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 №2

по дисциплине: Компьютерные сети
тема: «Протокол сетевого уровня IPX»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
Рубцов Константин Анатольевич

Белгород 2025 г.

Лабораторная работа №2
Протокол сетевого уровня IPX
Вариант 6

Цель работы: изучить протоколы IPX/SPX, основные функции библиотеки Winsock и разработать программу для приема/передачи пакетов.

Краткие теоретические сведения

Протокол **IPX** (Internetwork Packet Exchange) является оригинальным протоколом сетевого уровня стека Novell, разработанным в начале 80-х годов на основе протокола Internetwork Datagram Protocol (IDM) компании Xerox

Протокол IPX соответствует сетевому уровню модели OSI и поддерживает только дейтаграммный (без установления соединений) способ обмена сообщениями. В сети NetWare самая быстрая передача данных при наиболее экономном расходовании памяти реализуется именно протоколом IPX.

Для надежной передачи пакетов используется протокол транспортного уровня **SPX** (Sequenced Packet Exchange), который работает с установлением соединения и восстанавливает пакеты при их потере или повреждении. Если по каким-то причинам пакет не дошел до получателя, выполняется его повторная передача. Следовательно, последовательность отправления совпадает с последовательностью получения пакетов. Обмен пакетами на уровне сеанса связи реализован с помощью протокола SPX, который построен на базе IPX.

Фирма Novell в сетевой операционной системе NetWare применяла протокол IPX для обмена датаграммами и протокол SPX для обмена в сеансах.

Для некоторых приложений (например, для программ, передающих файлы между рабочими станциями) удобнее использовать сетевой протокол более высокого уровня, обеспечивающий гарантированную доставку пакетов в правильной последовательности. Разумеется, программа может сама следить за тем, чтобы все переданные пакеты были приняты. Однако в этом случае придется делать собственную надстройку над протоколом IPX - собственный протокол передачи данных.

SPX – протокол последовательного обмена пакетами (Sequenced Packet Exchange Protocol), разработанный Novell. Система адресов протокола SPX аналогична системе адресов протокола IPX и также состоит из 3 частей: номера сети, адреса станции и сокета.

Протокол SPX использует такой же блок ECB для передачи и приёма пакетов, что и протокол IPX. Однако, пакет, передаваемый при помощи протокола SPX, имеет более длинный заголовок. Дополнительно к 30 байтам стандартного заголовка пакета IPX добавляется еще 12 байт (рис. 2.1).

- Поле **ConnControl** представляет собой как набор битовых флагов, управляющих передачей данных по каналу SPX.
- Поле **DataStreamType** состоит из однобитовых флагов, которые используются для классификации данных, передаваемых или принимаемых при помощи протокола SPX.

- Поле **SourceConnID** содержит номер канала связи передающей программы, присвоенный драйвером SPX при создании канала связи. Этот номер должен указываться функции передачи пакета средствами SPX.

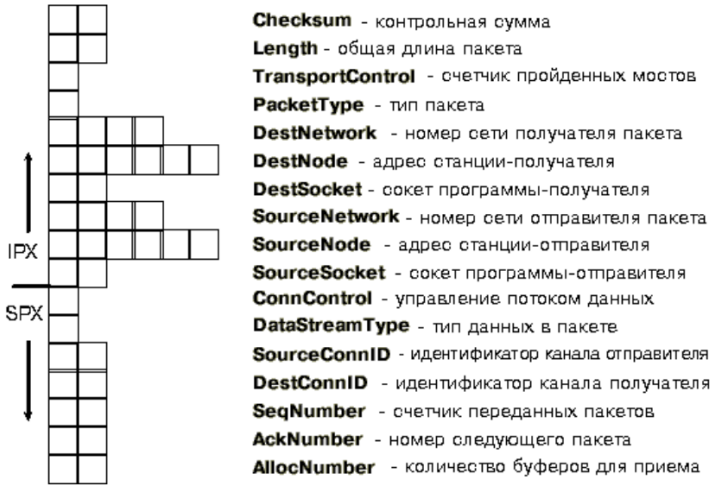


Рис. 2.1. Формат заголовка пакета SPX

- Поле **DestConnID** содержит номер канала связи принимающей стороны. Так как все пакеты приходят на один номер сокета и могут принадлежать разным каналам связи (на одном сокете можно открыть несколько каналов связи), необходимо классифицировать приходящие пакеты по номеру канала связи.
- Поле **SeqNumber** содержит счетчик пакетов, переданных по каналу в одном направлении. На каждой стороне канала используется свой счетчик. После достижения значения FFFFh счетчик сбрасывается в нуль, после чего процесс счета продолжается. Содержимым этого поля управляет драйвер SPX, поэтому программа не должна менять его значение.
- Поле **AckNumber** содержит номер следующего пакета, который должен быть принят драйвером SPX. Содержимым этого поля управляет драйвер SPX, поэтому программа не должна менять его значение.
- Поле **AllocNumber** содержит количество буферов, распределенных программой для приема пакетов. Содержимым этого поля управляет драйвер SPX, поэтому программа не должна менять его значение.

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

Winsock – это интерфейс сетевого программирования для Microsoft Windows. Winsock основывается на сокетной парадигме, изложенной в документах под названием Berkley System Distribution от University of California в Berkley.

Основные операционные среды (Unix подобные системы, Windows) базируются в настоящее время на идеологии соединителей (socket). Эта технология была разработана в университете г. Беркли (США) для системы Unix, поэтому соединители иногда называют соединителями Беркли. Соединители реализуют механизм взаимодействия не только партнеров по телекоммуникациям, но и процессов в ЭВМ вообще.

Winsock включает в себя несколько стилей программирования. Первый – это стандартный однопотоковый стиль с блокированием потока определенными командами,

второй – с использованием оконных процедур и третий – с использованием асинхронных процедур. Стандартная модель программирования от Berkley является de facto для сетей TCP/IP, но под Windows можно использовать эту библиотеку для программирования протоколов IPX/SPX.

Winsock предназначен для использования во всех версиях MS Windows, начиная с 3.0. Для того чтобы программа могла корректно 21 работать с библиотекой Winsock необходимо проверить версию библиотеки Winsock, а так же вообще наличие этой библиотеки в системе. Библиотека функции Winsock расположена в файле wsock32.dll (ws2_32.dll для версии 2.0 этой библиотеки) или winsock.dll для 32-бит и 16-бит приложений соответственно. Также, необходимо подключить заголовочные файлы winsock.h (winsock2.h), а для работы с протоколами IPX и SPX еще и заголовочный файл wsipx.h.

Основные функции API для работы с протоколом IPX

- **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.).
- **WSAGetLastError** (void) возвращает код ошибки, возникшей при выполнении последней операции. После работы с библиотекой, её необходимо выгрузить из памяти.
- **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**. Порядок байт на машинах PC отличается от порядка, используемого в сетях, поэтому необходимы некоторые преобразования определенных данных, например, номера порта, чтобы он был правильным при использовании функций библиотеки Winsock. Ниже приведены функции преобразования порядка байт:
- **u_short htons**(u_short hostshort);
- **u_long htonl**(u_long hostlong);
- **u_long ntohl**(u_long netlong);
- **u_short ntohs**(u_short netshort);

  В качестве параметра передаётся число, которое необходимо преобразовать. Функция возвращает преобразованное число.
- **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. Параметр af содержит сведения о семействе протоколов (AF_INET, AF_IPX). В данной лабораторной работе необходимо использовать константу AF_IPX. Параметр type – тип передаваемых данных (поток или дейтаграммы). В данной

лабораторной работе для IPX необходимо использовать константу SOCK_DGRAM, а для SPX – константу SOCK_SEQPACKET, которая означает, что пакеты будут отсылаться последовательно и в порядке очереди. Параметр protocol – протокол передачи данных. Для протокола IPX используется константа NSPROTO_IPX, для SPX – NSPROTO_SPX.

