**СОДЕРЖАНИЕ**

[Введение 4](#_Toc55145659)

[1 Методы разработки программных средств с использованием сетевых технологий 5](#_Toc55145660)

[1.1 Сетевой протокол TCP/IP 5](#_Toc55145661)

[1.2 Инструменты работы с сокетами на C# 6](#_Toc55145662)

[1.3 Общий принцип работы сокетов 7](#_Toc55145663)

[1.4 Клиент-серверная архитектура 8](#_Toc55145664)

[1.5 Распределенные системы 9](#_Toc55145665)

[2 Архитектура приложения 12](#_Toc55145666)

[2.1 Принцип работы сервера 12](#_Toc55145667)

[2.2 Принцип работы клиента 12](#_Toc55145668)

[3 Описание разработанного приложения 14](#_Toc55145669)

[3.1 Реализация программы 14](#_Toc55145670)

[3.2 Руководство пользователя 15](#_Toc55145671)

[3.3 Вывод проделанной работы 18](#_Toc55145672)

[Заключение 19](#_Toc55145673)

[Список использованных источников 20](#_Toc55145674)

[Приложение А 21](#_Toc55145675)

[Приложение Б 27](#_Toc55145676)

Введение

Сегодня миллионы компьютеров и устройств связаны в глобальную сеть интернет, либо в отдельные локальные подсети. В связи с этим возникает необходимость создания приложений, которые бы использовали все преимущества передачи данных по сети.

Примером может служить распределённые вычисления.

Данный термин обычно применяют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Как результат, эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в многомашинных вычислительных комплексах, образуемых объединением нескольких отдельных электронно-вычислительных машин (ЭВМ) с помощью каналов связи локальных или глобальных информационных сетей. Последовательные вычисления в распределённых системах выполняются с учётом одновременного решения многих задач. Особенностью распределённых многопроцессорных вычислительных систем, в отличие от локальных суперкомпьютеров, является возможность неограниченного наращивания производительности за счёт масштабирования.

Вся сеть состоит из отдельных элементов – хостов, которые представляют собой компьютеры и другие подключенные устройства. Между собой они соединены каналами связи (кабели *Ethernet*, *Wi-Fi* и т.д.) и маршрутизаторами.

Маршрутизаторы объединяют компьютеры в подсети и контролируют передачу данных между ними. Компьютеры-хосты применяют протоколы.

Протокол представляет собой соглашения о том, как пакеты данных будут передаваться по каналам коммуникации. Таким образом, протокол упорядочивает взаимодействие.

Существует множество различных протоколов. Протоколы, которые используются для передачи данных по сети, составляют семейство протоколов *TCP/IP*.

Основные из них: *Internet Protocol* (*IP*), *Transmission Control Protocol* (*TCP*) и *User Datagram Protocol* (*UDP*).

В данном курсовом проекте использовался протокол *TCP*.

*TCP* позволяет отследить потерю пакетов и их дублирование при передаче.

Приложение взаимодействует с уровнем *TCP* не напрямую, а через специальный *API*, который предоставляют сокеты.

Сокеты – это интерфейс для создания сетевых приложений, который опирается на встроенные возможности операционной системы.

В рамках курсового проекта необходимо было написать приложение для решения нелинейных алгебраических уравнений в распределённой среде.

Использование распределённых вычислений для систем нелинейных уравнений позволяют ускорить получение решения путём параллельной обработкой информации и производя вычисления некоторыми частями.

Целью курсового проекта является разработка программного продукта для распределенной обработки изображений на основе усредняющего среднегеометрического фильтра.

Актуальность данного курсового проекта обусловлена тем, что сегодня сетевые технологии находятся на достаточно высоком уровне развития. А это значит, что программисты в этой области нужны не менее чем в других областях.

В ходе выполнения курсового проекта были выполнены следующие задачи:

– обработка изображения на основе усредняющего среднегеометрического фильтра;

– результаты вычислений сравнены с однопоточным приложением и многопоточными приложениями;

– распараллеливание реализовано на уровне алгоритма;

– обеспечена обработку изображения разрешением более 1000 х 1000 пикселей;

– сетевое взаимодействие организовано между вычислительными узлами с использованием протокола TCP/IP в явном виде с использованием сокетов;

– разработка и реализация графического интерфейса приложения.

Visual Studio Community – современная среда проектирования, позволяющая разрабатывать Web-приложения на платформе Windows, используя шаблон Asp.net MVC Core на языке программирования C#, с использованием средств HTML, CSS и JavaScript.

