

**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 №3**

по дисциплине: Компьютерные сети

тема: ««Программирование протокола IP с использованием библиотеки Winsock»»

Выполнил: ст. группы ПВ-223

Игнатъев Артур Олегович

Проверили:

Рубцов Константин Анатольевич

Белгород 2025 г.

**Цель работы:** изучить принципы и характеристику протокола IP и разработать программу для приема/передачи пакетов с использованием библиотеки Winsock.

### **Краткие теоретические сведения**

Internet Protocol или IP (англ. internet protocol - межсетевой протокол) - маршрутизируемый сетевой протокол сетевого уровня семейства TCP/IP.

Протокол IP используется для негарантированной доставки данных, разделяемых на так называемые пакеты от одного узла сети к другому. Это означает, что на уровне этого протокола (третий уровень сетевой модели OSI) не даётся гарантий надёжной доставки пакета до адресата. В частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (когда приходят две копии одного пакета - в реальности это бывает крайне редко), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прибыть вовсе. Гарантии безошибочной доставки пакетов дают протоколы более высокого (транспортного) уровня сетевой модели OSI - например, TCP - который использует IP в качестве транспорта.

Обычно в сетях используется IP четвёртой версии, также известный как IPv4. В протоколе IP этой версии каждому узлу сети ставится в соответствие IP-адрес длиной 4 октета (1 октет состоит из 8 бит). При этом компьютеры в подсетях объединяются общими начальными битами адреса. Количество этих бит, общее для данной подсети, называется маской подсети (ранее использовалось деление пространства адресов по классам — А, В, С; класс сети определяется диапазоном значений старшего октета и определяет число адресуемых узлов в данной сети).

IP-пакет представляет собой форматированный блок информации, передаваемый по вычислительной сети. Соединения вычислительных сетей, которые не поддерживают пакеты, такие как традиционные соединения типа «точка-точка» в телекоммуникациях, просто передают данные в виде последовательности байтов, символов или битов. При использовании пакетного форматирования сеть может передавать длинные сообщения более надёжно и эффективно.

IP-адрес имеет длину 4 байта и обычно записывается в виде четырех чисел, представляющих значения каждого байта в десятичной форме, и разделенных точками, например, 128.10.2.30 – традиционная десятично-точечная форма представления адреса, 10000000 00001010 00000010 00011110 - двоичная форма представления этого же адреса.

Классы сетей IP IP-адреса разделяются на 5 классов: A, B, C, D, E. Адреса классов A, B и C делятся на две логические части: номер сети и номер узла. На рис. 3.1 показана структура IP-адреса разных классов.



Рис. 3.1. Структура IP-адреса разных классов

Идентификатор сети, также называемый адресом сети, обозначает один сетевой сегмент в более крупной объединенной сети, использующей протокол TCP/IP. IP-адреса всех систем, подключенных к одной сети, имеют один и тот же идентификатор сети. Этот идентификатор также используется для уникального обозначения каждой сети в более крупной объединенной сети. Идентификатор узла, также называемый адресом узла, определяет узел TCP/IP (рабочую станцию, сервер, маршрутизатор или другое устройство) в пределах каждой сети. Идентификатор узла уникальным образом обозначает систему в том сегменте сети, к которой она подключена.

У адресов класса A старший бит установлен в 0. Длина сетевого префикса 8 бит. Для номера узла выделяется 3 байта (24 бита). Таким образом, в классе A может быть 126 сетей ( $2^7 - 2$ , два номера сети имеют специальное значение). Каждая сеть этого класса может поддерживать максимум 16777214 узлов ( $2^{24} - 2$ ). Адресный блок класса A может содержать максимум 231 уникальных адресов, в то время как в протоколе IP версии 4 возможно существование 232 адресов. Таким образом, адресное пространство класса A занимает 50% всего адресного пространства протокола IP версии 4. Адреса класса A предназначены для использования в больших сетях, с большим количеством узлов. На данный момент все адреса класса A распределены.

У адресов класса B два старших бита установлены в 1 и 0 соответственно. Длина сетевого префикса – 16 бит. Поле номера узла тоже имеет длину 16 бит. Таким образом, число сетей класса B равно 16384 (214); каждая сеть класса B может поддерживать до

65534 узлов (216 - 2). Адресный блок сетей класса В предназначен для применения в сетях среднего размера. У адресов класса С три старших бита установлены в 1, 1 и 0 соответственно. Префикс сети имеет длину 24 бита, номер узла - 8 бит. Максимально возможное количество сетей класса С составляет 2097152 (2<sup>21</sup>). Каждая сеть может поддерживать максимум 254 узла (2<sup>8</sup> - 2). Класс С предназначен для сетей с небольшим количеством узлов.

