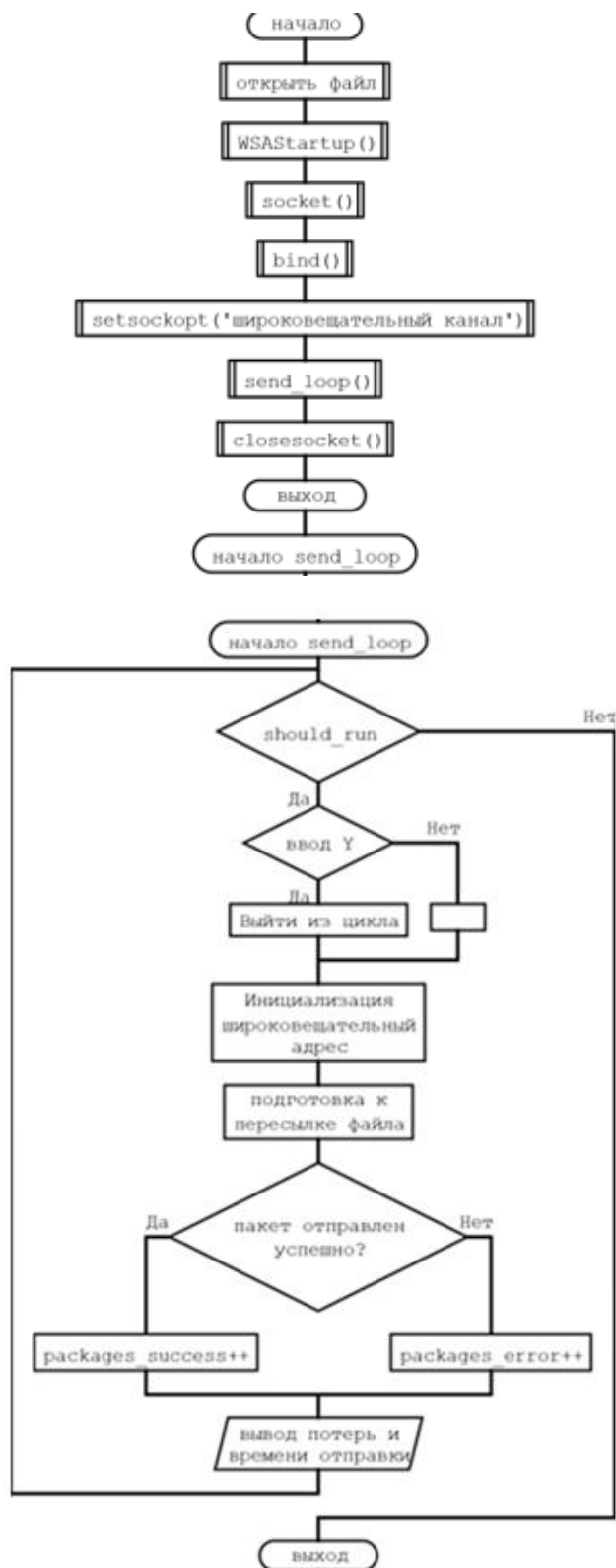
Для реализации серверной части использованы функции `socket` (у разных протоколов своя настройка) и `bind`. В части TCP сервера, применена `listen` для прослушивания сокета, после запускается поток на подключение пользователей с помощью функции `accept`.

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.

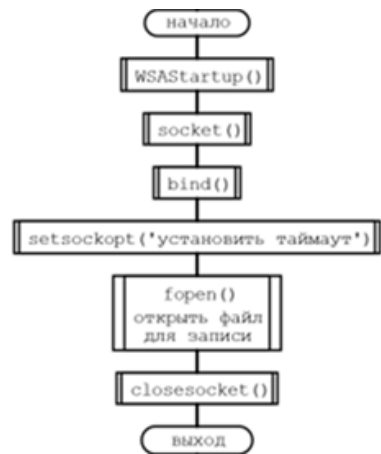
Пользователь также подключается через `socket`, в UDP выполняется `bind`, а TCP применяет `connect`. В дескрипторе реализации клиента адрес указывается в зависимости от выбранного протокола (UDP - собственный, TCP - сервера).

