**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 SPX»

Выполнил: ст. группы ПВ-223

Игнатьев Артур Олегович

Проверили:

Рубцов Константин Анатольевич

Белгород 2025 г.

**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.

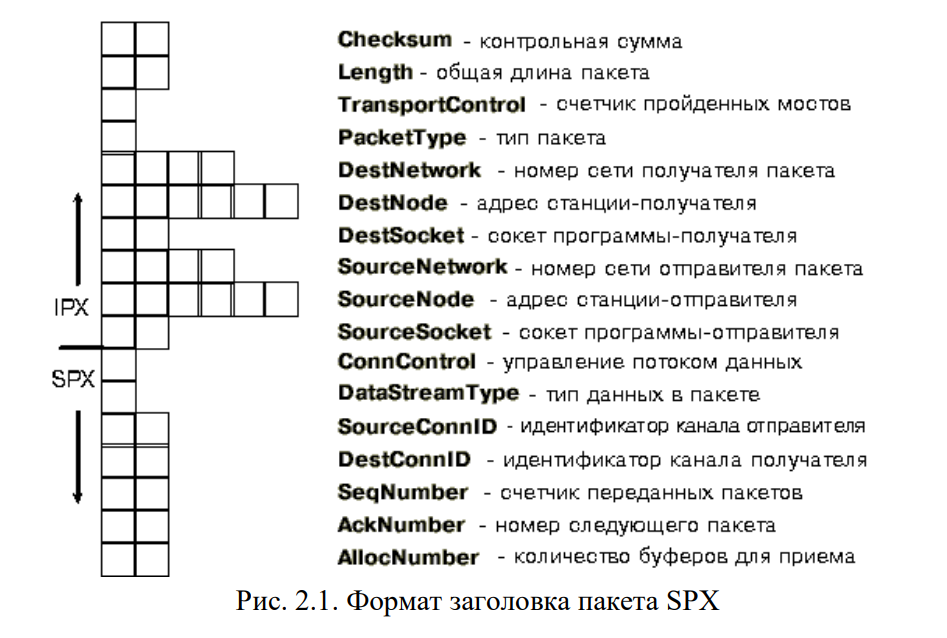
Фирма Novell в сетевой операционной системе NetWare применяла протокол IPX для обмена датаграммами и протокол SPX для обмена в сеансах.

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.



* 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.

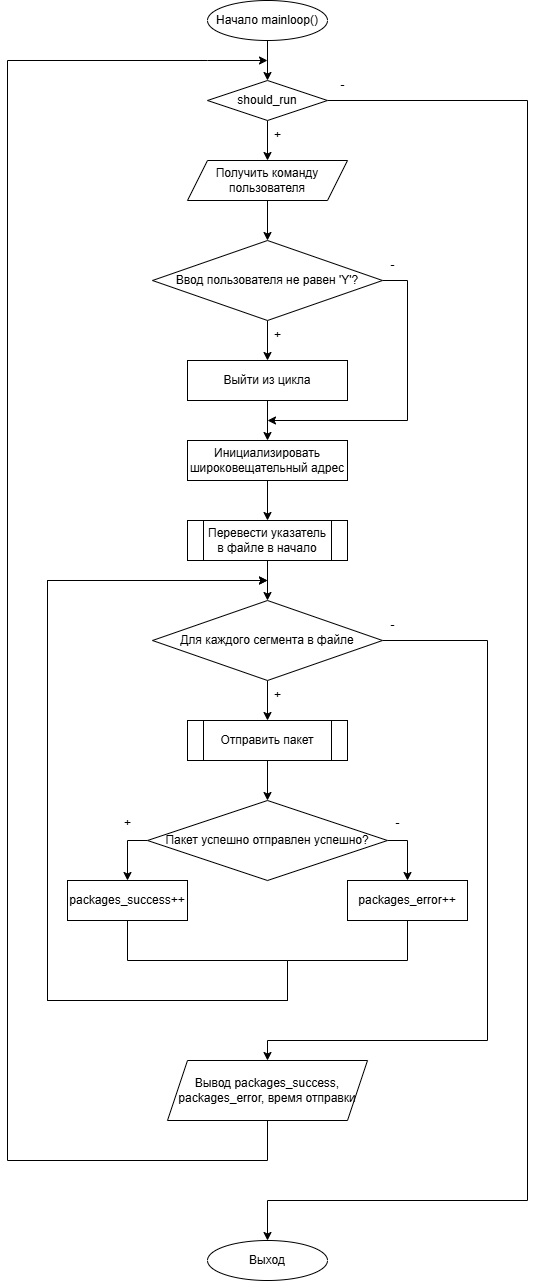
* **socket** (int af, int type, int protocol) возвращает либо дескриптор созданного сокета, либо ошибку INVALID\_SOCKET. Расширенный код ошибки можно получить при помощи функции WSAGetLastError.   
  Параметр af содержит сведения о семействе протоколов (AF\_INET, AF\_IPX). В данной лабораторной работе 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.