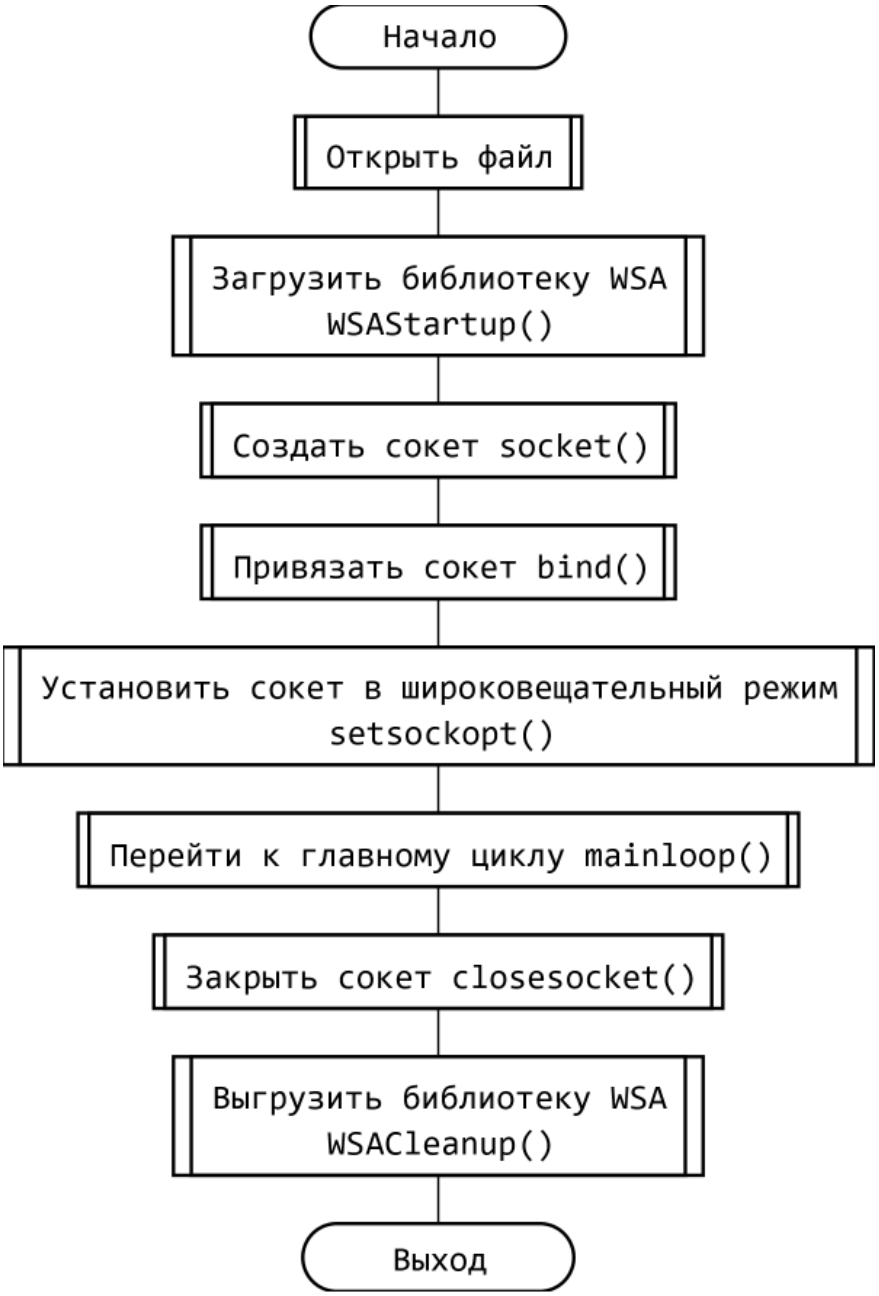
- Чтобы работать дальше с созданным сокетом его нужно привязать к какому-нибудь локальному адресу и порту. Этим занимается функция **bind** (SOCKET s, const struct sockaddr FAR* name, int namelen). Здесь s – дескриптор сокета, который данная функция именует; name – указатель на структуру имени сокета; namelen – размер, в байтах, структуры name. Если порт установить в 0, то система сама пытается подыскать свободный порт. Если в качестве адреса указать константу INADDR_ANY (0) для сетей TCP/IP или 0 в сетях IPX/SPX, то система попытается использовать все доступные адреса для сокета.
- **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. Эта функция используется сервером, чтобы информировать ОС, что он ожидает запросы связи на данном сокете. Без такой функции всякое требование связи с этим сокетом будет отвергнуто.
- **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.
- **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.
- 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,

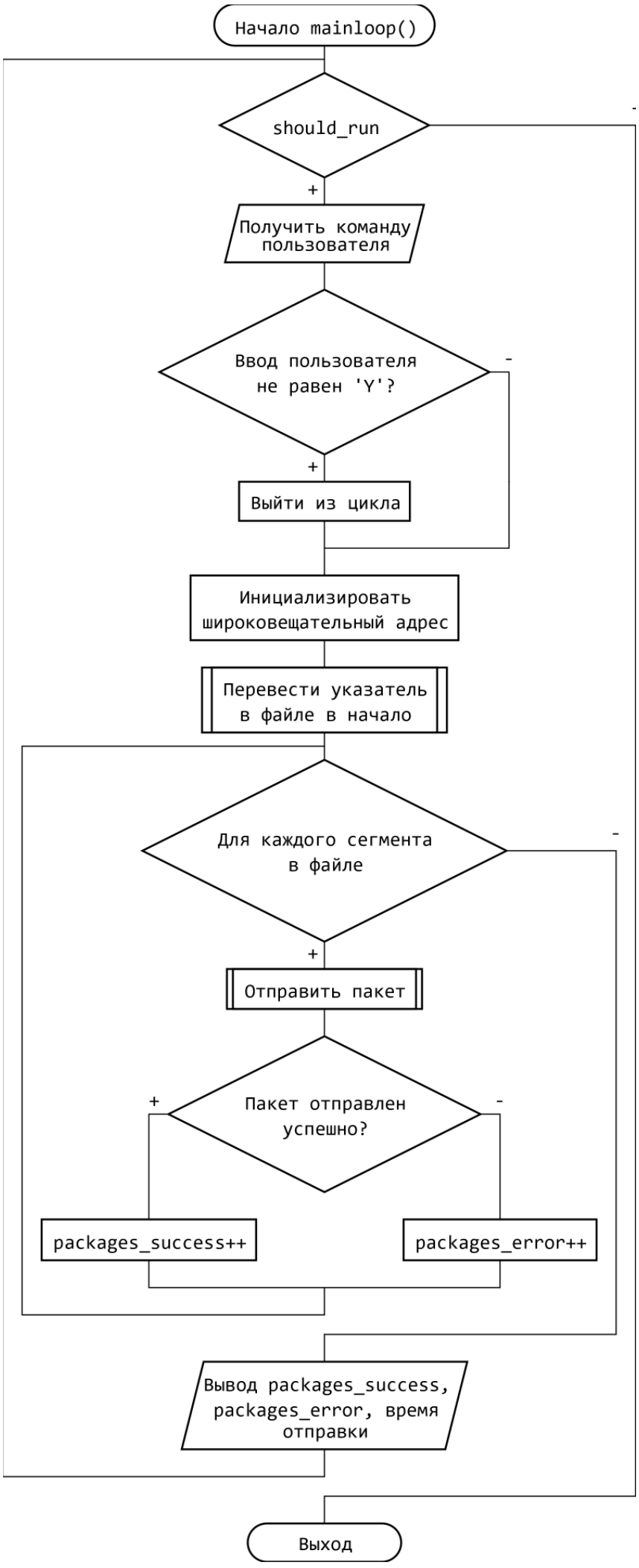
иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

- 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.

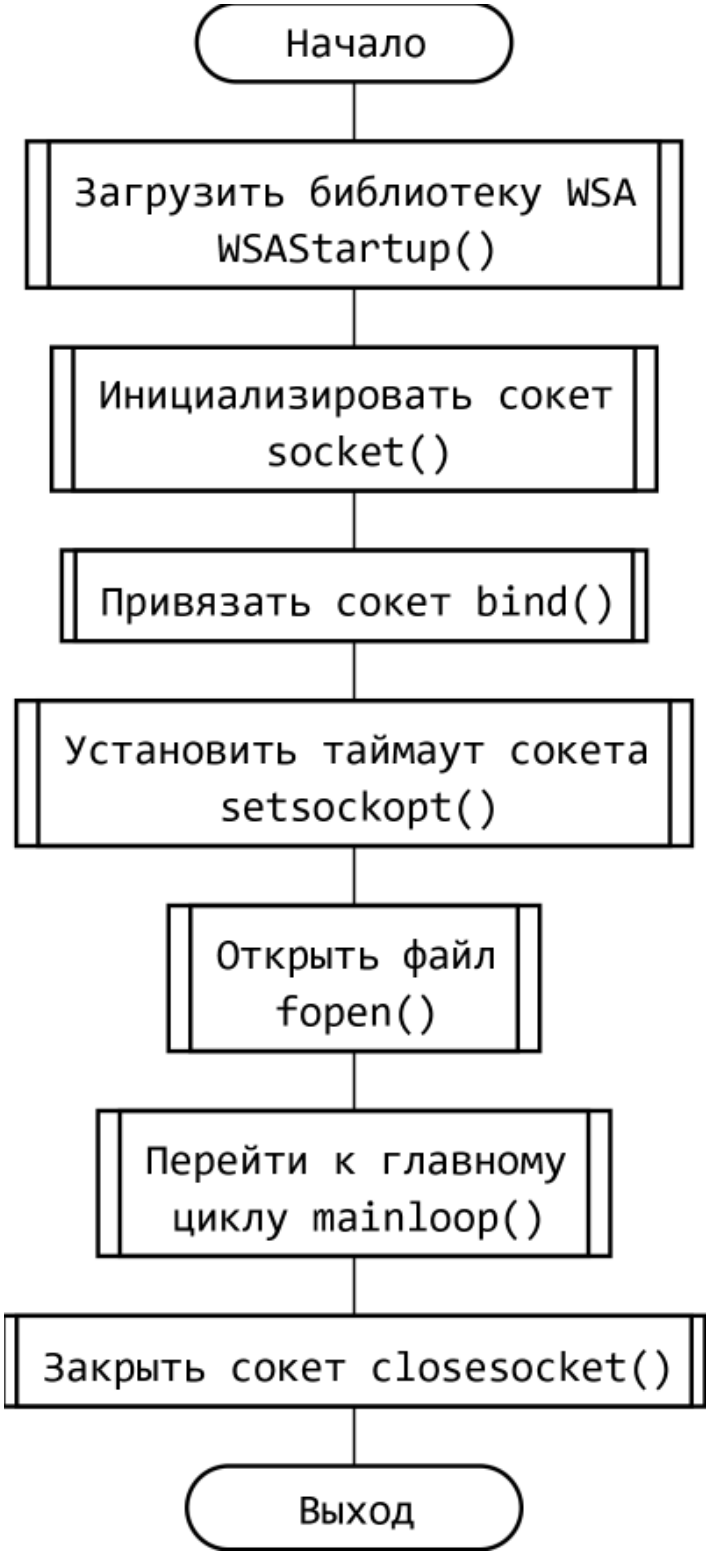
Разработка программы. Блок-схемы программы.