JavaScript – выполняется на стороне клиента и не имеет функционала для работы с сокетами, но через функционал fetch может взаимодействовать с MVC проектом через запросы.

В свою очередь C# имеет богатый функционал для работы с сокетами напрямую.

Для разработки приложения используются современные сетевые технологии и инструменты, которые необходимо знать с практического применения будущему программисту, что и обуславливает актуальность настоящей работы.

# 1 Методы разработки программных средств с использованием сетевых технологий

## 1.1 Сетевой протокол TCP/IP

Согласно определению протокол − это набор правил и методик, с помощью которых определяется порядок установления связи между компьютерами. Вполне естественно, что эти правила едины для всех участников диалога − компьютеров, между которыми происходит обмен информацией.

Рассмотрим два основных применяемых в *Internet* протокола: *IP* и *TCP*.

Одна из основных задач, выполняемых на сетевом уровне, − маршрутизация. В данном случае маршрутизация осуществляется на основе IP-адресов, присваиваемых устройствам в сети.

IP-адрес, позволяет однозначно идентифицировать любой компьютер, подключенный к *Internet*. Состоит *IP*-адрес из двух частей: первая часть служит для идентификации локальной сети в *Internet*, вторая − для идентификации компьютера в локальной сети. В этом случае части *IP*-адреса называются октетами. Все *IP*-адреса делятся на классы. Существует четыре класса *IP*-адресов (*А, В*, *С* и *D*). В настоящее время свободными остались только адреса класса *С*, в то время как адреса классов *А* и *В* давно зарезервированы крупными компаниями. Адреса класса *D* предназначаются для передачи широковещательных сообщений многочисленным пользователям (многоабонентская рассылка), В адресах, относящихся к классу А, первый октет определяет адрес сети, а три остальных октета − адрес хоста. В силу этого максимальное количество сетей класса А составляет 126.

Конечно, использование классов нельзя назвать наиболее эффективным способом использования ограниченного пространства IP-адресов. «Узкие рамки» допустимого количества IP-адресов не всегда позволяют «втиснуть» диапазон адресов, требуемых той или иной компании. И здесь на помощь приходит так называемая бесклассовая адресация. Именно этот метод назначения адресов предусматривается протоколом Ipv6, который для выделения IP-адресов обеспечивает адресное пространство размером 128 бит.

Транспортный уровень модели OSI обеспечивает надежную связь между компьютерами. Одним из механизмов, направленных на повышение надежности, является подтверждение приема данных компьютером без повреждений и потерь «по дороге». Протоколы этого уровня также ответственны за идентификацию получаемых сообщений. Разделение сообщений осуществляется с помощью различных портов. Порт представляет собой точку логического соединения, которая используется в целях идентификации конкретного приложения, осуществляющего прием/передачу сообщений. С портами связано понятие сокета − конечной точки соединения. Для практической реализации связи необходимо создание сокета. Один из распространенных вариантов сокетных интерфейсов называется Windows Sockets (Windsock). Пакет протоколов TCP/IP включает два протокола, имеющих отношение к транспортному уровню (TCP и UDP). Рассмотрим эти протоколы немного подробнее.

Основная задача протокола TCP − установка сеанса соединения между двумя компьютерами. При этом используются сообщения уведомления и ответа. Каждому пакету данных присваивается своя нумерация, что позволяет собирать их в правильной последовательности на принимающем конце. Благодаря нумерации принимающий компьютер обнаруживает «недостачу» пакетов данных, в результате надежность протокола TCP значительно повышается, но происходит значительное снижение производительности.

## 1.2 Инструменты работы с сокетами на C#

В основе межсетевых взаимодействий по протоколу TCP лежат сокеты. В .NET сокеты представлены классом System.NET.Sockets.Socket, который предоставляет низкоуровневый интерфейс для приема и отправки сообщений по сети.

Рассмотрим основные свойства данного класса.

AddressFamily − возвращает все адреса, используемые сокетом. Данное свойство представляет одно из значений, определенных в одноименном перечислении AddressFamily. Перечисление содержит 18 различных значений, наиболее используемые:

Available − возвращает объем данных, которые доступны для чтения;

Connected − возвращает true, если сокет подключен к удаленному хосту;

LocalEndPoint − возвращает локальную точку, по которой запущен сокет и по которой он принимает данные;

ProtocolType − возвращает одно из значений перечисления ProtocolType, представляющее используемый сокетом протокол;

RemoteEndPoint − возвращает адрес удаленного хоста, к которому подключен сокет;

SocketType − возвращает тип сокета. Представляет одно из значений из перечисления SocketType.