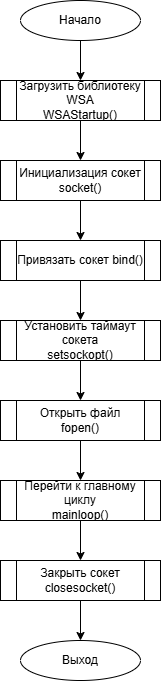
**Разработка программы. Блок-схемы программы**

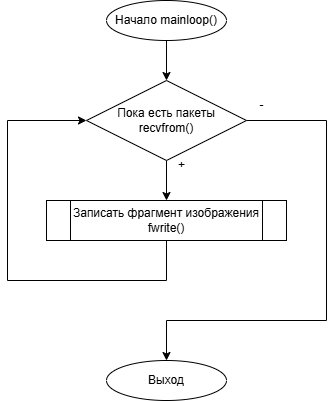
**Сервер IPX**





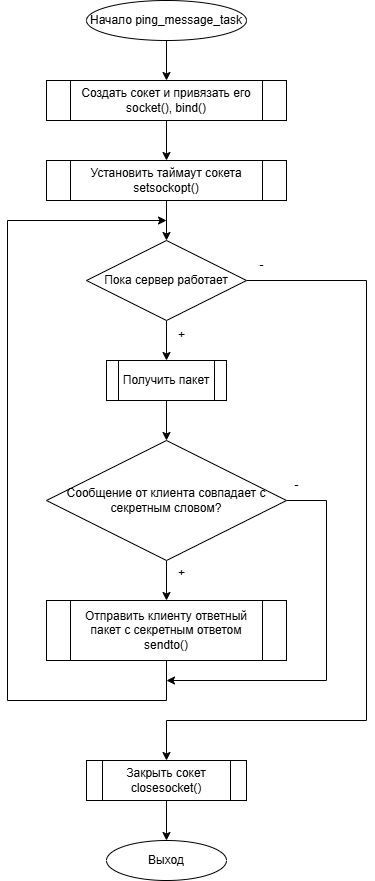