Разработка программы. Блок-схемы программы

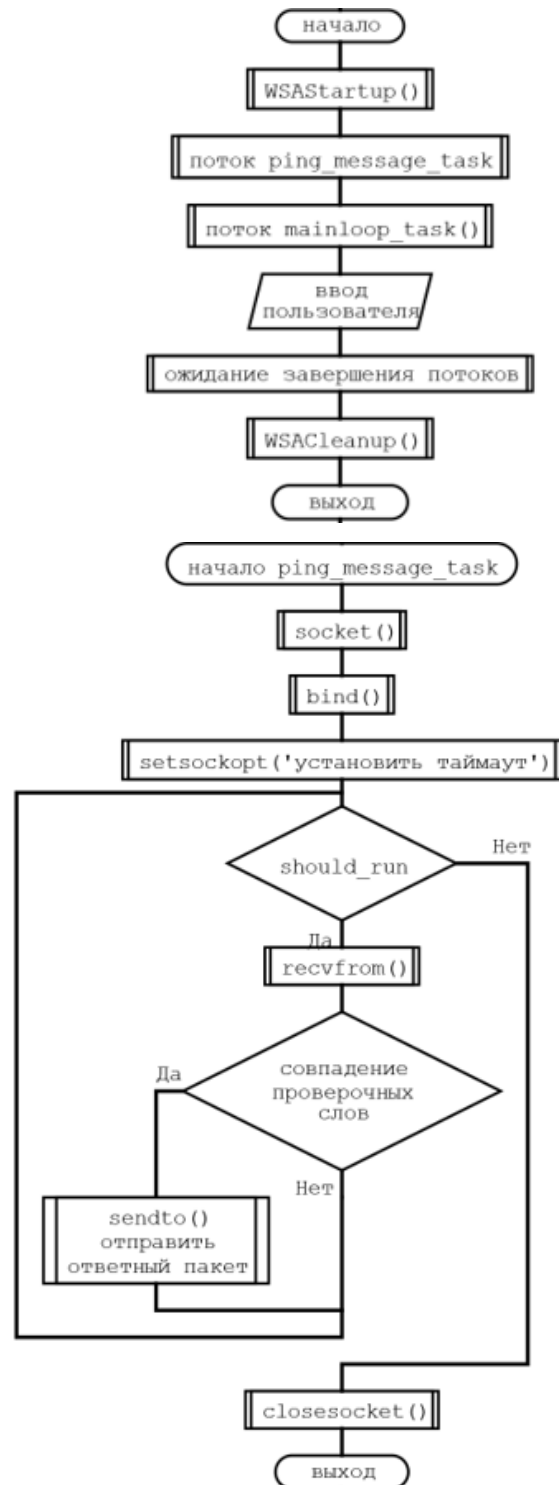
Сервер UDP

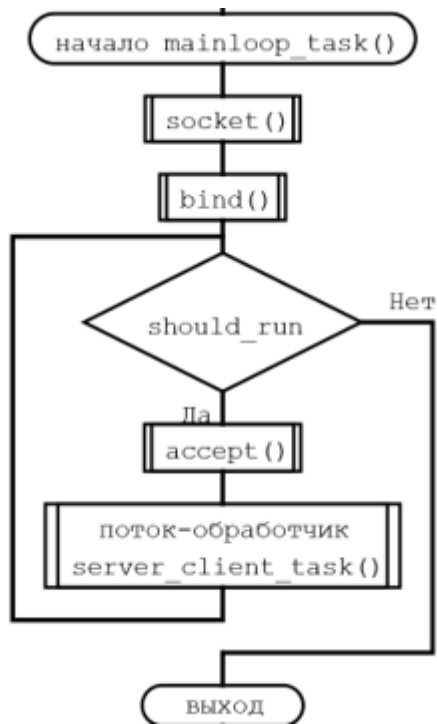


Клиент UDP

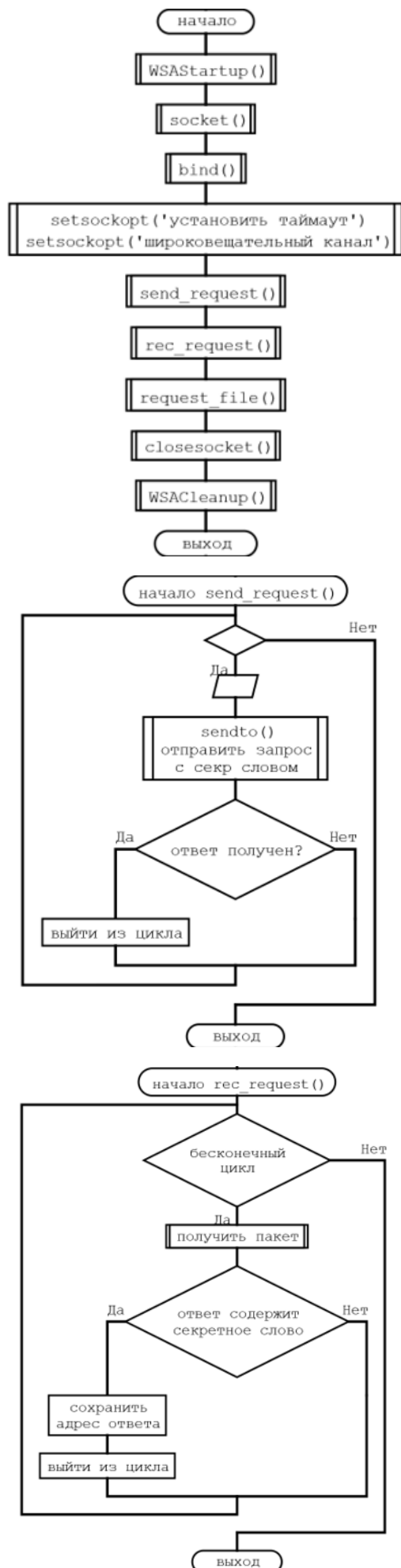