Адреса класса D представляют собой специальные адреса, не относящиеся к отдельным сетям. Первые 4 бита этих адресов равны 1110. Таким образом, значение первого октета этого диапазона адресов находится в пределах от 224 до 239. Адреса класса D используются для многоадресных пакетов, с помощью которых во многих разных протоколах данные передаются многочисленным группам узлов. Эти адреса можно рассматривать как заранее запрограммированные в логической структуре большинства сетевых устройств. Это означает, что при обнаружении в пакете адреса получателя такого типа устройство на него обязательно отвечает. Например, если один из хостов передает пакет с IP-адресом получателя 224.0.0.5, на него отвечают все маршрутизаторы (использующие протокол OSPF), которые находятся в сегменте сети с этим адресом Ethernet.

Адреса в диапазоне 240.0.0.0 - 255.255.255.255 называются адресами класса E. Первый октет этих адресов начинается с битов 1111. Эти адреса зарезервированы для будущих дополнений в схеме адресации IP. Но возможность того, что эти дополнения когда-либо будут приняты, находится под вопросом, поскольку уже появилась версия 6 протокола IP (IPv6).

Некоторые IP-адреса являются зарезервированными. Для таких адресов существуют следующие соглашения об их особой интерпретации:

1. Если все биты IP-адреса установлены в нуль, то он обозначает адрес данного устройства.
2. Если в поле номера сети стоят нули, то считается, что получатель принадлежит той же самой сети, что и отправитель.

3. Если все биты IP-адреса установлены в единицу, то пакет с таким адресом должен рассылаться всем узлам, находящимся в той же сети, что и отправитель. Такая рассылка называется ограниченным широковещательным сообщением.

4. Если все биты номера узла установлены в нуль, то пакет предназначен для данной сети.

5. Если все биты в поле номера узла установлены в единицу, то пакет рассылается всем узлам сети с данным номером сети. Такая рассылка называется широковещательным сообщением.

6. Если первый октет адреса равен 127, то адрес обозначает тот же самый узел. Такой адрес используется для взаимодействия процессов на одной и той же машине (например, для целей тестирования). Этот адрес имеет название возвратного.

Поля номеров сети и подсети образуют расширенный сетевой префикс. Для выделения расширенного сетевого префикса используется маска подсети (subnet mask).

Маска подсети – это 32-разрядное двоичное число, в разрядах расширенного префикса содержащая единицу; в остальных разрядах находится ноль. Расширенный сетевой префикс получается побитным сложением по модулю два (операция XOR) IP адреса и маски подсети.

При таком построении очевидно, что число подсетей представляет собой степень двойки –  $2^n$ , где  $n$  - длина поля номера подсети. Таким образом, характеристики IP-адреса полностью задаются собственно IP-адресом и маской подсети.

Стандартные маски подсетей для классов А, В, С приведены в табл. 3.1.

Таблица 3.1.

Стандартные маски подсетей для классов А, В, С

| Класс адреса | Биты маски подсети                     | Маска подсети |
|--------------|--|---------------|
| Класс А      | 11111111 00000000 00000000<br>00000000 | 255.0.0.0     |
| Класс В      | 11111111 11111111 00000000<br>00000000 | 255.255.0.0   |
| Класс С      | 11111111 11111111 11111111<br>00000000 | 255.255.255.0 |

Для упрощения записи применяют следующую нотацию (так называемая CIDR нотация): IP-адрес/длина расширенного сетевого префикса. Например, адрес 192.168.0.1 с маской 255.255.255.0 будет в данной нотации выглядеть как 192.168.0.1/24 (24 – это число единиц, содержащихся в маске подсети).

Для разбития сети на подсети необходимо найти минимальную степень двойки, большую или равную числу требуемых подсетей. Затем эту степень прибавить к префиксу сети. Количество IP-адресов в каждой подсети будет на 2 меньше теоретически возможного, потому что сеть должна будет вместить адрес сети и бродкастовый адрес.

### Основные функции API для работы с протоколом IP.

**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. 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 именуется; name – указатель на структуру имени сокета; namelen – размер, в байтах, структуры name.

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 IP 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.