**Клиент IPX**

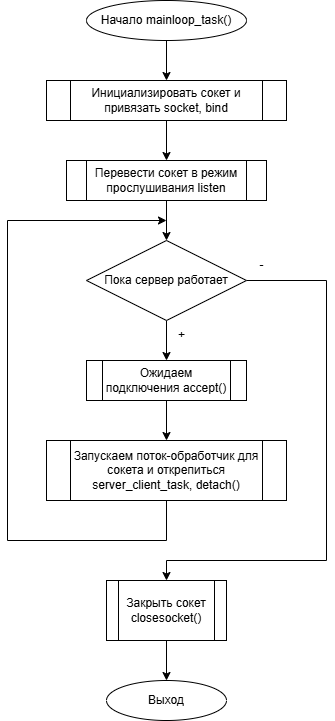


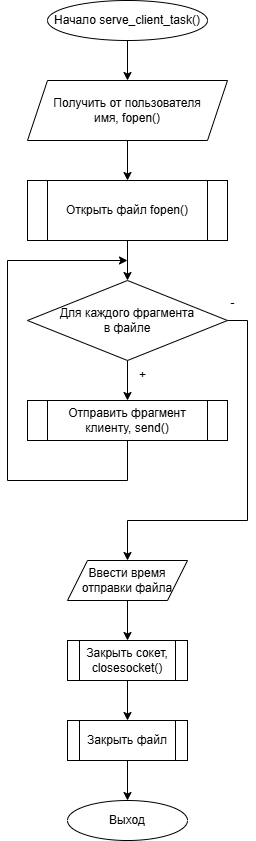


**Сервер SPX**

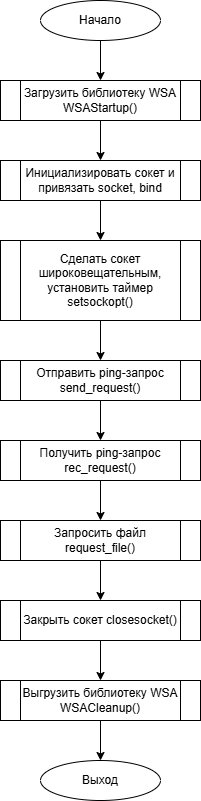


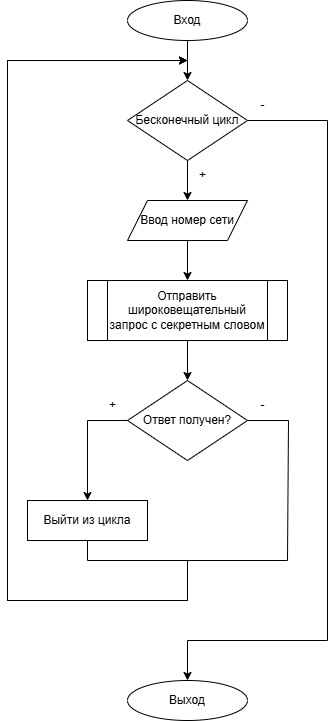