Для создания объекта сокета можно использовать один из его конструкторов, как показано на рисунке 1.1.

Socket socket = new Socket(AddressFamily. InterNetwork, SocketType.Stream, ProtocolType.Tcp);

Рисунок 1.1 – Создание сокета

Таким образом, при создании сокета можно указывать разные комбинации протоколов, типов сокета, значений из перечисления AddressFamily. Однако в то же время не все комбинации являются корректными. Так, для работы через протокол Tcp, необходимо обязательно указать параметры: AddressFamily.InterNetwork, SocketType.Stream и ProtocolType.Tcp.

Поэтому использование сокетов может потребовать некоторого знания принципов работы отдельных протоколов.

## 1.3 Общий принцип работы сокетов

При работе с сокетами вне зависимости от выбранных протоколов необходимо использовать методы класса Socket:

Accept() − создает новый объект Socket для обработки входящего подключения;

Bind() − связывает объект Socket с локальной конечной точкой;

Close() − закрывает сокет;

Connect() − устанавливает соединение с удаленным хостом;

Listen() − начинает прослушивание входящих запросов;

Poll() − определяет состояние сокета;

Receive() − получает данные;

Send() − отправляет данные;

Shutdown() − блокирует на сокете прием и отправку данных;

При применении протокола TCP, который требует установление соединения, сервер должен вызвать метод Bind для установки точки для прослушивания входящих подключений и затем запустить прослушивание подключений с помощью метода Listen. Далее с помощью метода Accept можно получить входящие запросы на подключение в виде объекта Socket, который используется для взаимодействия с удаленным узла. У полученного объекта Socket вызываются методы Send и Receive соответственно для отправки и получения данных. Если необходимо подключиться к серверу, то вызывается метод Connect. Для обмена данными с сервером также применяются методы Send или Receive.

## 1.4 Клиент-серверная архитектура

Сервер (аппаратное обеспечение) – компьютер (или специальное компьютерное оборудование), выделенный и/или специализированный для выполнения определенных сервисных функций. Клиент – аппаратный или программный компонент вычислительной системы, посылающий запросы серверу.

В базовой модели клиент-сервер все процессы в распределенных системах делят­ся на две возможно перекрывающиеся группы. Процессы, реализующие некото­рую службу, например службу файловой системы или базы данных, называются серверами. Процессы, запрашивающие службы у серверов путем посыл­ки запроса и последующего ожидания ответа от сервера, называются клиентами. Взаимодействие клиента и сервера, известное также под названием ре­ жим работы запрос-ответ, что показано на рисунке 1.2.

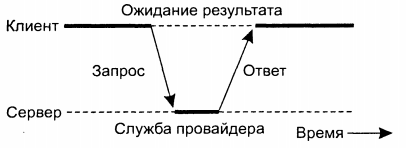


Рисунок 1.2 – Обобщенное взаимодействие между клиентом и сервером

Если базовая сеть так же надежна, как локальные сети, взаимодействие меж­ду клиентом и сервером может быть реализовано посредством простого протоко­ла, не требующего установления соединения. В этом случае клиент, запрашивая службу, облекает свой запрос в форму сообщения с указанием в нем службы, к ко­торой он желает воспользоваться, и необходимых для этого исходных данных. Затем сообщение посылается серверу. Последний, в свою очередь, постоянно ожидает входящего сообщения, получив его, обрабатывает, упаковывает резуль­тат обработки в ответное сообщение и отправляет его клиенту. Использование не требующего соединения протокола дает существенный вы­игрыш в эффективности. До тех пор, пока сообщения не начнут пропадать или повреждаться, можно вполне успешно применять протокол типа запрос-ответ. К сожалению, создать протокол, устойчивый к случайным сбоям связи, − нетривиальная задача. Все, что можно сделать, − это дать клиенту возможность повторно послать запрос, на который не был получен ответ. Проблема, однако, состоит в том, что клиент не может определить, действительно ли первоначаль­ное сообщение с запросом было потеряно или ошибка произошла при передаче ответа. Если потерялся ответ, повторная посылка запроса может привести к по­вторному выполнению операции.