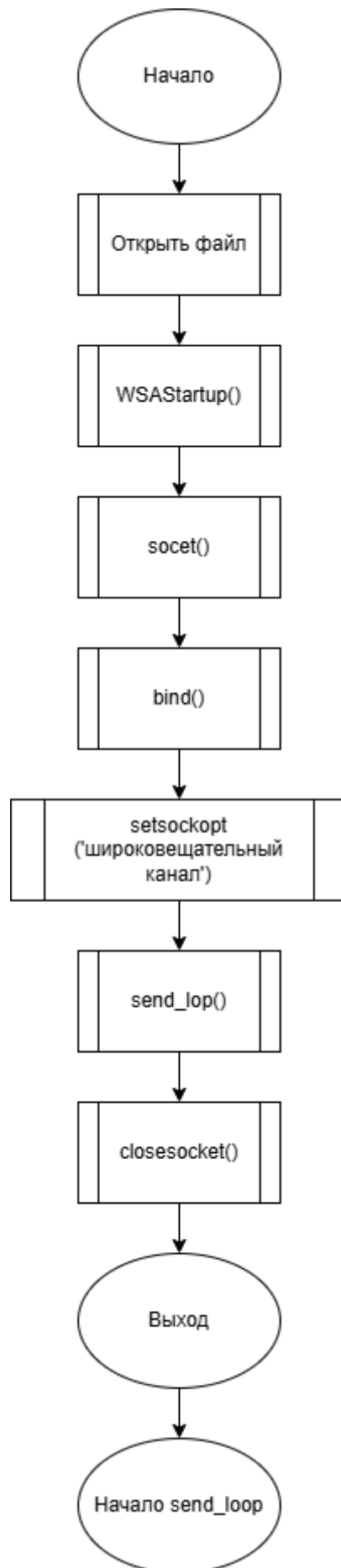
Прием данных по протоколу IP осуществляется с помощью функции **recvfrom** (SOCKET s, char FAR\* buf, int len, int flags, struct sockaddr FAR\* from, int FAR\* fromlen). Если операция выполнена успешно, возвращает количество полученных байт, иначе возвращает SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

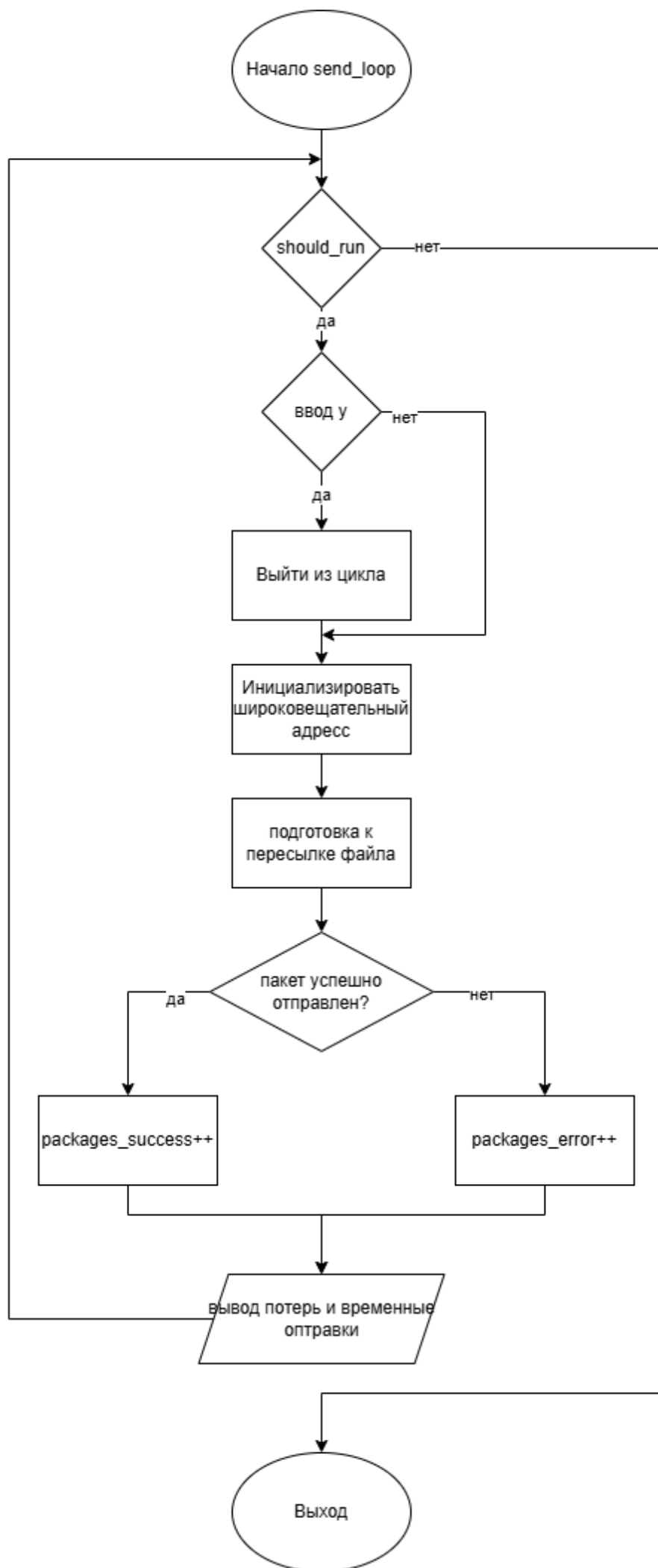
Функция **closesocket**(SOCKET s) служит для закрытия сокета. Возвращает 0, если операция была выполнена успешно, иначе возвращает SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.