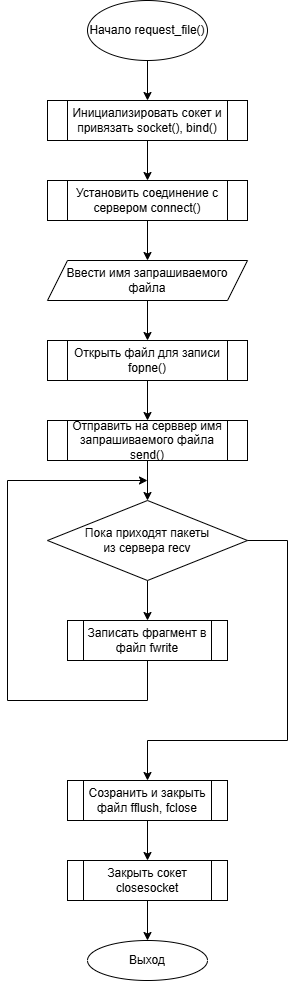


**Клиент SPX**









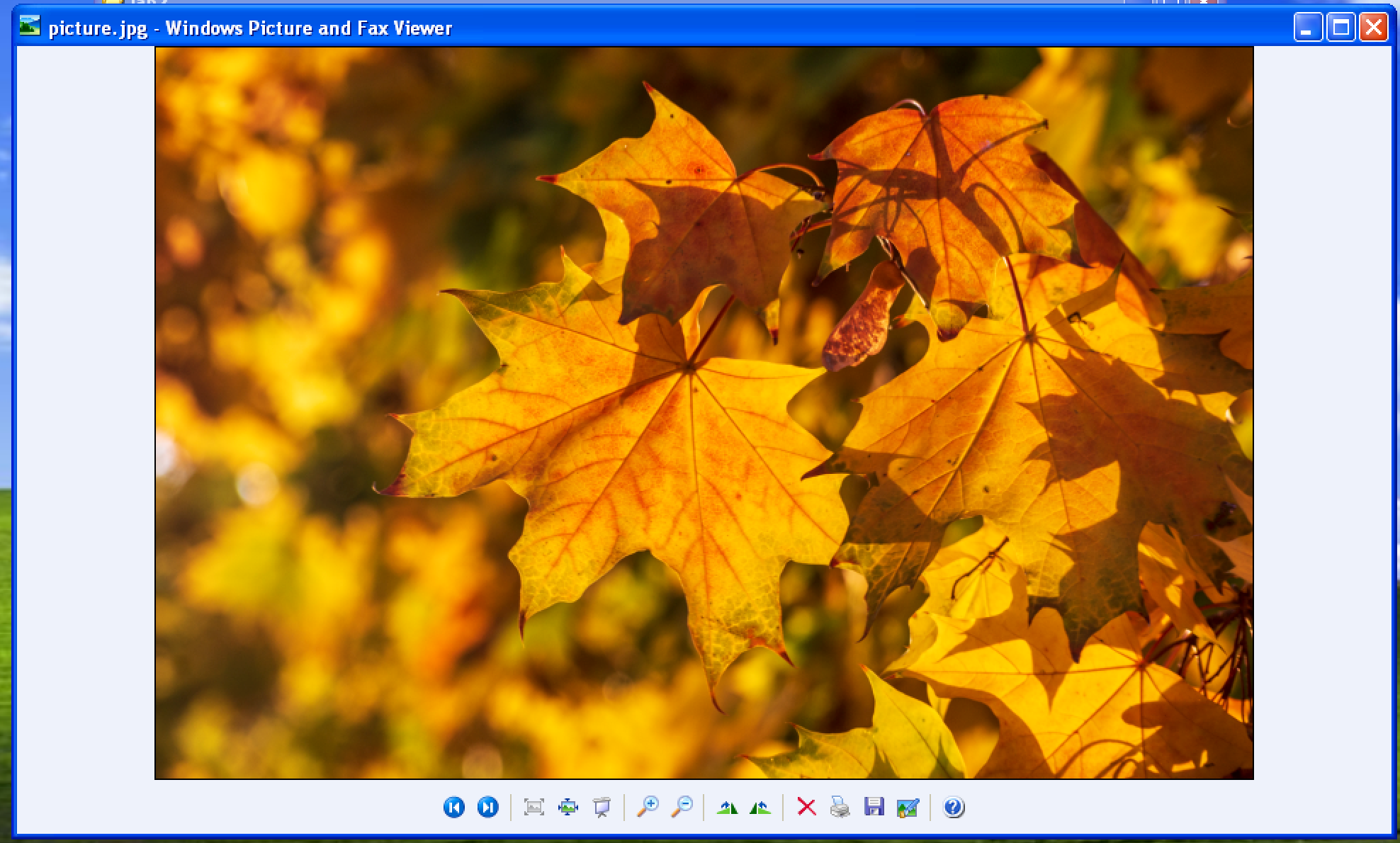
**Анализ функционирования программы**

Изображение было выбрано размером 5.37Mb.