В качестве альтернативы во многих системах клиент-сервер используется на­дежный протокол с установкой соединения. Хотя это решение в связи с его отно­сительно низкой производительностью не слишком хорошо подходит для ло­кальных сетей, оно великолепно работает в глобальных системах, для которых ненадежность является «врожденным» свойством соединений. Так, практически все прикладные протоколы Интернета основаны на надежных соединениях по протоколу TCP/IP.

В этих случаях всякий раз, когда клиент запрашивает служ­бу, до посылки запроса серверу он должен установить с ним соединение. Сервер обычно использует для посылки ответного сообщения то же самое соединение, после чего оно разрывается. Проблема состоит в том, что установка и разрыв со­единения в смысле затрачиваемого времени и ресурсов относительно дороги, особенно если сообщения с запросом и ответом невелики.

## 1.5 Распределенные системы

Несмотря на рост вычислительной мощности ПК, многие задачи по- прежнему требовали много больших вычислительных ресурсов. Появилась необходимость создания нового типа взаимодействия, новой структуры, обеспечивающей распределенную обработку информации. В этой модели взаимодействия каждая из машин призвана решать свои задачи, что делает возможной специализацию: каждый компьютер работает над конкретной задачей, для решения которой он оптимизирован (в модели «клиент-сервера» сервер тоже специализирован и выполняет свои специфические задачи, но он при этом «самодостаточен» и никак не связан с другими серверами). При этом для решения задач ему необходимо получать результаты работы других ПК и, в свою очередь, передавать им свои результаты, что стало возможным только с объединением компьютеров в вычислительную сеть.

Распределение задач между компьютерами сети позволяет расширить функциональные возможности каждого из них путем организации совместного доступа к ресурсам. Одной из заметных тенденций развития вычислительной индустрии стала модель совместной обработки данных. В этой модели несколько компьютеров используются для решения одной и той же задачи, а не только для обмена результатами вычислений. При использовании модели совместных вычислений возрастают суммарная вычислительная мощность и доступные ресурсы (оперативная и дисковая память), повышается отказоустойчивость всей системы в целом.

Несмотря на то что все распределенные системы содержат по нескольку процес­соров, существуют различные способы их организации в систему. В особенности это относится к вариантам их соединения и организации взаимного обмена.

За прошедшие годы было предложено множество различных схем классифи­кации компьютерных систем с несколькими процессорами, но ни одна из них не стала действительно популярной и широко распространенной. Нас интересуют исключительно системы, построенные из набора независимых компьютеров. На рисунке 1.3 подразделяются все компьютеры на две группы.

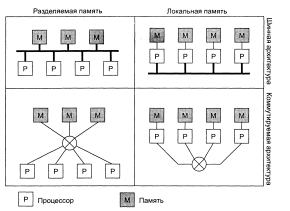


Рисунок 1.3 − Различные базовые архитектуры процессоров и памяти распределенных компьютерных систем

Системы, в которых компьютеры используют память совместно, обычно называются мультипроцес­сорами (multiprocessors), а работающие каждый со своей памятью − мультикомпьютерами (muldcomputers). Основная разница между ними состоит в том, что мультипроцессоры имеют единое адресное пространство, совместно используе­мое всеми процессорами. Если один из процессоров записывает, например, зна­чение 44 по адресу 1000, любой другой процессор, который после этого прочтет значение, лежащее по адресу 1000, получит 44. Все машины задействуют одну и ту же память. В отличие от таких машин в мультикомпьютерах каждая машина использует свою собственную память. После того как один процессор запишет значение 44 по адресу 1000, другой процессор, прочитав значение, лежащее по адресу 1000, получит то значение, которое хранилось там раньше. Запись по этому адресу значения 44 другим процессором никак не скажется на содержимом его памяти. Типичный пример мультикомпьютера − несколько персональных компьютеров, объединенных в сеть. Каждая из этих категорий может быть подразделена на дополнительные кате­ гории на основе архитектуры соединяющей их сети.

На рисунке 1.3 эти две архитек­туры обозначены как шинная (bus) и коммутируемая (switched). Под шиной по­нимается одиночная сеть, плата, шина, кабель или другая среда, соединяющая все машины между собой. Подобную схему использует кабельное телевидение: кабельная компания протягивает вдоль улицы кабель, а всем подписчикам дела­ются отводки от основного кабеля к их телевизорам.