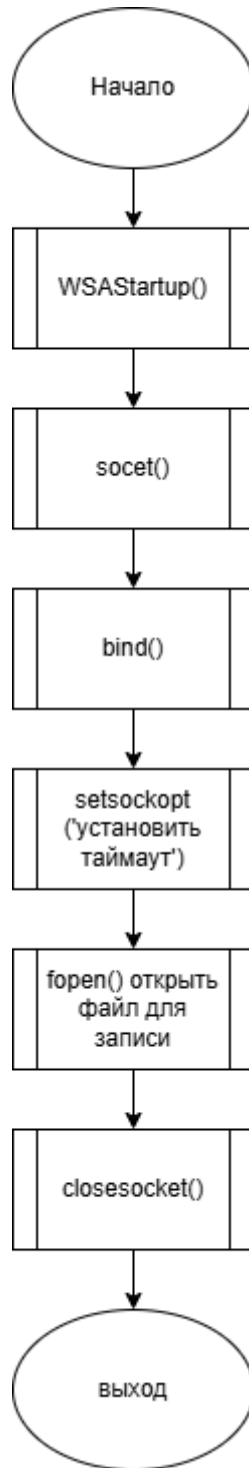
## Разработка программы. Блок-схемы программы

### Сервер IP





## Клиент IP



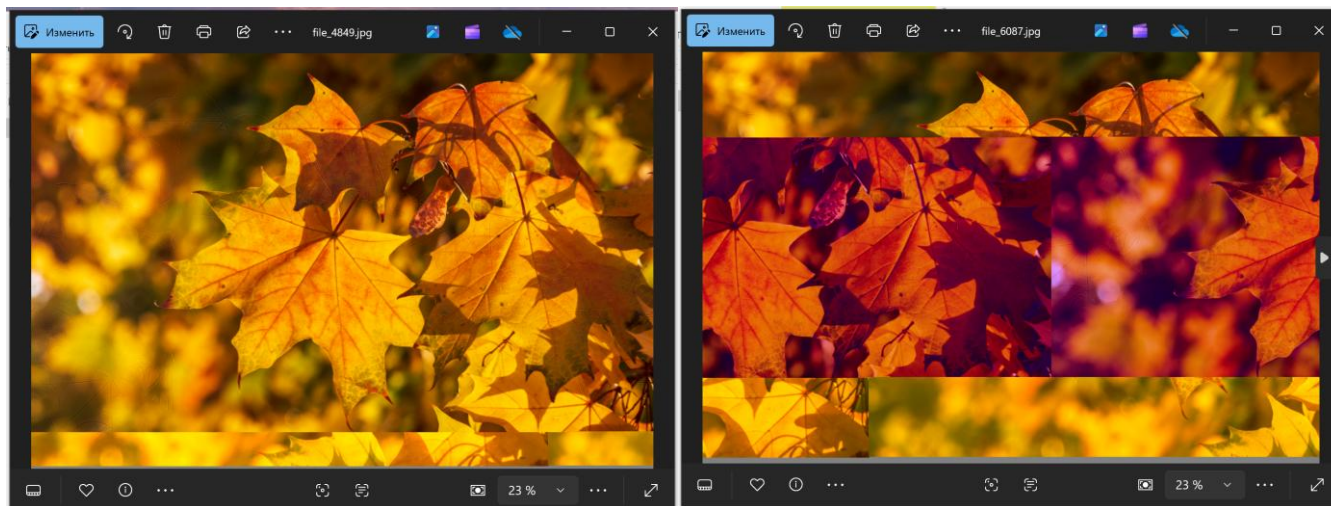
## Анализ функционирования программы

Пример переданного изображения

Исходное изображение

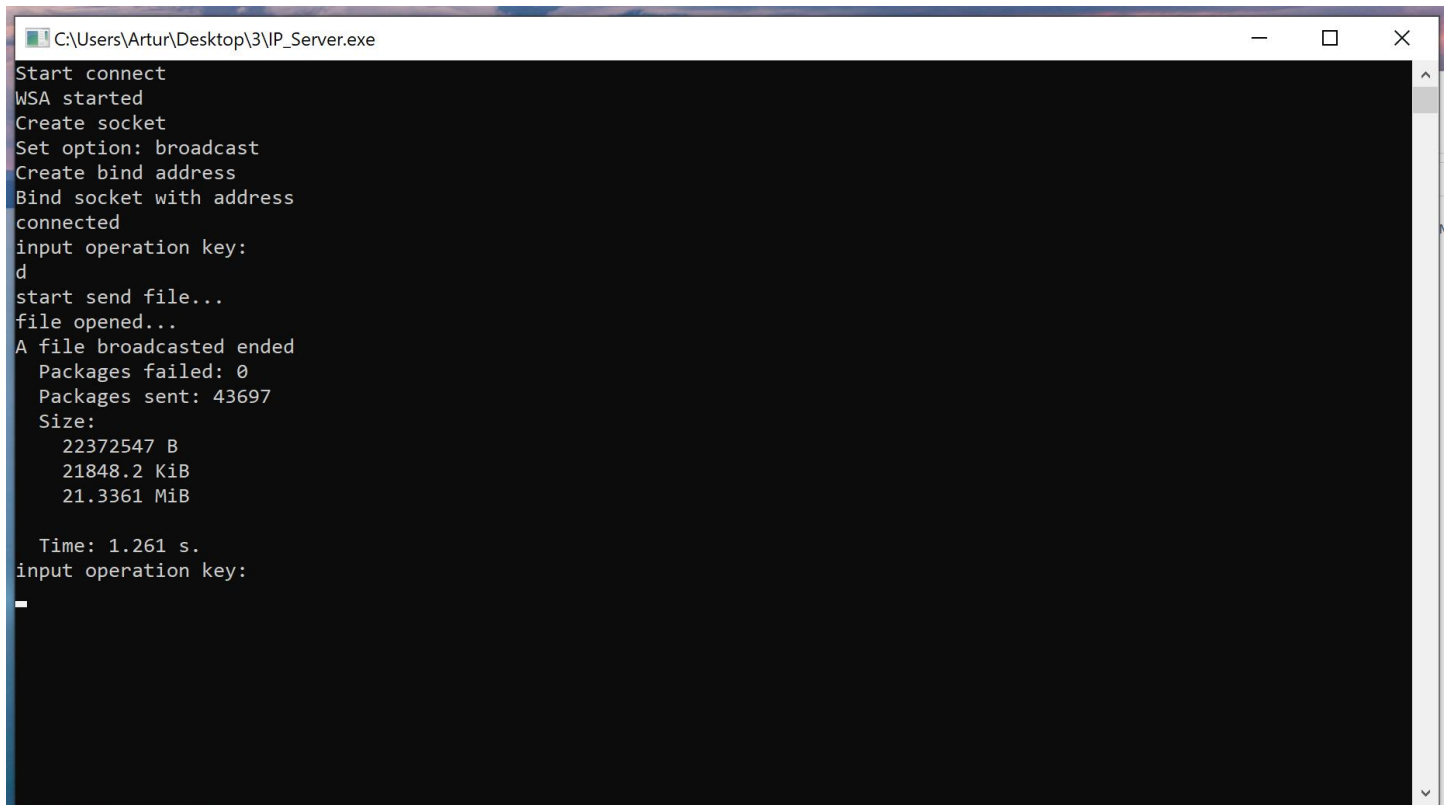


Переданные изображения



## Работа программы:

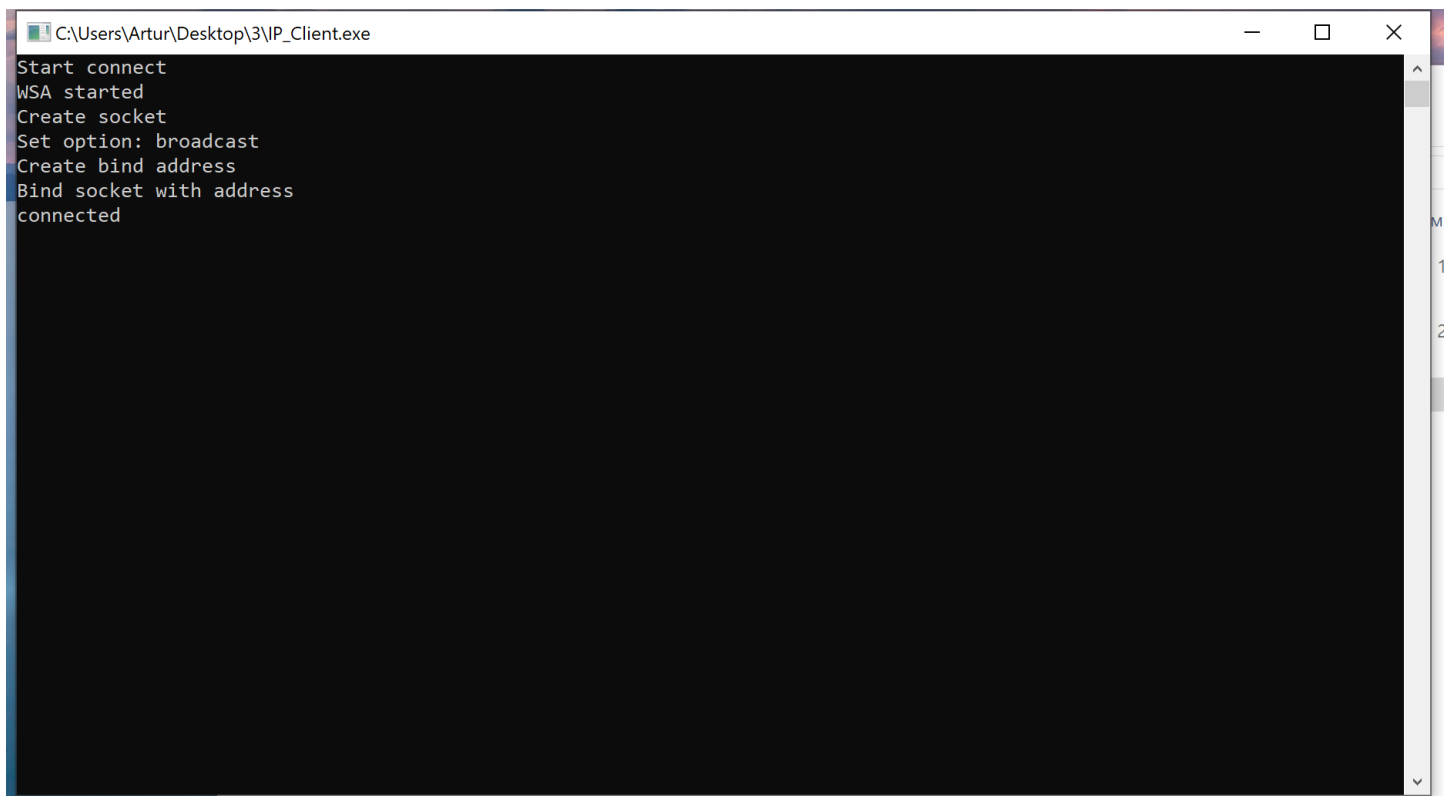
### Сервер



```
C:\Users\Artur\Desktop\3\IP_Server.exe
Start connect
WSA started
Create socket
Set option: broadcast
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: 1.261 s.
input operation key:
█
```

### Клиент



```
C:\Users\Artur\Desktop\3\IP_Client.exe
Start connect
WSA started
Create socket
Set option: broadcast
Create bind address
Bind socket with address
connected
```






Передача осуществлялась 2м клиентам

|             | Время, сек |
|-------------|------------|
| Передача №1 | 1.261      |
| Передача №2 | 0.943      |
| Передача №3 | 0.896      |
| Передача №4 | 0.887      |
| Передача №5 | 0.9        |
| Среднее     | 0.9774     |






| Размер изображения Мбайт | Скорость передачи, Мбит/с |
|--------------------------|---------------------------|
| 21.3361                  | 174,6355637               |