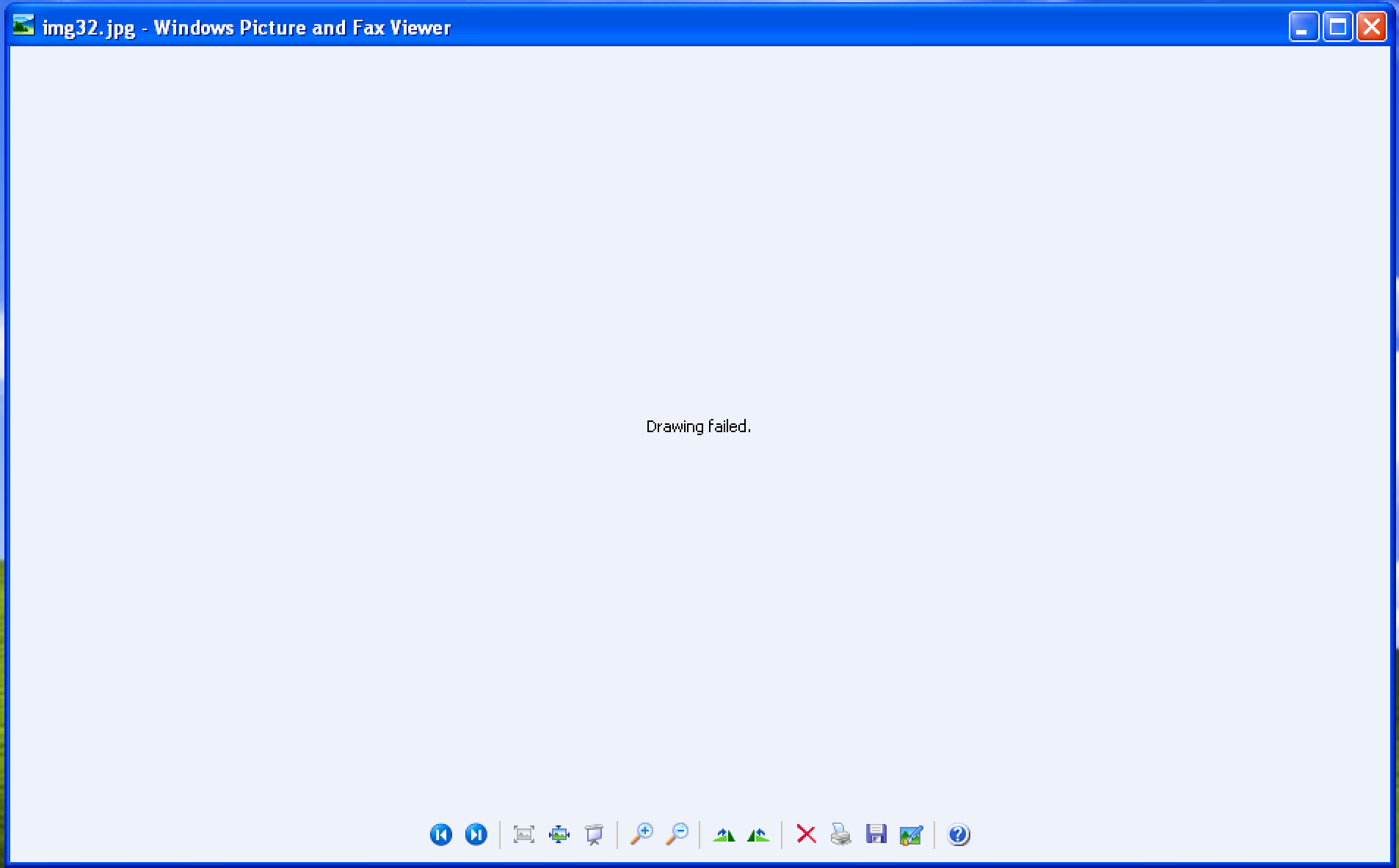
**Анализ IPX протокола**

Пример переданного изображения

Исходное изображение



Переданное изображение



Передача осуществлялась 3м клиентам

|  |  |
| --- | --- |
|  | Время, сек |
| Передача №1 | 1.542 |
| Передача №2 | 1.492 |
| Передача №3 | 1.492 |
| Передача №4 | 1.532 |
| Передача №5 | 1.502 |
| Среднее | 1.512 |
| Дисперсия | 0.00055 |