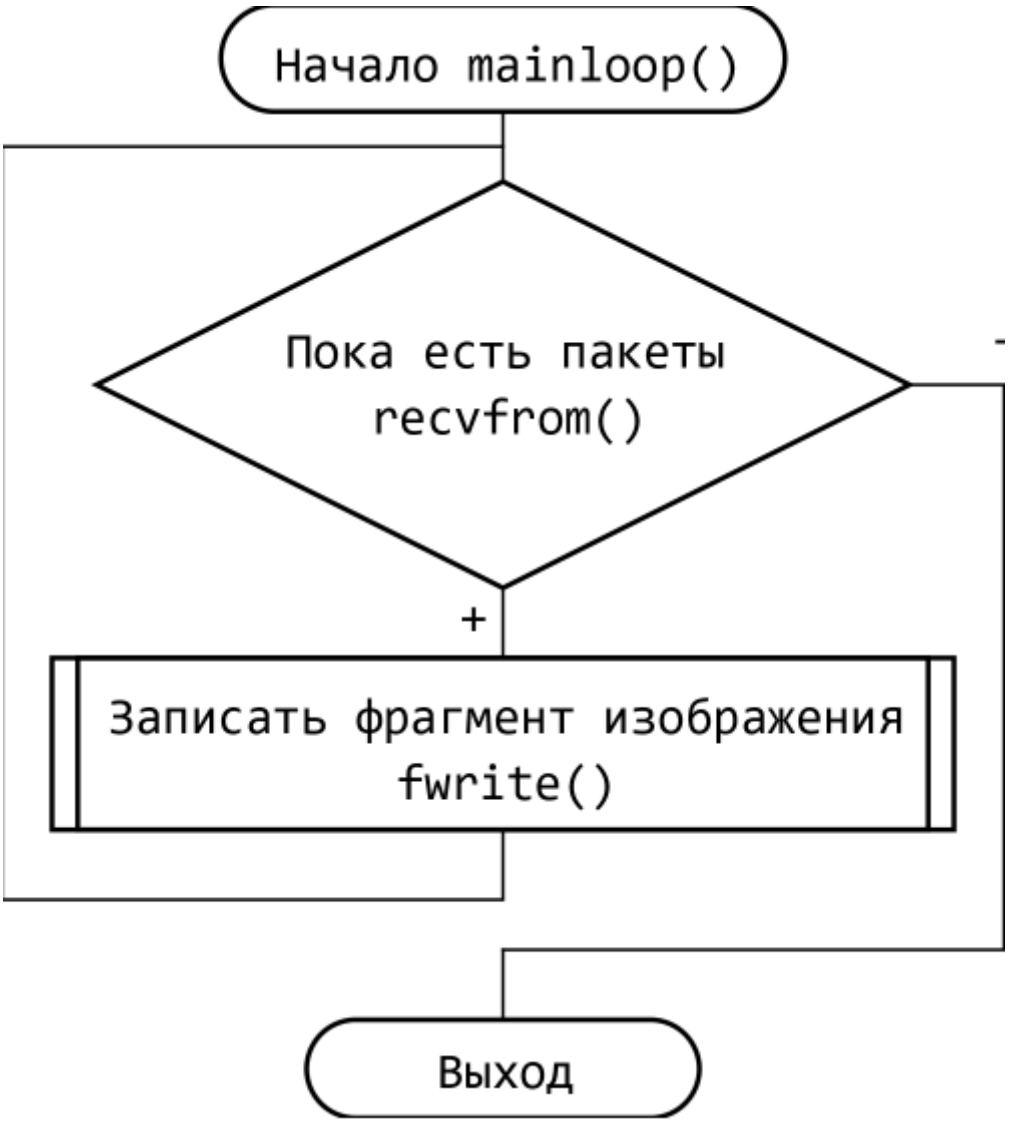
Сервер IPX




Начало mainloop()
should_run
-
+
Получить команду пользователя
Ввод пользователя не равен 'Y'?
-
+
Выйти из цикла
Инициализировать широковещательный адрес
Перевести указатель в файле в начало
Для каждого сегмента в файле
-
+
Отправить пакет
Пакет отправлен успешно?
+
-
packages_success++
packages_error++
Вывод packages_success, packages_error, время отправки
Выход

Клиент IPX

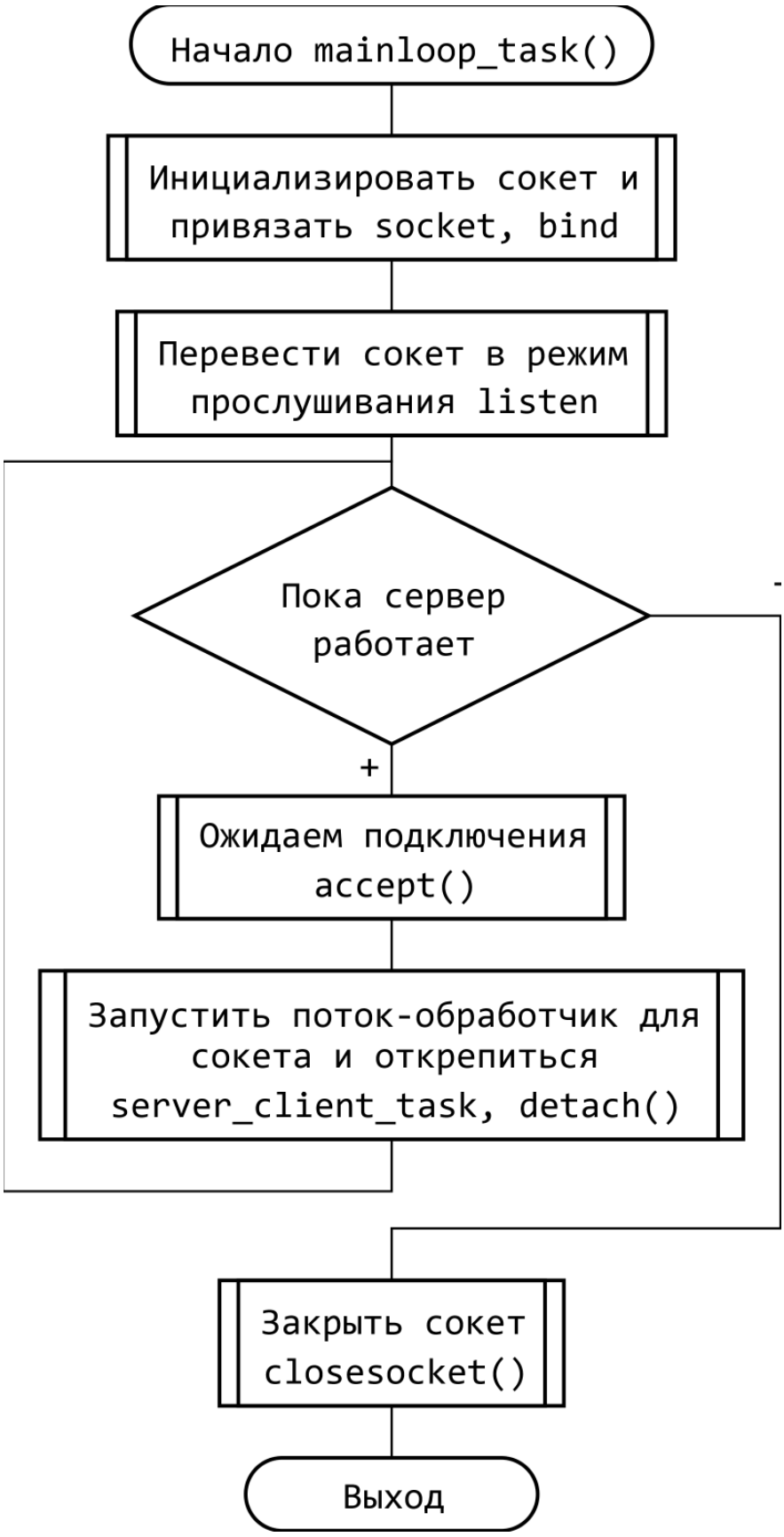

Начало
Загрузить библиотеку WSA
WSAStartup()
Инициализировать сокет
socket()
Привязать сокет bind()
Установить таймаут сокета
setsockopt()
Открыть файл
fopen()
Перейти к главному
циклу mainloop()
Закрыть сокет closesocket()
Выход