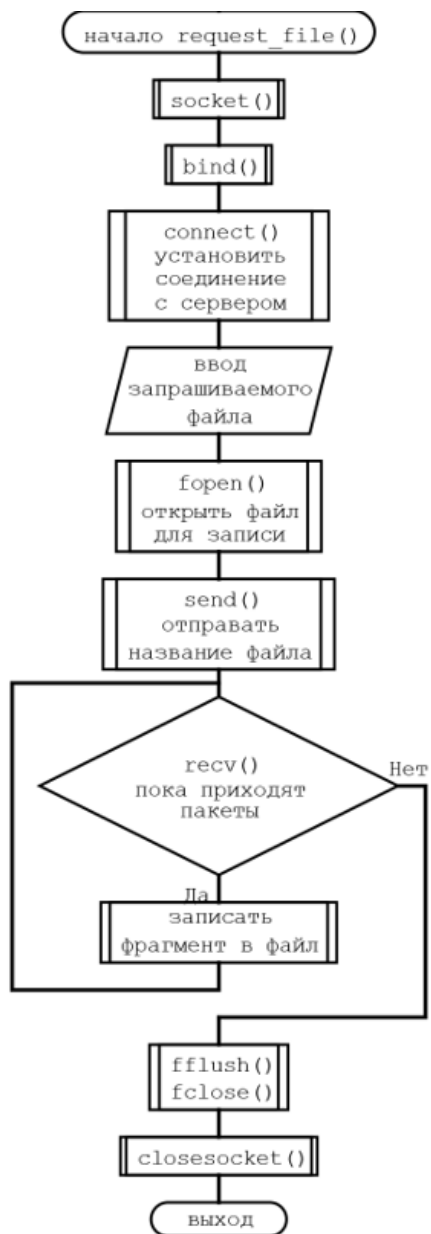
Сервер TCP





Клиент ТСП





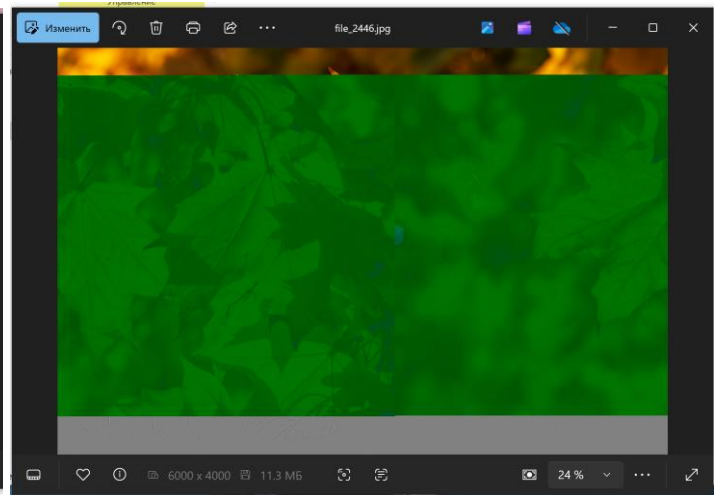
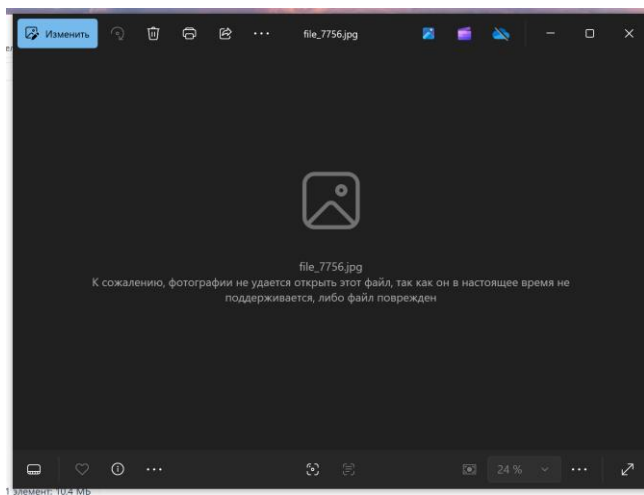
Анализ функционирования программы