Размер полученных изображений:

Клиент 1:

|   |                 |            |           |
|---|-----------------|------------|-----------|
|  file_3894.jpg | 06.05.2025 9:42 | Файл "JPG" | 11 648 КБ |
|  file_4849.jpg | 06.05.2025 9:31 | Файл "JPG" | 11 740 КБ |
|  file_7865.jpg | 06.05.2025 9:42 | Файл "JPG" | 11 682 КБ |
|  file_8134.jpg | 06.05.2025 9:41 | Файл "JPG" | 13 744 КБ |
|  file_8159.jpg | 06.05.2025 9:40 | Файл "JPG" | 12 774 КБ |

Клиент 2:

|   |                 |            |           |
|---|-----------------|------------|-----------|
|  file_1796.jpg | 06.05.2025 9:40 | Файл "JPG" | 8 380 КБ  |
|  file_3202.jpg | 06.05.2025 9:41 | Файл "JPG" | 9 059 КБ  |
|  file_4364.jpg | 06.05.2025 9:42 | Файл "JPG" | 12 124 КБ |
|  file_6087.jpg | 06.05.2025 9:31 | Файл "JPG" | 12 479 КБ |
|  file_6835.jpg | 06.05.2025 9:42 | Файл "JPG" | 9 825 КБ  |

Вывод по работе IP: Скорость передачи по сравнению с протоколом IPX выросла в 1.55 раз и изображение возможно открыть и просмотреть, хоть и полученное изображение передано с потерями. По сравнению с протоколом SPX скорость выросла в 21.7 раз. Тестирование проводилось на виртуальных машинах Oracle VirtualBox с установленной системой Windows 10 с последними обновлениями. На виртуальных машинах был выставлен режим моста. Каждой машине было выделено 2 ядра и 4гб ОЗУ.

**Вывод:** в ходе лабораторной изучили принципы и характеристику протокола IP и разработать программу для приема/передачи пакетов с использованием библиотеки Winsock.

Код программы:

Файл IP\_Client.cpp

```
#include <WinSock2.h>
#include <winsock.h>
#include <ws2tcpip.h>
#include <iostream>
#include <exception>
#include <string>
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <random>

#define FILE_FRAGMENT_SIZE 512

bool should_run = false;

void throw_err_with_code()
{
    std::string err_msg = "error with code: ";
    err_msg += std::to_string(WSAGetLastError());
    throw std::runtime_error(err_msg);
}

void startup_wsa();
SOCKET get_socket_descriptor();
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);

SOCKET connect(const char *address, unsigned short port)
{
    std::clog << "Start connect" << std::endl;

    startup_wsa();

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

    SOCKET socket_descriptor = get_socket_descriptor();
    std::clog << "Create socket" << std::endl;

    set_option_timeout(socket_descriptor, 10000);
    std::clog << "Set option: broadcast" << std::endl;

    sockaddr_in bind_addr = get_bind_addr(address, port);
    std::clog << "Create bind address" << std::endl;

    bind_socket_with_address(socket_descriptor, bind_addr);
    std::clog << "Bind socket with address\nconnected" << std::endl;

    return socket_descriptor;
}

void startup_wsa()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 0);

    if (WSAStartup(wVersionRequested, &wsaData) == SOCKET_ERROR)
        throw_err_with_code();
}
```

```

SOCKET get_socket_descriptor()
{
    SOCKET res = socket(
        AF_INET,
        SOCK_DGRAM,
        IPPROTO_IP);

    if (res == INVALID_SOCKET)
        throw_err_with_code();

    return res;
}

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();
}

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;
}

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();
}

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;
}

void recv_file(SOCKET con);

void handle_client(SOCKET con)
{
    should_run = true;

    recv_file(con);
}

std::string generate_filename();
std::ofstream create_output_file(std::string filename);
void save_file_fragment(std::ofstream &file, const char *data, size_t size);

void recv_file(SOCKET con)

```



```

{
    char buffer[FILE_FRAGMENT_SIZE];
    std::string filename = generate_filename();
    std::ofstream out_file = create_output_file(filename);
    int bytes_received;

    auto a = std::chrono::high_resolution_clock::now();
    while (should_run && (bytes_received = recvfrom(
        con,
        buffer,
        sizeof(buffer),
        0,
        nullptr,
        nullptr)) != SOCKET_ERROR)
        save_file_fragment(out_file, buffer, bytes_received);
    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;
}

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";
}

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;
}

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");
    }
}

int main()
{
    auto con = connect("192.168.41.97", 0x8081);

    handle_client(con);

    disconnect(con);

    return 0;
}