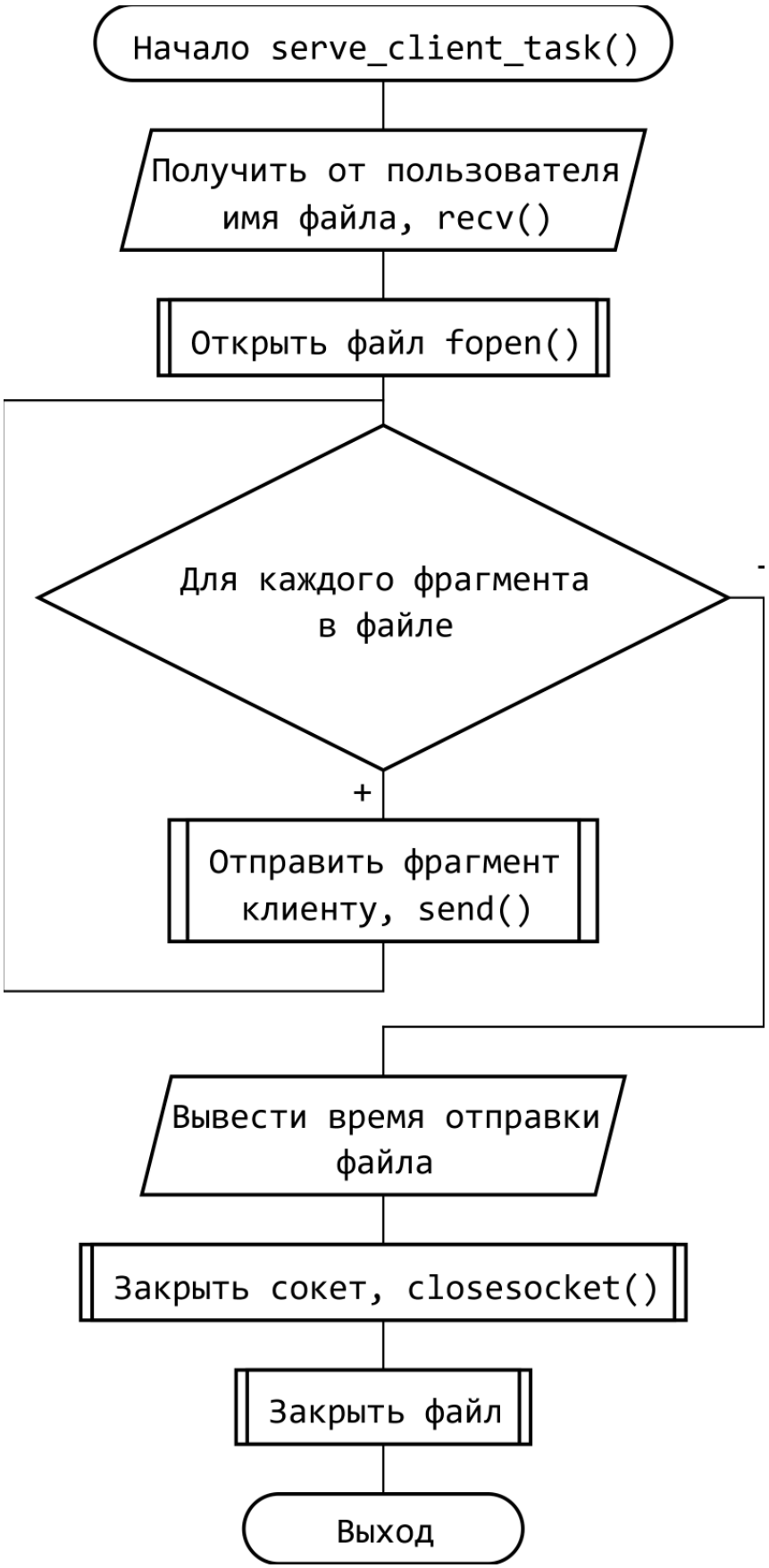

Начало mainloop()
Пока есть пакеты
recvfrom()
+
Записать фрагмент изображения
fwrite()
Выход

Сервер SPX

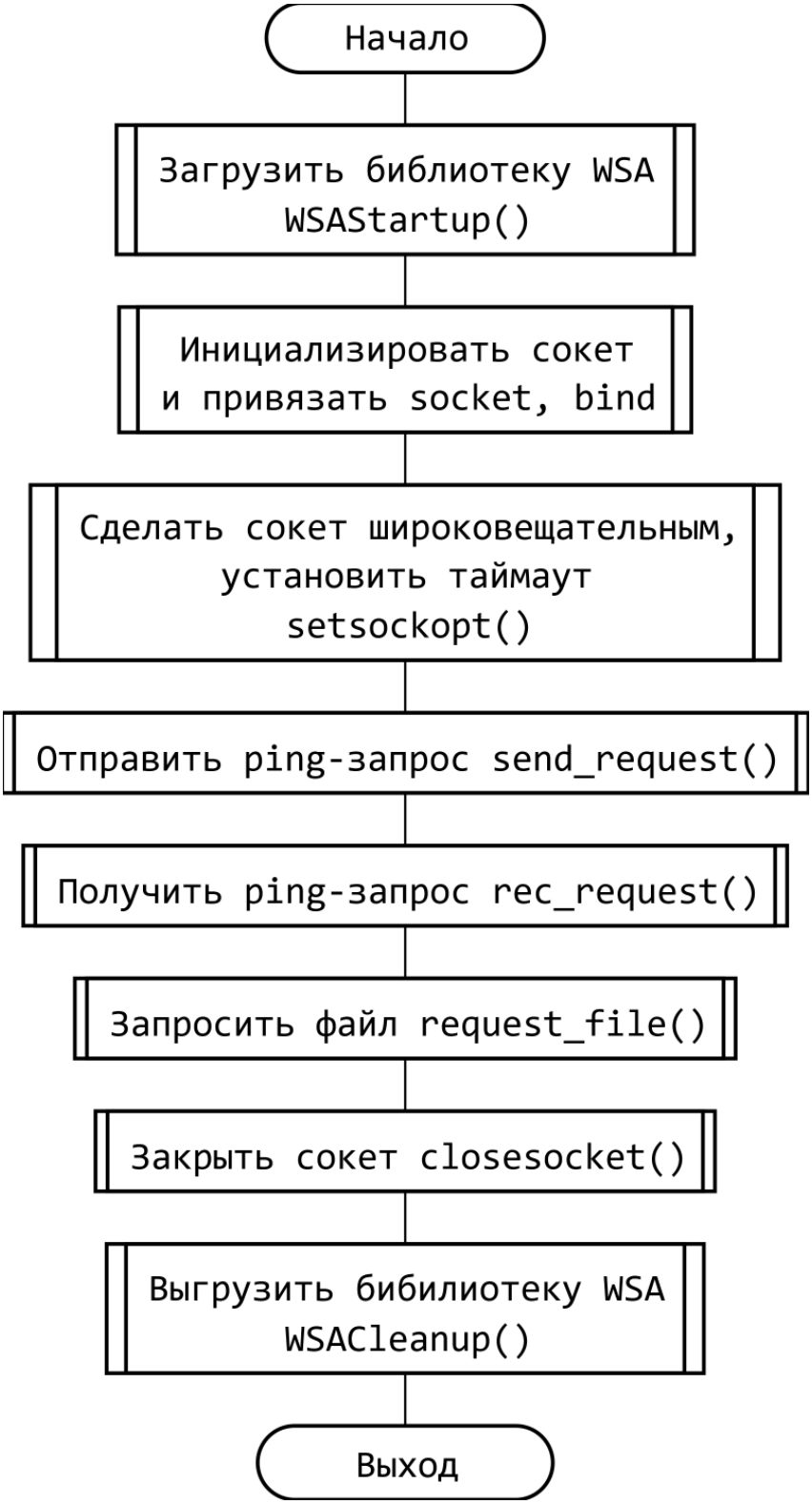

Начало
Загрузить библиотеку WSA
WSAStartup()
Запустить потоки
ping_message_task,
mainloop_task
Ждём ввод
пользователя
Ожидаем завершение
потоков
Выгрузить библиотеку WSA
WSACleanup()
Выход