Коммутируемые системы, в отличие от шинных, не имеют единой магистрали, такой как у кабельного телевидения. Вместо нее от машины к машине тянутся отдельные каналы, выполненные с применением различных технологий связи. Сообщения передаются по каналам с принятием явного решения о коммутации с конкретным выходным каналом для каждого из них. Так организована гло­бальная телефонная сеть. Для гомогенных мультикомпьютерных систем характерна одна соединяющая компьютеры сеть, ис­пользующая единую технологию. Одинаковы также и все процессоры, которые в основном имеют доступ к одинаковым объемам собственной памяти. Гомоген­ные мультикомпьютерные системы нередко используются в качестве параллель­ных (работающих с одной задачей), в точности как мультипроцессорные. В отличие от них гетерогенные мультикомпьютерные системы могут содер­жать целую гамму независимых компьютеров, соединенных разнообразными сетя­ми. Так, например, распределенная компьютерная система может быть построе­на из нескольких локальных компьютерных сетей, соединенных коммутируемой магистралью.

В отличие от мультипроцессоров построить мультикомпьютерную систему отно­сительно несложно. Каждый процессор напрямую связан со своей локальной па­мятью. Единственная оставшаяся проблема − это общение процессоров между собой. Понятно, что и тут необходима какая-то схема соединения, но поскольку интересует только связь между процессорами, объем трафика будет на не­сколько порядков ниже, чем при использовании сети для поддержания трафика между процессорами и памятью.

В мультикомпьютерных системах с шинной архитектурой процессоры соеди­няются при помощи разделяемой сети множественного доступа, например Fast Ethernet. Скорость передачи данных в сети обычно равна 100 Мбит/с. Как и в случае мультипроцессоров с шинной архитектурой, мультикомпьютерные систе­мы с шинной архитектурой имеют ограниченную масштабируемость. В зависи­мости от того, сколько узлов в действительности нуждаются в обмене данными, обычно не следует ожидать высокой производительности при превышении сис­темой предела в 25-100 узлов.

# 2 Архитектура приложения

## 2.1 Принцип работы сервера

Сервер получает задачу от клиента, выполняет фильтрацию данных, и отправляет результат клиенту.

Функциональная схема сервера показана на рисунке 2.1.

Обрабатывает данные

Отправляет данные

Закрывает сокет

Конец

Сервер

Устанавливает конечную точку

Создает сокет TCP/IP

Назначает сокет локальной конечной точке

Слушает входящие сокеты

Ожидает подключение

Получает данные

Выполняет расчеты

Рисунок 2.1 – Функциональная схема сервера

## 2.2 Принцип работы клиента

Клиент формирует задачу, последовательно отправляет задачу по частям всем серверам, переходит в режим ожидания получения данных от всех серверов, последовательно получает обработанные части от серверов и выводит результат.

Функциональная схема клиента показана на рисунке 2.2.

Клиент

Подготовка данных для отправки

Устанавливает удаленную точку i

Создает сокет с типом TCP

Соединяет сокет с удаленной точкой i

Отправляет данные на удаленную точку i

Получает данные от удаленной точки i

Освобождает сокет

i=0,n

i=0,n

Формирует результат

Конец

Рисунок 2.2 – Функциональная схема клиента

# 3 Описание разработанного приложения

## 3.1 Реализация программы

Диаграмма сотрудничества показана на рисунке 3.1.

HTML, CSS

JavaScript

Asp.Net MVC

Сервер (консольное приложение)

Рисунок 3.1 – Диаграмма сотрудничества

## 3.2 Руководство пользователя

Для правильной работы программного комплекса необходимо запустить минимум одну копию серверной программы ServerImg.exe и одну копию клиентского приложения ImageProcessin.

***3.2.1*** Запуск. После запуска двух копий ServerImg.exe и одной копии клиента, экран монитора выглядит, как показано на рисунке 3.2.