Пример переданного изображения

Исходное изображение

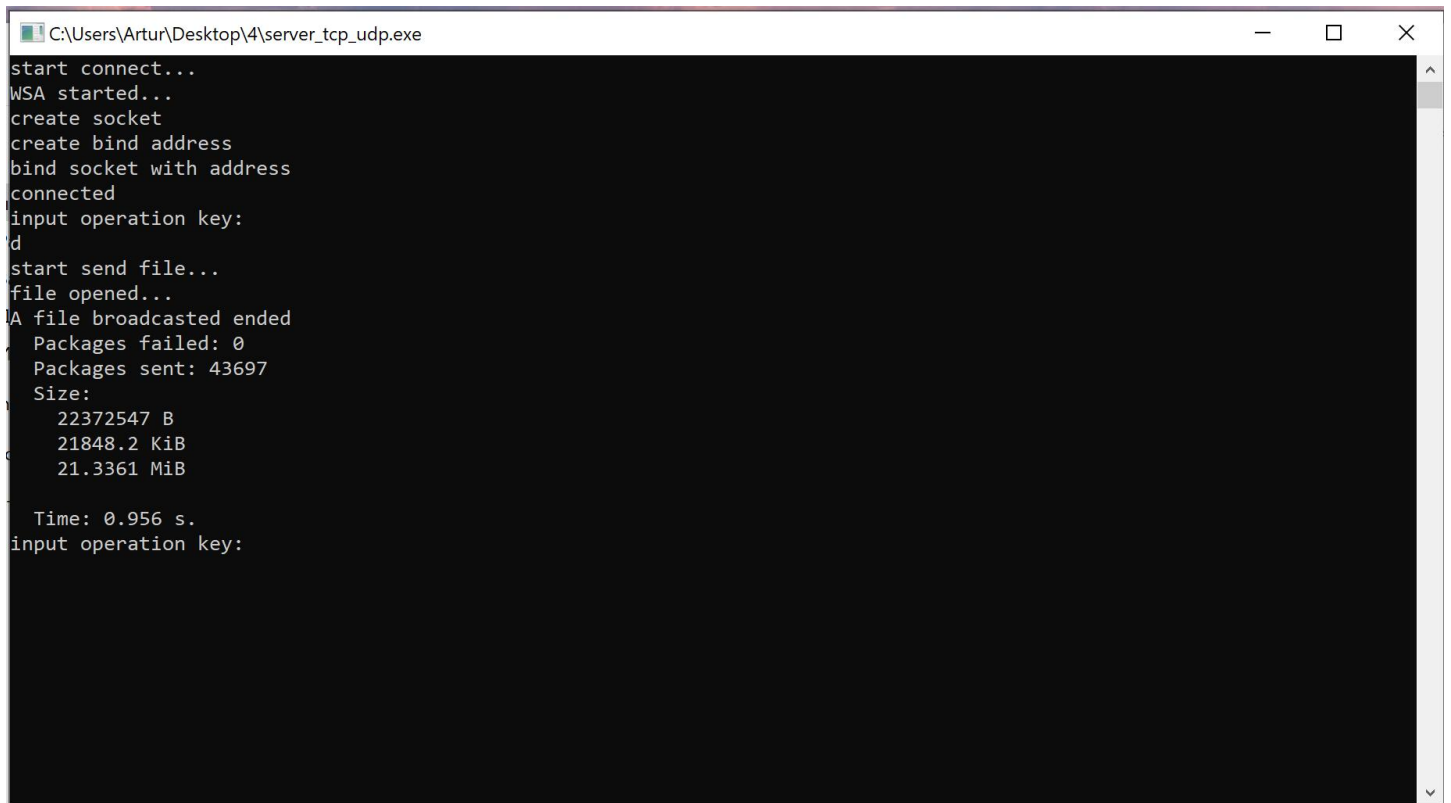