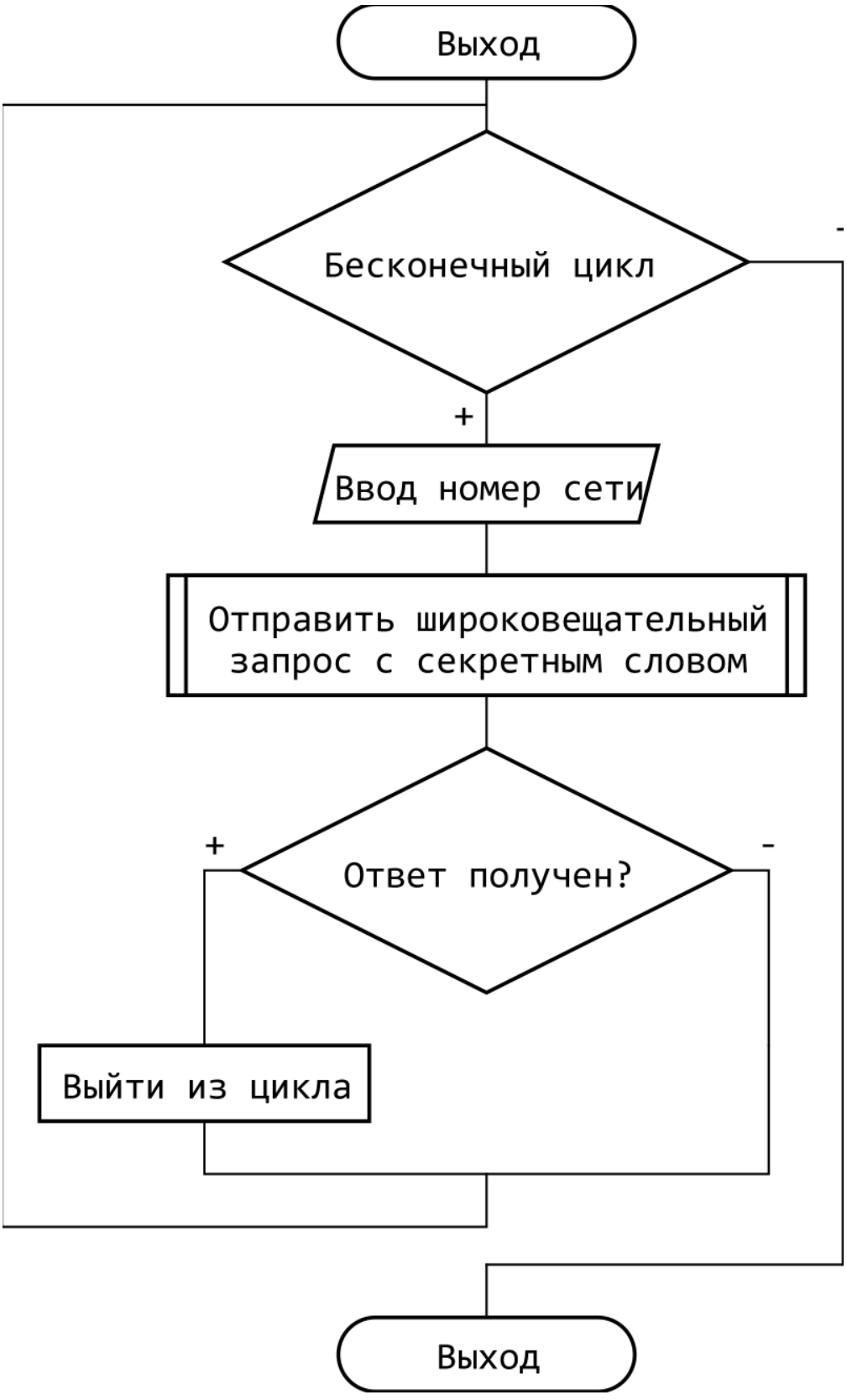

Начало ping_message_task
Создать сокет и привязать его
socket(), bind()
Установить таймаут сокета
setsockopt()
Пока сервер
работает
-
+
Получить пакет
Сообщение от
клиента совпадает
с секретным словом?
-
+
Отправить клиенту
ответный пакет
с секретным ответом
sendto()
Закрыть сокет closesocket()
Выход


Начало mainloop_task()
Инициализировать сокет и привязать socket, bind
Перевести сокет в режим прослушивания listen
Пока сервер работает
-
+
Ожидаем подключения accept()
Запустить поток-обработчик для сокета и открепиться server_client_task, detach()
Закрыть сокет closesocket()
Выход

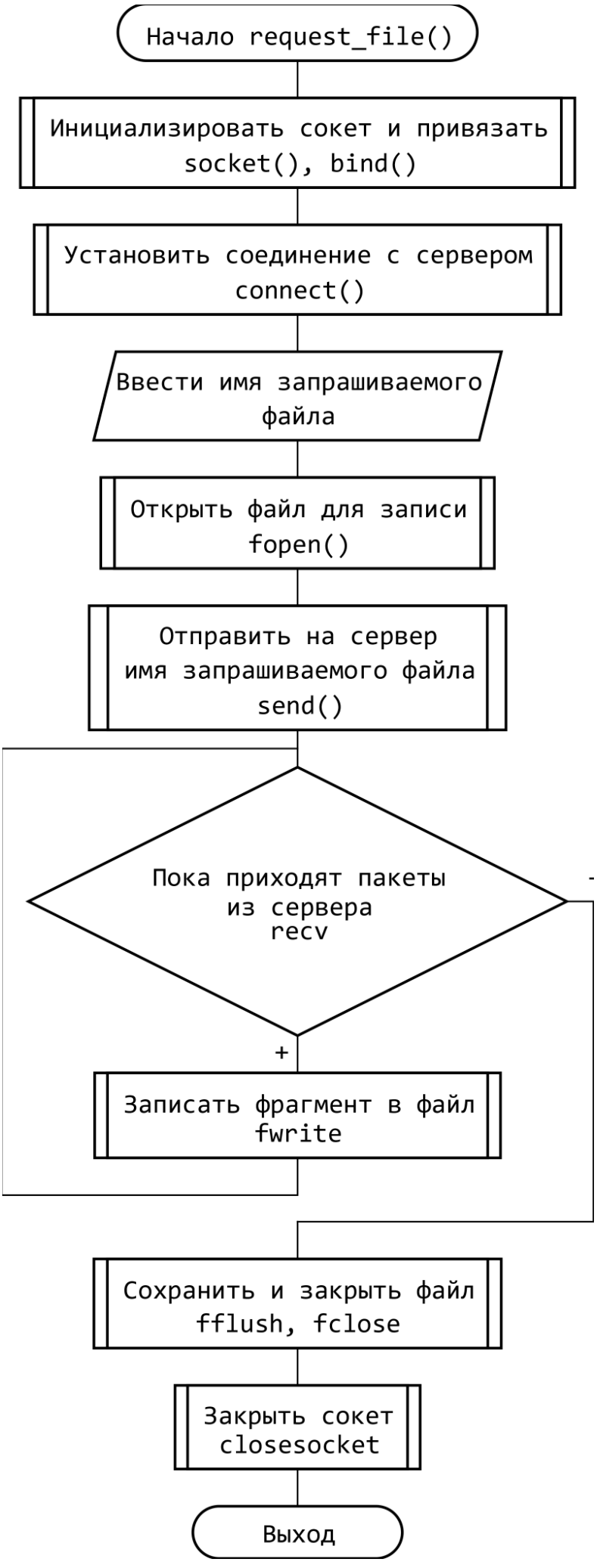

Начало serve_client_task()
Получить от пользователя имя файла, recv()
Открыть файл fopen()
Для каждого фрагмента в файле
-
+
Отправить фрагмент клиенту, send()
Вывести время отправки файла
Закрыть сокет, closesocket()
Закрыть файл
Выход

Клиент SPX


Начало
Загрузить библиотеку WSA
WSAStartup()
Инициализировать сокет
и привязать socket, bind
Сделать сокет широковещательным,
установить таймаут
setsockopt()
Отправить ping-запрос send_request()
Получить ping-запрос rec_request()
Запросить файл request_file()
Закрыть сокет closesocket()
Выгрузить бибилиотеку WSA
WSACleanup()
Выход


Выход
Бесконечный цикл
-
+
Ввод номер сети
Отправить широковещательный запрос с секретным словом
Ответ получен?
+
-
Выйти из цикла
Выход


Начало
Бесконечный цикл
-
+
Получить пакет
Ответ содержит
секретное слово?
-
+
Сохранить адрес ответа,
выйти из цикла
Выход


Начало request_file()
Инициализировать сокет и привязать
socket(), bind()
Установить соединение с сервером
connect()
Ввести имя запрашиваемого
файла
Открыть файл для записи
fopen()
Отправить на сервер
имя запрашиваемого файла
send()
Пока приходят пакеты
из сервера
recv
-
+
Записать фрагмент в файл
fwrite
Сохранить и закрыть файл
fflush, fclose
Закрыть сокет
closesocket
Выход

Анализ функционирования программ

В качестве эксперимента было выбрано изображение размером 31450 Кбайт или 30,7 Мбайт.

Анализ для IPX клиент-серверного взаимодействия:

Было проведено 5 замеров для 3 клиентов и 1 сервера:

	Время, сек.
№1	38,585
№2	38,275
№3	38,495
№4	38,073
№5	38,495
Среднее	38,3846
Дисперсия	0,0434108

Размер файла, Мбайт
30,7

Скорость передачи, Мбит/с
6,398399358

Малая дисперсия говорит о том, что соединение IPX довольно стабильно и равномерно. Также была получена скорость передачи в ~6.4 Мбит/с, по сравнению с результатами в прошлой лабораторной работы скорость выросла в ~4 раза. Передача по широковещательному каналу даёт меньшую нагрузку на компоненты сети, так как роутерам не нужно строить маршруты и просматривать таблицу маршрутизации. Также широковещательная передача не даёт большую нагрузку на сервер при увеличении клиентов, так как сервер поддерживает только один сокет, по которому и ведётся передача данных. Однако в протоколе IPX есть и серьёзные недостатки. В качестве источника передачи было выбрано изображение, которое в процессе передачи было повреждено. Протокол IPX не даёт гарантии доставки пакета, из-за чего и появляются ошибки.