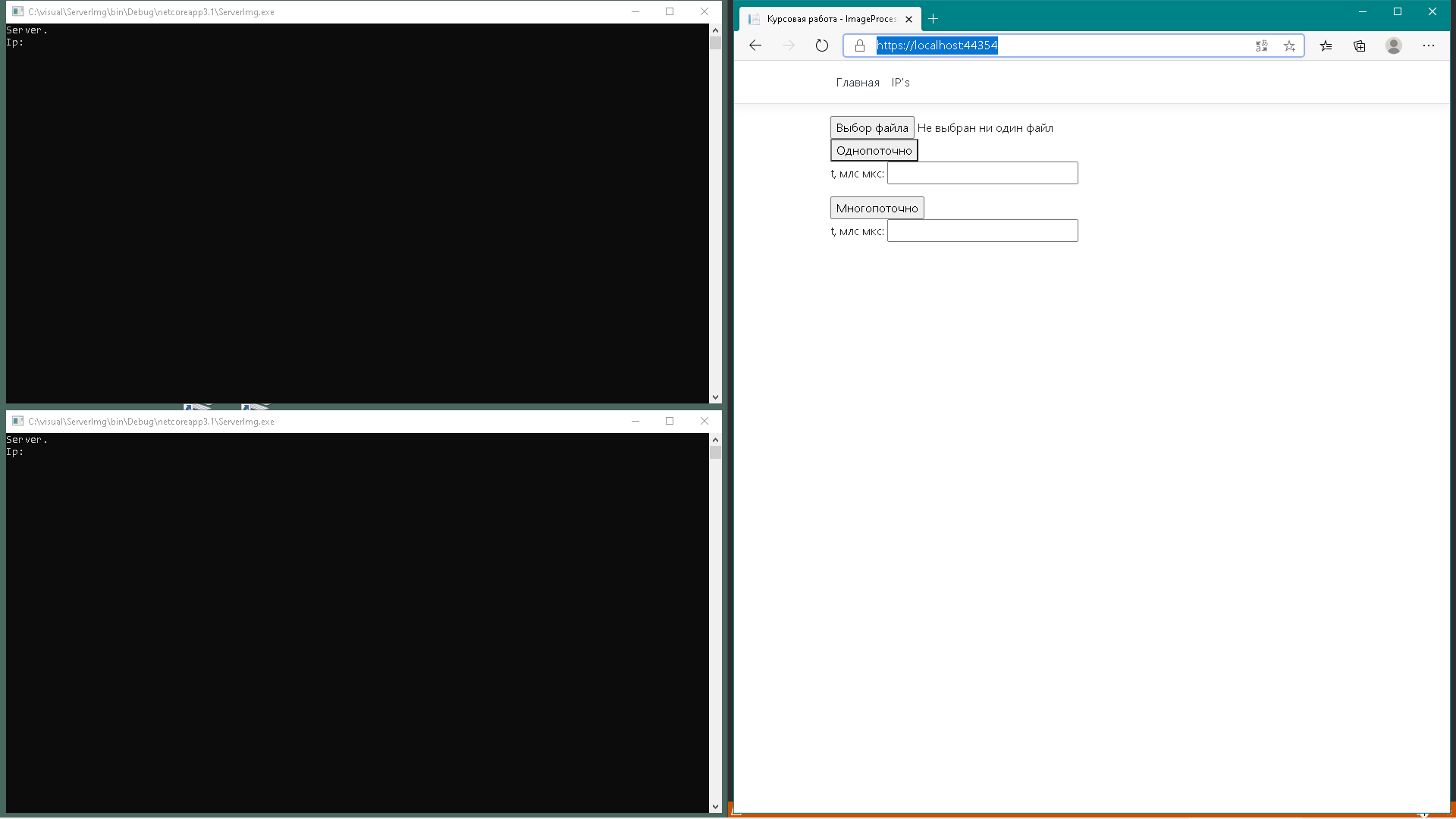


Рисунок 3.2 – Запуск программного комплекса

Из рисунка видно, что сервера перешли в режим ожидания ввода Ip-адреса прослушки. Запускаем ipconfig, что узнать адрес, как показано на рисунке 3.3.