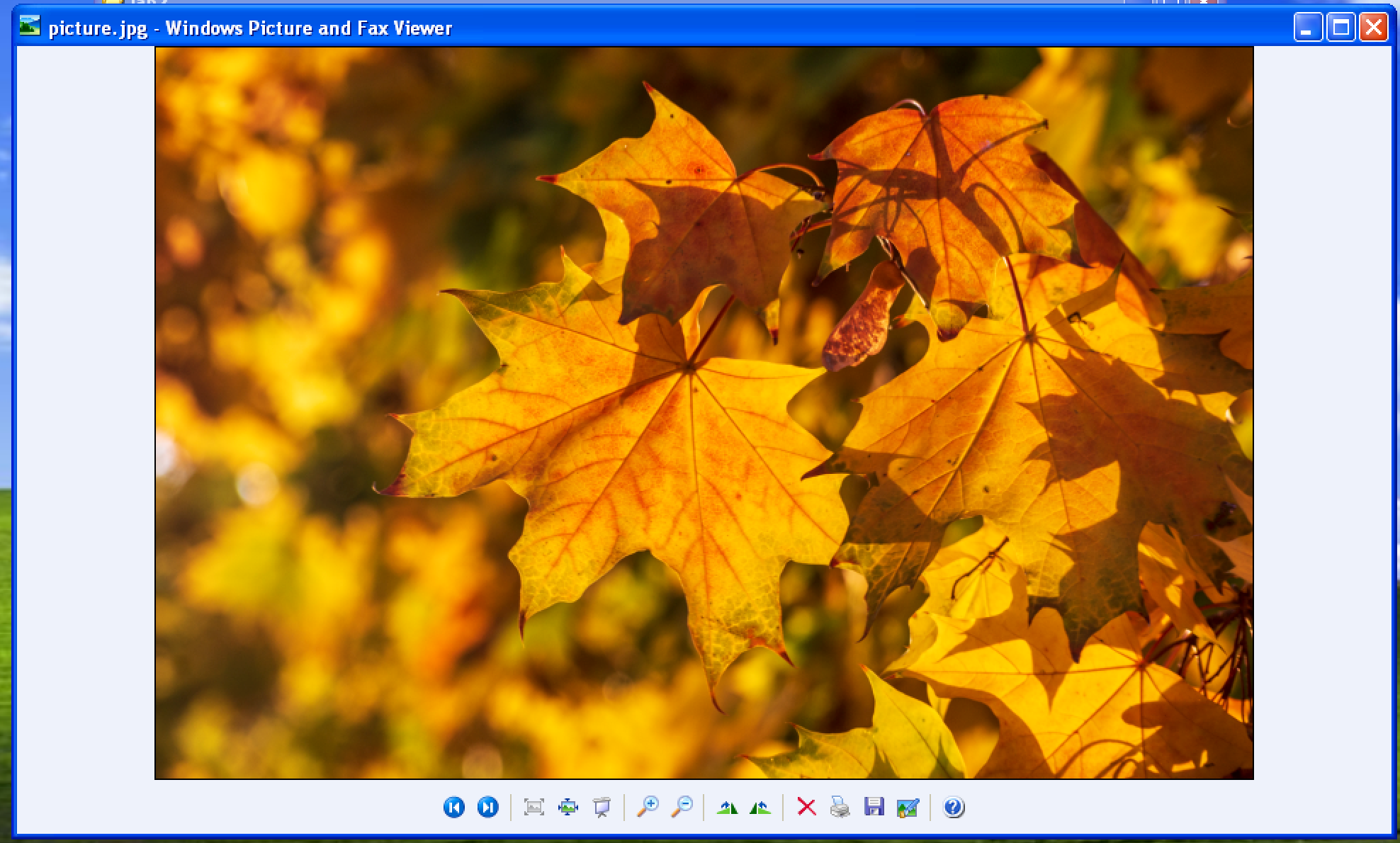
|  |  |
| --- | --- |
| Размер изображения Мбайт | Скорость передачи, Мбит/с |
| 21.3 | 112.6984127 |

Вывод по работе IPX: дисперсия малая, это значит протокол IPX стабилен. Получили скорость передачи в 112,7 Мбит/c. В прошлой работе скорость передачи изображения в 1,81 Мбайт для 3х клиентов составило 4,525 Мбит/c. Значит скорость выросла в 24,9 раза. Выделено было 1 ядро у каждой из машин, и было загружено на 100% при передаче. Можно было выделить больше и тогда скорость должна ещё вырасти.