Анализ для SPX клиент-серверного взаимодействия

Было проведено 5 замеров для 3 клиентов и 1 сервера:

	Время, сек.
№1	176,914
№2	198,265
№3	200,318
№4	116,337
№5	117,889
№6	119,201
№7	130,477
№8	182,722
№9	179,808
№10	114,694
№11	114,254
№12	115,516
№13	128,995
№14	181,511
№15	182,166
Среднее	150,604467
Дисперсия	1229,70151

Размер файла, Мбайт
30,7

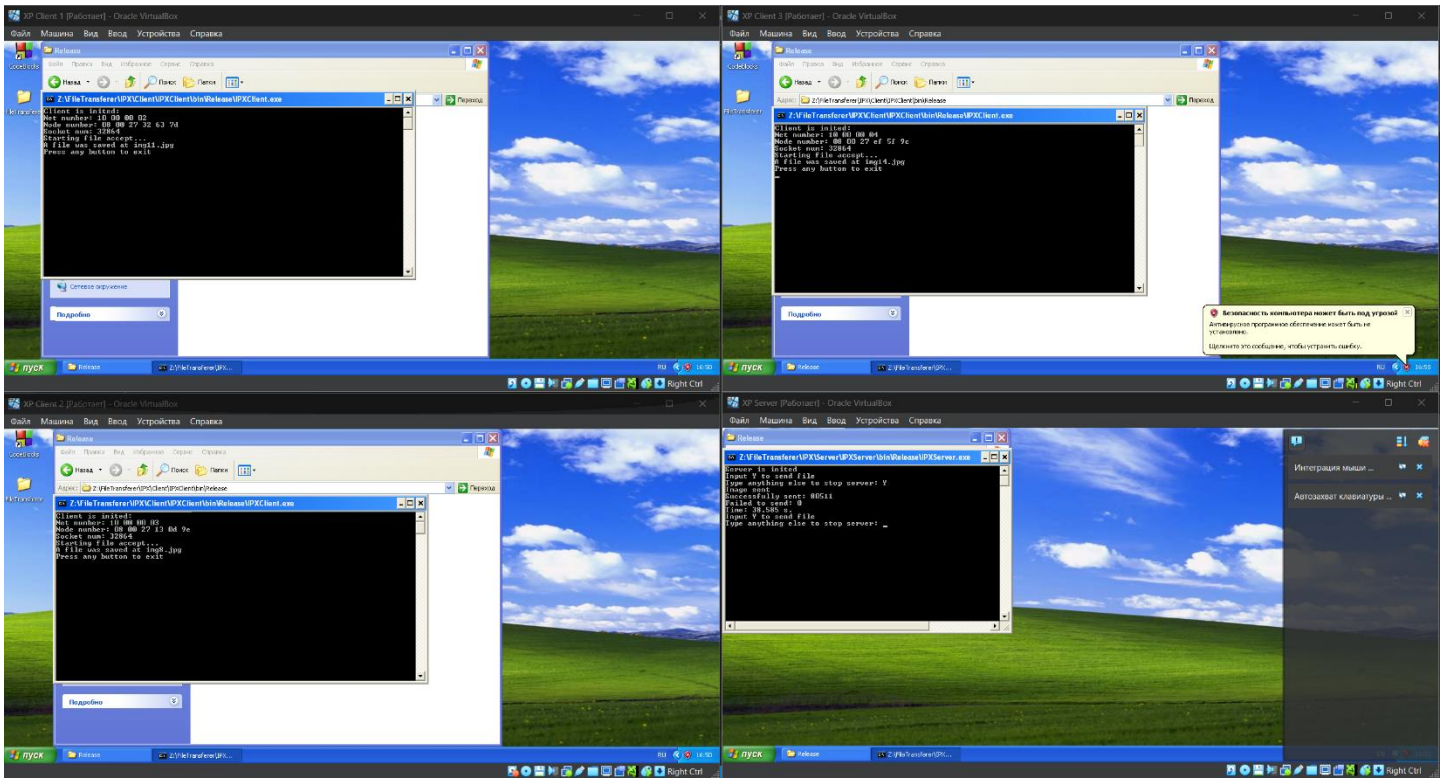
Скорость передачи, Мбит/с
1,630761726

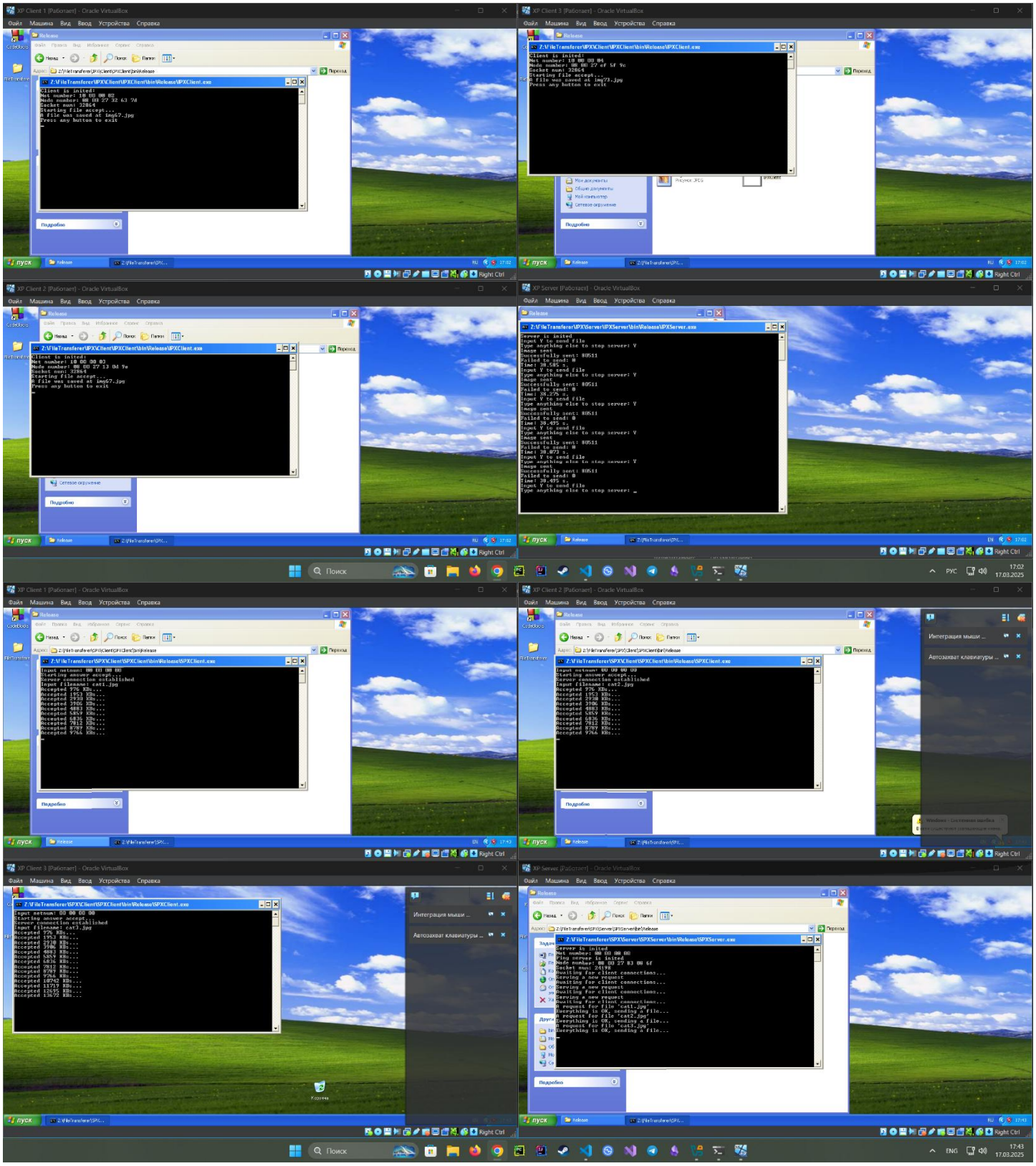
В результате получили гораздо большую дисперсию в сравнении с IPX приложением, что может говорить о нестабильности скорости передачи. Это может быть связано с необходимостью переотправки повреждённых пакетов, а также может говорить о нестабильной конфигурации сети автора отчёта. Среднее время также выросло, а скорость сравнилась со старой реализацией IPX сервера. А также протокол SPX не поддерживает широковещательную передачу. Однако в результате изображение всеми 15 клиентами было получено и не содержит ошибок. А также такой способ соединения гораздо более удобен, так как поставляет установить двусторонний канал общения клиента и сервера.