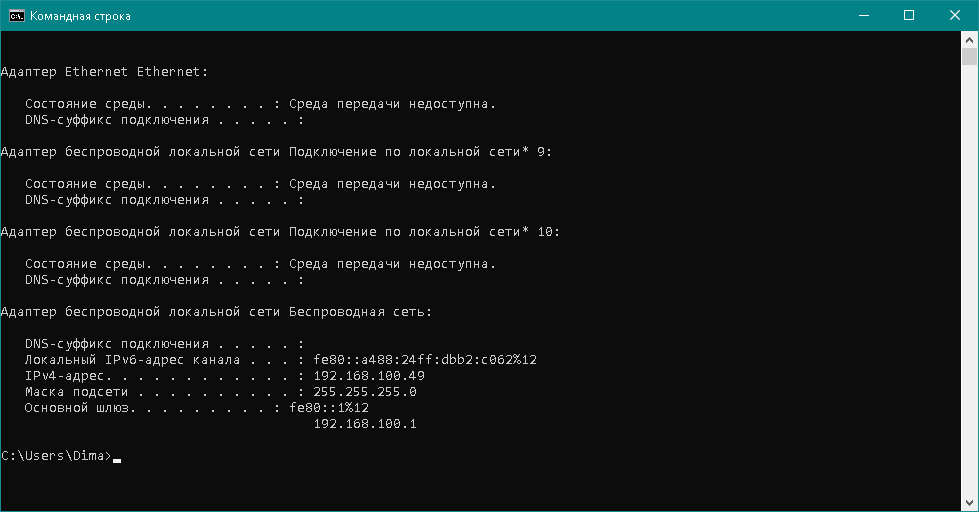


Рисунок 3.3 – Ввод параметров первого сервера

Вводим 192.168.100.49 и порты 10000, 10001, как показано на рисунке 3.4. Из рисунка видно, что сервера перешли в режим ожидания подключения клиента.