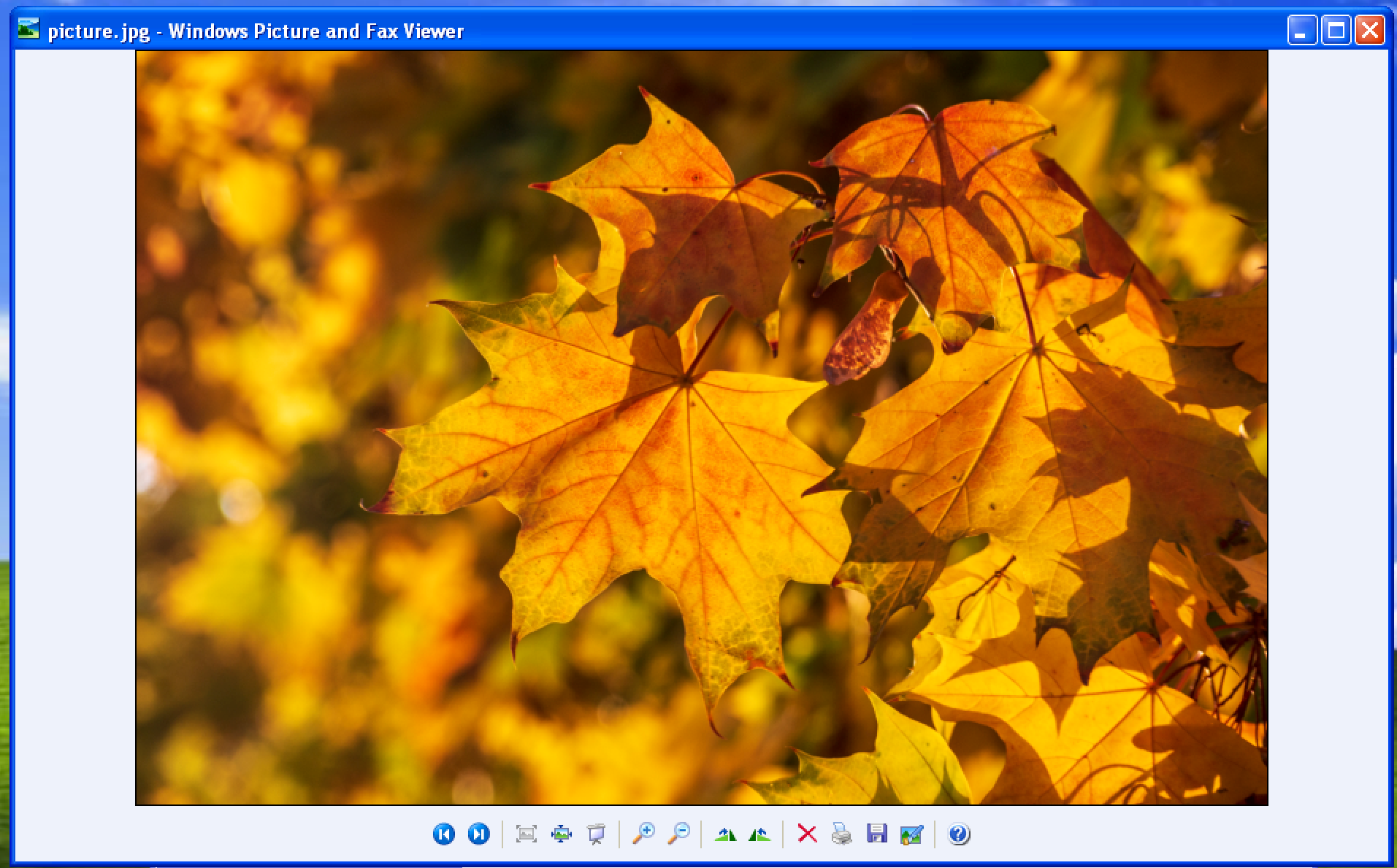
**Анализ SPX протокола**

Пример переданного изображения

Исходное изображение



Переданное изображение



Передача осуществлялась 3м клиентам

|  |  |
| --- | --- |
|  | Время, сек |
| Передача №1 | 22.482 |
| Передача №2 | 19.721 |
| Передача №3 | 20.693 |
| Передача №4 | 21.203 |
| Передача №5 | 19.668 |
| Среднее | 20.7534 |
| Дисперсия | 1.3594473 |

|  |  |
| --- | --- |
| Размер изображения Мбайт | Скорость передачи, Мбит/с |
| 21.3 | 8.2107 |

Вывод по работе SPX: дисперсия выше чем у IPX, но всё ещё не высокая. Такая дисперсия связана с началом передачи в ручном режиме и одновременно невозможно запустить на 3х клиентах. Скорость заметно уменьшилась и стала всего в 2 раза быстрее чем реализация IPX в MS-DOS. Время выполнения заметно увеличилось. Связано это с переотправкой повреждённых пакетов и возможно с заполненностью канала каждым клиентом из за отсутствия широковещательного канала. Но стоит отметить, что файлы передаются без повреждений, что достаточно важно.

**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.

Код программы:

Файл ipx\_client.cpp

#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 << "Bye" << std::endl;

return 0;

}

Файл ipx\_server.cpp

#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 Z to send file\nType anything else to stop server: ";

std::cout.flush();

std::cin >> input;

if (input != "Z") {

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("picture.jpg", "rb");

if (!source) {

std::cout << "picture 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 << "Bye" << std::endl;

return 0;

}

Файл spx\_client.cpp

#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;

}

Файл spx\_server.cpp

#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();

}