Переданные изображения UDP



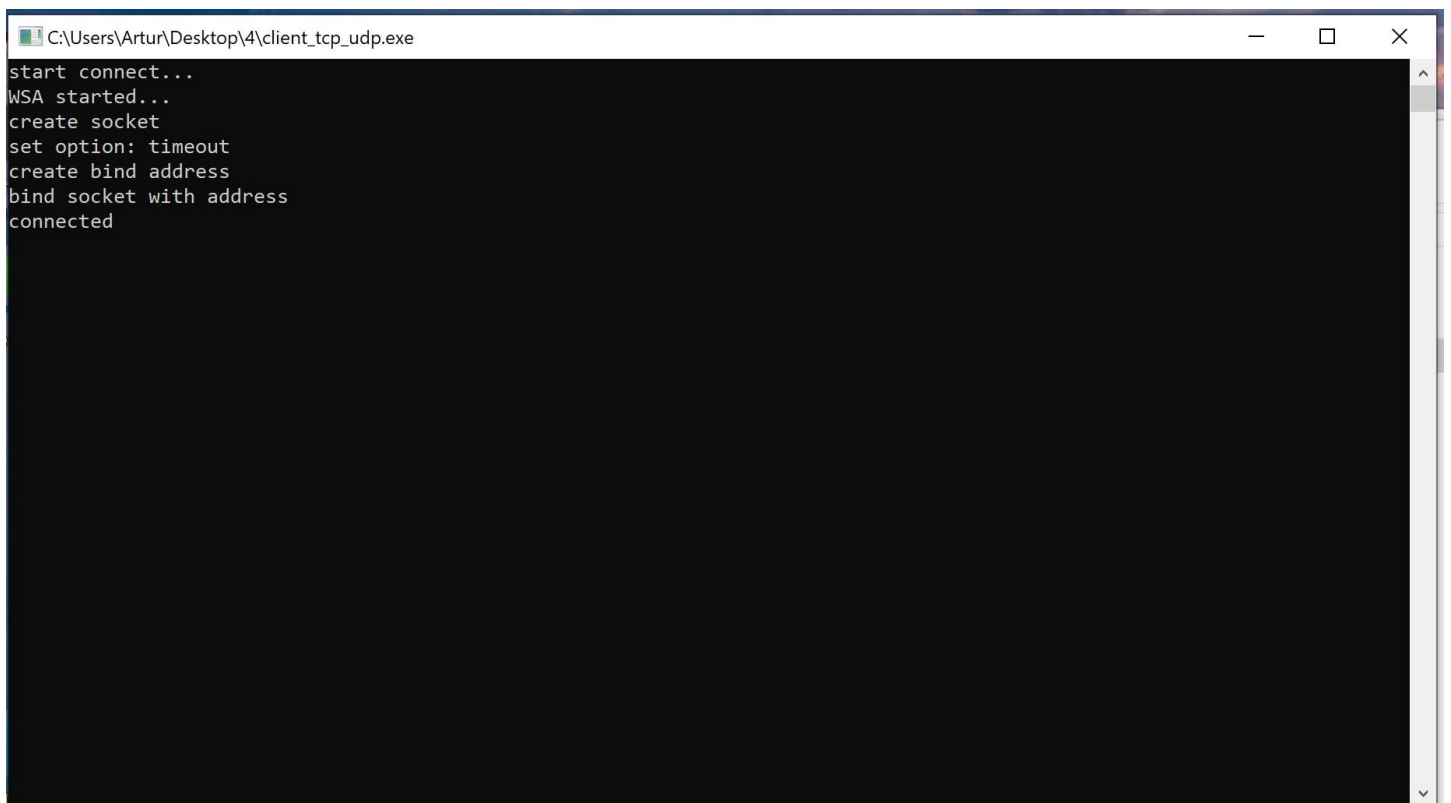
Работа программы:

Сервер



```
C:\Users\Artur\Desktop\4\server_tcp_udp.exe
start connect...
WSA started...
create socket
create bind address
bind socket with address
connected
input operation key:
d
start send file...
file opened...
A file broadcasted ended
Packages failed: 0
Packages sent: 43697
Size:
22372547 B
21848.2 KiB
21.3361 MiB
Time: 0.956 s.
input operation key:
```

Клиент



```
C:\Users\Artur\Desktop\4\client_tcp_udp.exe
start connect...
WSA started...
create socket
set option: timeout
create bind address
bind socket with address
connected
```






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:

 file_2139.jpg	20.05.2025 9:08	Файл "JPG"	8 494 КБ
 file_5545.jpg	20.05.2025 9:07	Файл "JPG"	9 798 КБ
 file_6999.jpg	20.05.2025 9:08	Файл "JPG"	13 895 КБ
 file_7756.jpg	20.05.2025 9:02	Файл "JPG"	10 737 КБ
 file_9423.jpg	20.05.2025 9:07	Файл "JPG"	13 431 КБ

Клиент 2:

 file_2446.jpg	20.05.2025 9:02	Файл "JPG"	11 608 КБ
 file_2453.jpg	20.05.2025 9:07	Файл "JPG"	9 740 КБ
 file_6757.jpg	20.05.2025 9:07	Файл "JPG"	12 046 КБ
 file_6997.jpg	20.05.2025 9:07	Файл "JPG"	10 534 КБ
 file_8638.jpg	20.05.2025 9:08	Файл "JPG"	10 181 КБ

Переданные изображения ТСР



Работа программы:

Сервер

```
C:\Users\Artur\Desktop\4\server_tcp_udp.exe
start connect...
WSA started...
create socket
create bind address
bind socket with address
connected
listen started
start accept clients
input operation key:
client accepted
client accepted
d
start send file...
file opened...
A file broadcasted ended
  Packages failed: 0
  Packages sent: 43697
  Size:
    22372547 B
    21848.2 KiB
    21.3361 MiB
  Time: 0.571 s.
start send file...
file opened...
A file broadcasted ended
  Packages failed: 0
  Packages sent: 43697
  Size:
    22372547 B
    21848.2 KiB
    21.3361 MiB
  Time: 0.431 s.
input operation key:
_
```

Клиент

```
C:\Users\Artur\Desktop\4\client_tcp_udp.exe
start connect...
WSA started...
create socket
set option: timeout
create bind address
connect to server
connected
```








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:

 file_1147.jpg	20.05.2025 13:17	Файл "JPG"	21 849 КБ
 file_2162.jpg	20.05.2025 13:25	Файл "JPG"	21 849 КБ
 file_4291.jpg	20.05.2025 9:19	Файл "JPG"	21 849 КБ
 file_4500.jpg	20.05.2025 13:23	Файл "JPG"	21 849 КБ
 file_5767.jpg	20.05.2025 13:25	Файл "JPG"	21 849 КБ
 file_9711.jpg	20.05.2025 13:24	Файл "JPG"	21 849 КБ
 image.jpg	15.04.2025 6:22	Файл "JPG"	21 849 КБ

Клиент 2:

 file_3542.jpg	20.05.2025 13:17	Файл "JPG"	21 849 КБ
 file_4930.jpg	20.05.2025 13:23	Файл "JPG"	21 849 КБ
 file_5536.jpg	20.05.2025 13:24	Файл "JPG"	21 849 КБ
 file_7563.jpg	20.05.2025 13:25	Файл "JPG"	21 849 КБ
 file_9676.jpg	20.05.2025 9:19	Файл "JPG"	21 849 КБ
 file_9966.jpg	20.05.2025 13:25	Файл "JPG"	21 849 КБ

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 клиентов, 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;  
}
```