Вывод: в ходе лабораторной изучили протоколы IPX/SPX, основные функции библиотеки Winsock и разработали программы для приема/передачи пакетов. Протокол SPX менее стабилен в скорости передачи и более затратен по времени, не поддерживает широковещательную отправку, однако позволяет передавать данные без потерь и гарантирует доставку пакетов а также более удобен для установления двустороннего общения от клиента к серверу.

Текст программ. Скриншоты программ.

Ссылка на репозиторий с кодом: https://github.com/IAmProgrammist/comp_net/tree/lab2





```c
#define WIN32_LEAN_AND_MEAN

#include <time.h>
#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <wsipx.h>
#include <stdlib.h>
#include <conio.h>

#define IPX_SOCKET (0x8060)
#define CLIENT_REQUEST_SIZE 512
#define IMAGE_PART_SIZE 546
```

```
SOCKET socket_descriptor;
SOCKADDR_IPX name = {};
SOCKADDR_IPX server_sockaddr = {};

FILE *source;

void mainloop() {
    std::cout << "Starting file accept..." << std::endl;
    char* buffer = (char*)malloc(sizeof(char) * IMAGE_PART_SIZE);
    while (1) {
        int bytes_received;
        if ((bytes_received = recvfrom(
                socket_descriptor,
                buffer,
                sizeof(char) * IMAGE_PART_SIZE,
                0,
                nullptr, nullptr)) != SOCKET_ERROR) {
                   fwrite(buffer, sizeof(char), bytes_received, source);
                } else {
                   break;
                }
    }
}

int main()
{
    char filename[20];

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    socket_descriptor = socket(
                                    AF_IPX,
                                    SOCK_DGRAM,
                                    NSPROTO_IPX
                                    );
    name.sa_family = AF_IPX;
    name.sa_socket = htons(IPX_SOCKET);
    err = bind(socket_descriptor, (sockaddr*)&name, sizeof(name));

    if (err != 0) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    int namelen = sizeof(name);
    getsockname(socket_descriptor, (sockaddr*)(&name), &namelen);

    std::cout << "Client is inited:\n";
    printf("Net number: %02hhx %02hhx %02hhx %02hhx\n", name.sa_netnum[0], name.sa_netnum[1],
name.sa_netnum[2], name.sa_netnum[3]);
    printf("Node number: %02hhx %02hhx %02hhx %02hhx %02hhx %02hhx\n", name.sa_nodenum[0],
name.sa_nodenum[1], name.sa_nodenum[2], name.sa_nodenum[3], name.sa_nodenum[4],
name.sa_nodenum[5]);
    std::cout << "Socket num: " << htons(name.sa_socket) << "\n";
    std::cout.flush();
```

```
    int timeout_time = 10000;

    if (setsockopt(socket_descriptor, SOL_SOCKET, SO_RCVTIMEO, (char*) &timeout_time,
sizeof(timeout_time)) == SOCKET_ERROR) {
        printf("Unable to set timeout: %d\n", WSAGetLastError());
        return 1;
    }

    srand(time(NULL));
    sprintf(filename, "img%d.jpg", rand() % 100 + 1);
    source = fopen(filename, "wb");

    mainloop();

    fflush(source);
    fclose(source);

    err = WSACleanup();

    if (err != 0) {
        printf("WSACleanup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    closesocket(socket_descriptor);

    std::cout << "A file was saved at " << filename << "\nPress any button to exit" << std::endl;
    getchar();
    getchar();

    std::cout << "Have a good day!" << std::endl;

    return 0;
}
```

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <wsipx.h>
#include <stdlib.h>
#include <conio.h>
#include <iomanip>
#include <chrono>
#include <thread>

#define CLIENT_REQUEST_SIZE 512
#define IMAGE_PART_SIZE 400
#define IPX_SOCKET (0x8060)

SOCKET socket_descriptor;
SOCKADDR_IPX name = {};

FILE *source;

void mainloop() {
    bool should_run = true;
    int bytes_read;

    while(should_run) {
        std::string input;
        std::cout << "Input Y to send file\nType anything else to stop server: ";
        std::cout.flush();
```

```
        std::cin >> input;

        if (input != "Y") {
            break;
        }

        SOCKADDR_IPX client_sockaddr = {};
        client_sockaddr.sa_family = AF_IPX;
        memset(client_sockaddr.sa_netnum, 0, 4);
        memset(client_sockaddr.sa_nodenum, 0xFF, 6);
        client_sockaddr.sa_socket = htons(IPX_SOCKET);

        int client_sockaddr_size = sizeof(client_sockaddr);

        int packages_success = 0, packages_error = 0;
        fseek(source, 0, SEEK_SET);

        auto a = std::chrono::high_resolution_clock::now();
        char image_buffer[IMAGE_PART_SIZE];

        while ((bytes_read = fread(image_buffer, sizeof(char), IMAGE_PART_SIZE, source))) {
            if (sendto(socket_descriptor,
                    image_buffer,
                    bytes_read,
                    0,
                    (sockaddr*)&client_sockaddr,
                    client_sockaddr_size) == SOCKET_ERROR) {
                packages_error++;
            } else {
                packages_success++;
            }
        }
        auto b = std::chrono::high_resolution_clock::now();

        std::cout <<
        "Image sent\nSuccessfully sent: " << packages_success <<
        "\nFailed to send: " << packages_error <<
        "\nTime: " << std::chrono::duration_cast<std::chrono::milliseconds>(b - a).count() /
1000.0 << " s." << std::endl;
    }
}

int main()
{
    source = fopen("cat.jpg", "rb");
    if (!source) {
        std::cout << "Cat image not found :(" << std::endl;
        return 1;
    }

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    socket_descriptor = socket(
                                    AF_IPX,
                                    SOCK_DGRAM,
                                    NSPROTO_IPX
                                    );
```

```
    name.sa_family = AF_IPX;
    err = bind(socket_descriptor, (sockaddr*)&name, sizeof(name));

    if (err != 0) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    int namelen = sizeof(name);
    getsockname(socket_descriptor, (sockaddr*)(&name), &namelen);
    std::cout << "Server is inited" << std::endl;

    bool broadcast = true;
    if (setsockopt(socket_descriptor, SOL_SOCKET, SO_BROADCAST, (char*)&broadcast,
sizeof(broadcast)) == SOCKET_ERROR) {
        printf("Unable to set broadcast\n");
        closesocket(socket_descriptor);
        WSACleanup();
    }

    mainloop();

    err = WSACleanup();

    if (err != 0) {
        printf("WSACleanup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    closesocket(socket_descriptor);

    std::cout << "Have a good day!" << std::endl;