```

## Файл IP\_Server.cpp

```
#include <WinSock2.h>
#include <winsock.h>
#include <ws2tcpip.h>
#include <iostream>
#include <exception>
#include <string>
#include <iostream>
#include <fstream>
#include <chrono>
#include <pthread.h>

#define FILE_FRAGMENT_SIZE 512

bool should_run = false;

struct server_args
{
    SOCKET server_socket;
    sockaddr_in client_sockaddr;
};

void throw_err_with_code()
{
    std::string err_msg = "error with code: ";
    err_msg += std::to_string(WSAGetLastError());
    throw std::runtime_error(err_msg);
}

void startup_wsa();
SOCKET get_socket_descriptor();
void set_option_timeout(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);

SOCKET connect(const char *address, unsigned short port)
{
    std::clog << "Start connect" << std::endl;

    startup_wsa();

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

    SOCKET socket_descriptor = get_socket_descriptor();
    std::clog << "Create socket" << std::endl;

    set_option_timeout(socket_descriptor);
    std::clog << "Set option: broadcast" << std::endl;

    sockaddr_in bind_addr = get_bind_addr(address, port);
    std::clog << "Create bind address" << std::endl;

    bind_socket_with_address(socket_descriptor, bind_addr);
    std::clog << "Bind socket with address\nconnected" << std::endl;

    return socket_descriptor;
}

void startup_wsa()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD(2, 0);

    if (WSAStartup(wVersionRequested, &wsaData) == SOCKET_ERROR)
```

```

        throw_err_with_code();
    }

SOCKET get_socket_descriptor()
{
    SOCKET res = socket(
        AF_INET,
        SOCK_DGRAM,
        IPPROTO_IP);

    if (res == INVALID_SOCKET)
        throw_err_with_code();

    return res;
}

void set_option_timeout(SOCKET socket_descriptor)
{
    bool broadcast = true;
    if (
        setsockopt(
            socket_descriptor,
            SOL_SOCKET,
            SO_BROADCAST,
            (char *)&broadcast,
            sizeof(broadcast)) == SOCKET_ERROR)
        throw_err_with_code();
}

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;
}

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();
}

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;
}

void send_file_with_input(SOCKET con, sockaddr_in client_addr);
void send_file_by_default(SOCKET con, sockaddr_in client_addr);

void *start_loop_server(void *args)
{
    struct server_args typed_args = *((server_args *)args);

    SOCKET con = typed_args.server_socket;
    sockaddr_in client_addr = typed_args.client_sockaddr;

```

```

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':
            send_file_with_input(con, client_addr);
            break;

        case 'd':
        case 'D':
            send_file_by_default(con, client_addr);
            break;

        case 'c':
        case 'C':
            should_run = false;
            break;

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

return nullptr;
}

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;
}

void send_file_by_path(SOCKET con, sockaddr_in client_addr, std::string file_path);

void send_file_with_input(SOCKET con, sockaddr_in client_addr)
{
    std::cout << "input path to file from current dir:" << std::endl;
    std::string file_path;
    std::cin >> file_path;

    send_file_by_path(con, client_addr, file_path);
}

void send_file_by_default(SOCKET con, sockaddr_in client_addr)
{
    send_file_by_path(con, client_addr, std::string("./image.jpg"));
}

std::ifstream *new_get_file(std::string path);
std::ifstream get_file(std::string path);
void send_file(SOCKET con, sockaddr_in client_addr, std::ifstream *file);

void send_file_by_path(SOCKET con, sockaddr_in client_addr, std::string file_path)
{
    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);

file->close();
delete file;
}

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;
}

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;
}

void send_file(SOCKET con, sockaddr_in client_addr, std::ifstream *file)
{
    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;
        int total_bytes = 0;

        if (sendto(
            con,
            buffer,
            bytes_read,
            0,
            (sockaddr *)&client_addr,
            sizeof(client_addr)) == 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;
}

int main()
{
    auto con = connect("192.168.41.96", 0x8000);
    auto client_addr = get_bind_addr("192.168.41.255", 0x8081);

    struct server_args args = {con, client_addr};
    auto server_thread = run_server(args);

    pthread_join(server_thread, NULL);

    disconnect(con);

    return 0;
}
```

---