    return 0;
}
```

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <wsipx.h>
#include <stdlib.h>
#include <conio.h>
#include <iomanip>
#include <chrono>
#include <thread>

#define CLIENT_REQUEST_SIZE 512
#define IMAGE_PART_SIZE 400
#define BACKLOG_MAX_SIZE

#define IPX_SOCKET (0x8060)
#define SPX_SOCKET (0x5E86)

SOCKET socket_descriptor;
SOCKADDR_IPX name = {};
SOCKADDR_IPX server_sockaddr = {};

SOCKADDR_IPX server_address;

void send_request() {
    char data[CLIENT_REQUEST_SIZE] = "What is the Music of Life?";

    while (true) {
        std::cout << "Input netnum: ";
```

```
        std::cout.flush();
        scanf("%02hhx %02hhx %02hhx %02hhx", server_sockaddr.sa_netnum, server_sockaddr.sa_netnum
+ 1, server_sockaddr.sa_netnum + 2, server_sockaddr.sa_netnum + 3);

        memset(server_sockaddr.sa_nodenum, 0xFF, 6);
        server_sockaddr.sa_socket = htons(IPX_SOCKET);
        server_sockaddr.sa_family = AF_IPX;

        if (sendto(socket_descriptor,
                   data,
                   CLIENT_REQUEST_SIZE * sizeof(char),
                   0,
                   (sockaddr*)&server_sockaddr,
                   sizeof(server_sockaddr)) == SOCKET_ERROR) {
            printf("Unable to connect to server: %d\nTry again\n", WSAGetLastError());
        } else {
            return;
        }
    }
}

void rec_request() {
    std::cout << "Starting answer accept..." << std::endl;
    char* buffer = (char*)malloc(sizeof(char) * IMAGE_PART_SIZE);
    while (1) {
        SOCKADDR_IPX receive_name = {};
        receive_name.sa_family = AF_IPX;
        int receive_name_size = sizeof(receive_name);

        int bytes_received;
        if (bytes_received = recvfrom(
                 socket_descriptor,
                 buffer,
                 sizeof(char) * IMAGE_PART_SIZE,
                 0,
                 (SOCKADDR*) &receive_name,
                 &receive_name_size) != SOCKET_ERROR) {
                     if (strcmp(buffer, "Silence, my Brother.") == 0) {
                        server_address = receive_name;
                        server_address.sa_socket = htons(SPX_SOCKET);
                        server_address.sa_family = AF_IPX;
                        break;
                     }
        } else {
            break;
        }
    }
}

void request_file() {
    SOCKADDR_IPX rf_name = {};

    SOCKET socket_descriptor = socket(
                                    AF_IPX,
                                    SOCK_SEQPACKET,
                                    NSPROTO_SPX
                                    );
    rf_name.sa_family = AF_IPX;
    if (bind(socket_descriptor, (sockaddr*)&rf_name, sizeof(rf_name)) == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        return;
    }

    if (connect(socket_descriptor, (sockaddr*)&server_address, sizeof(server_address)) ==
SOCKET_ERROR) {
        printf("Failed to connect to server %d\n", WSAGetLastError());
        closesocket(socket_descriptor);
```

```
        return;
    }

    std::cout << "Server connection established\n";

    std::string filename;
    std::cout << "Input filename: ";
    std::cout.flush();
    std::cin >> filename;

    FILE* save_file = fopen(filename.c_str(), "wb");

    if (!save_file) {
        std::cout << "Unable to open file for saving" << std::endl;
        closesocket(socket_descriptor);
        return;
    }

    if (send(socket_descriptor, filename.c_str(), filename.size() + 1, 0) == SOCKET_ERROR) {
        printf("Failed to send request for file %d\n", WSAGetLastError());
        closesocket(socket_descriptor);
        return;
    }

    int diff = 0;
    int total_diff = 0;
    int bytes_read;
    char image_buffer[IMAGE_PART_SIZE];
    while ((bytes_read = recv(socket_descriptor, image_buffer, sizeof(char) * IMAGE_PART_SIZE,
0)) > 0) {
        diff += bytes_read;
        total_diff += bytes_read;

        while (diff > 1000000) {
            std::cout << "Accepted " << total_diff / 1024 << " KBs..." << std::endl;
            diff -= 1000000;
        }

        fwrite(image_buffer, sizeof(char), bytes_read, save_file);
    }

    std::cout << "File accepted succesfully! Saving result into '" << filename << "'" <<
std::endl;

    fflush(save_file);
    fclose(save_file);

    closesocket(socket_descriptor);
}

int main()
{
    char filename[20];

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD(2, 0);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    socket_descriptor = socket(
```

```
                                    AF_IPX,
                                    SOCK_DGRAM,
                                    NSPROTO_IPX
                                    );
    name.sa_family = AF_IPX;
    err = bind(socket_descriptor, (sockaddr*)&name, sizeof(name));

    if (err != 0) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    int timeout_time = 4000;
    bool allow_broadcast = true;

    setsockopt(socket_descriptor, SOL_SOCKET, SO_RCVTIMEO, (char*) &timeout_time,
sizeof(timeout_time));
    setsockopt(socket_descriptor, SOL_SOCKET, SO_BROADCAST, (char*) &allow_broadcast,
sizeof(allow_broadcast));

    send_request();
    rec_request();
    request_file();

    err = WSACleanup();

    if (err != 0) {
        printf("WSACleanup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    closesocket(socket_descriptor);

    std::cout << "Have a good day!" << std::endl;

    return 0;
}
```

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <iostream>
#include <winsock2.h>
#include <wsipx.h>
#include <stdlib.h>
#include <conio.h>
#include <iomanip>
#include <chrono>
#include <thread>

#define CLIENT_REQUEST_SIZE 512
#define IMAGE_PART_SIZE 400
#define BACKLOG_MAX_SIZE 10
#define IPX_SOCKET (0x8060)
#define SPX_SOCKET (0x5E86)

bool should_run = true;
std::thread *ping_thread     = nullptr;
std::thread *mainloop_thread = nullptr;

void ping_message_task() {
    SOCKET socket_descriptor;
    SOCKADDR_IPX name = {};
    socket_descriptor = socket(
                                    AF_IPX,
                                    SOCK_DGRAM,
```

```
                                    NSPROTO_IPX
                                    );
    name.sa_family = AF_IPX;
    name.sa_socket = htons(IPX_SOCKET);
    int err = bind(socket_descriptor, (sockaddr*)&name, sizeof(name));

    if (err != 0) {
        printf("bind failed with error: %d\n", WSAGetLastError());
    }

    int namelen = sizeof(name);
    getsockname(socket_descriptor, (sockaddr*)(&name), &namelen);
    std::cout << "Ping server is inited\n";
    int timeout_time = 1000;
    setsockopt(
                socket_descriptor,
                SOL_SOCKET,
                SO_RCVTIMEO,
                (char*) &timeout_time,
                sizeof(timeout_time)
               );

    char* accept_client_data = (char*) malloc(sizeof(char) * CLIENT_REQUEST_SIZE);
    char answer[] = "Silence, my Brother.";

    while(should_run) {
        SOCKADDR_IPX client_sockaddr = {};
        client_sockaddr.sa_family = AF_IPX;
        int client_sockaddr_size = sizeof(client_sockaddr);

        if (recvfrom(
                 socket_descriptor,
                 accept_client_data,
                 sizeof(char) * CLIENT_REQUEST_SIZE,
                 0,
                 (sockaddr*)&client_sockaddr,
                 &client_sockaddr_size
                 ) == SOCKET_ERROR) {
        } else {
            if (strcmp(accept_client_data, "What is the Music of Life?") == 0) {
                std::this_thread::sleep_for(std::chrono::milliseconds(100));
                sendto(socket_descriptor,
                       answer,
                       sizeof(answer),
                       0,
                       (sockaddr*)&client_sockaddr,
                       client_sockaddr_size);
            }
        }
    }

    closesocket(socket_descriptor);
    free(accept_client_data);
}

void serve_client_task(SOCKET client_socket) {
    printf("Serving a new request\n");
    char client_buffer[CLIENT_REQUEST_SIZE] = {};
    if (recv(client_socket, client_buffer, sizeof(client_buffer), 0) == SOCKET_ERROR) {
        printf("Failed to get message for client: %d\n", WSAGetLastError());
        closesocket(client_socket);
        return;
    }

    printf("A request for file '%s'\n", client_buffer);

    FILE* source = fopen(client_buffer, "rb");
```

```
    if (!source) {
        std::cout << "File not found" << std::endl;
        closesocket(client_socket);
        return;
    }

    std::cout << "Everything is OK, sending a file..." << std::endl;

    auto a = std::chrono::high_resolution_clock::now();
    int bytes_read = 0;
    char image_buffer[IMAGE_PART_SIZE];
    while ((bytes_read = fread(image_buffer, sizeof(char), IMAGE_PART_SIZE, source))) {
        if (send(client_socket,
               image_buffer,
               bytes_read,
               0) == SOCKET_ERROR) {
                printf("Failed to send package: %d\n", WSAGetLastError());
                closesocket(client_socket);
                fclose(source);
                return;
        }
    }

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

    std::cout << "File sent successfully\nTime:" <<
    std::chrono::duration_cast<std::chrono::milliseconds>(b - a).count() / 1000.0 << " s." <<
std::endl;

    closesocket(client_socket);
    fclose(source);
}

void mainloop_task() {
    SOCKET socket_descriptor;
    SOCKADDR_IPX name = {};
    socket_descriptor = socket(
                                    AF_IPX,
                                    SOCK_SEQPACKET,
                                    NSPROTO_SPX
                                    );
    name.sa_family = AF_IPX;
    name.sa_socket = htons(SPX_SOCKET);

    if (bind(socket_descriptor, (sockaddr*)&name, sizeof(name)) == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
    }

    int namelen = sizeof(name);
    getsockname(socket_descriptor, (sockaddr*)(&name), &namelen);
    std::cout << "Server is inited\n";
    printf("Net number: %02hhx %02hhx %02hhx %02hhx\n", name.sa_netnum[0], name.sa_netnum[1],
name.sa_netnum[2], name.sa_netnum[3]);
    printf("Node number: %02hhx %02hhx %02hhx %02hhx %02hhx %02hhx\n", name.sa_nodenum[0],
name.sa_nodenum[1], name.sa_nodenum[2], name.sa_nodenum[3], name.sa_nodenum[4],
name.sa_nodenum[5]);
    std::cout << "Socket num: " << htons(name.sa_socket) << "\n";
    std::cout.flush();

    if (listen(socket_descriptor, BACKLOG_MAX_SIZE) == SOCKET_ERROR) {
        printf("Couldn't startup server (listen failed): %d\n", WSAGetLastError());
        return;
    }

    while (should_run) {
        std::cout << "Awaiting for client connections..." << std::endl;
        SOCKADDR_IPX clientAddr;
```

```
        int clientAddrSize = sizeof(clientAddr);

        SOCKET client_descriptor = accept(
                                            socket_descriptor,
                                            (sockaddr*) &clientAddr,
                                            &clientAddrSize);

        if (client_descriptor != INVALID_SOCKET) {
            std::thread serve_client_thread(serve_client_task, client_descriptor);
            serve_client_thread.detach();
        }
    }

    closesocket(socket_descriptor);
}

int main()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD(2, 2);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", WSAGetLastError());
        return 1;
    }

    ping_thread = new std::thread(ping_message_task);
    mainloop_thread = new std::thread(mainloop_task);

    getchar();

    should_run = false;
    std::cout << "Have a good day! Quitting threads..." << std::endl;
    ping_thread->join();
    mainloop_thread->join();
    delete ping_thread;
    delete mainloop_thread;

    WSACleanup();
}
```