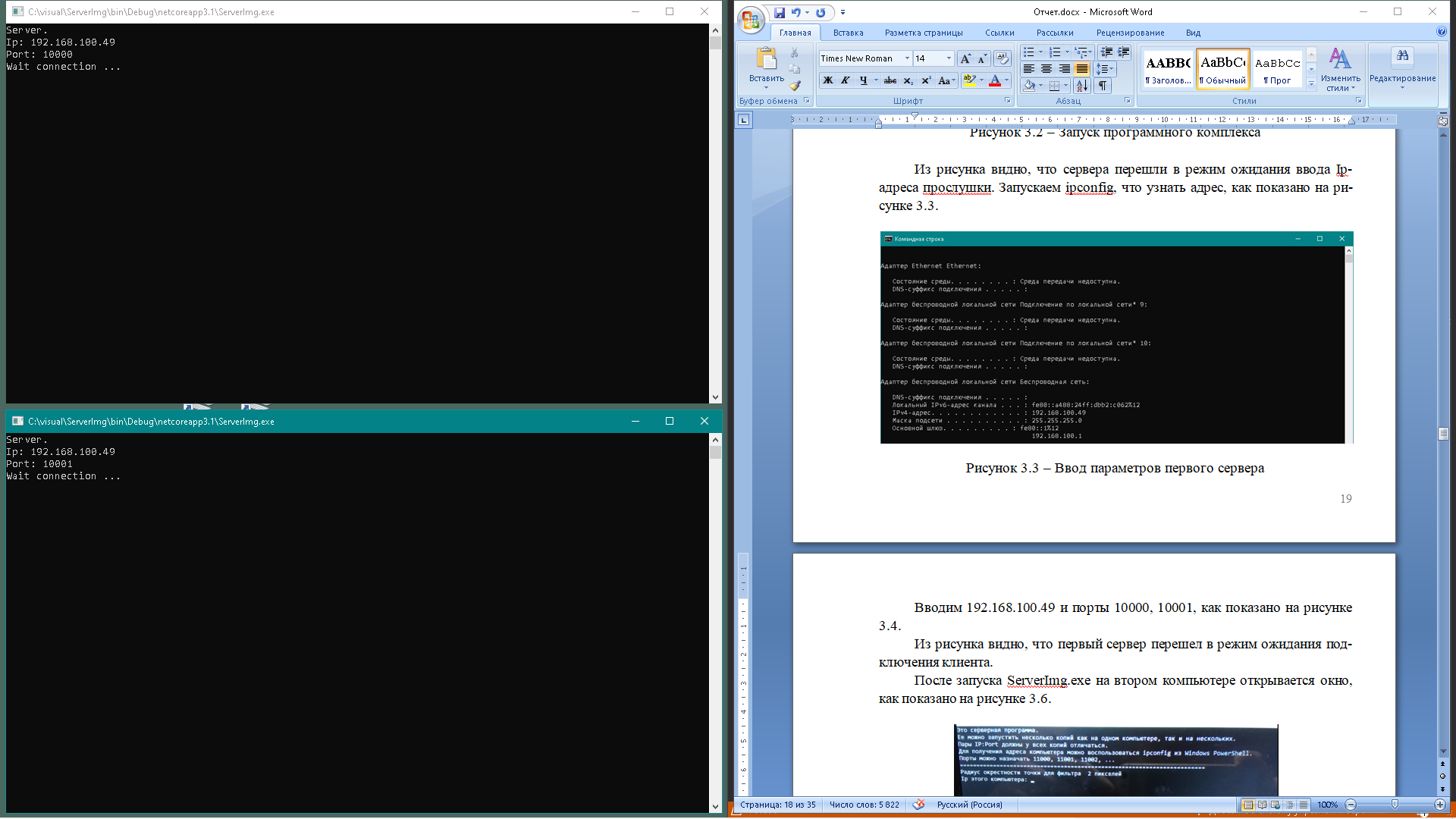


Рисунок 3.4 – Ожидание подключение клиентов

Далее переходим на клиентскую программу, вкладку Ip’s и вводим ip-адреса серверов и их порты, как показано на рисунке 3.5.

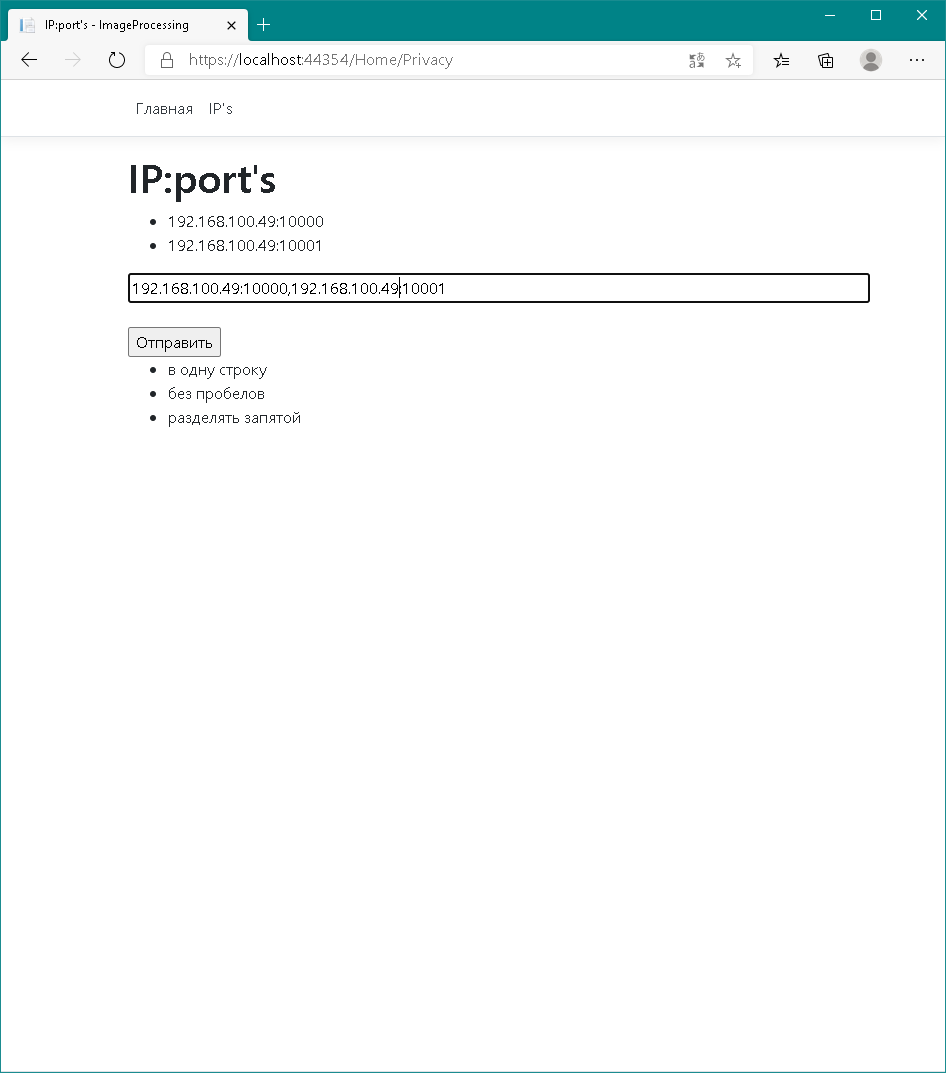


Рисунок 3.5 – Ввод параметров серверов на стороне клиента

***3.2.2*** Линейная обработка. Чтобы начать обработку изображения, его нужно выбрать, нажав на кнопку «Выбор файл». Изображение выбирается на стороне клиента. Результат показан на рисунке 3.6.

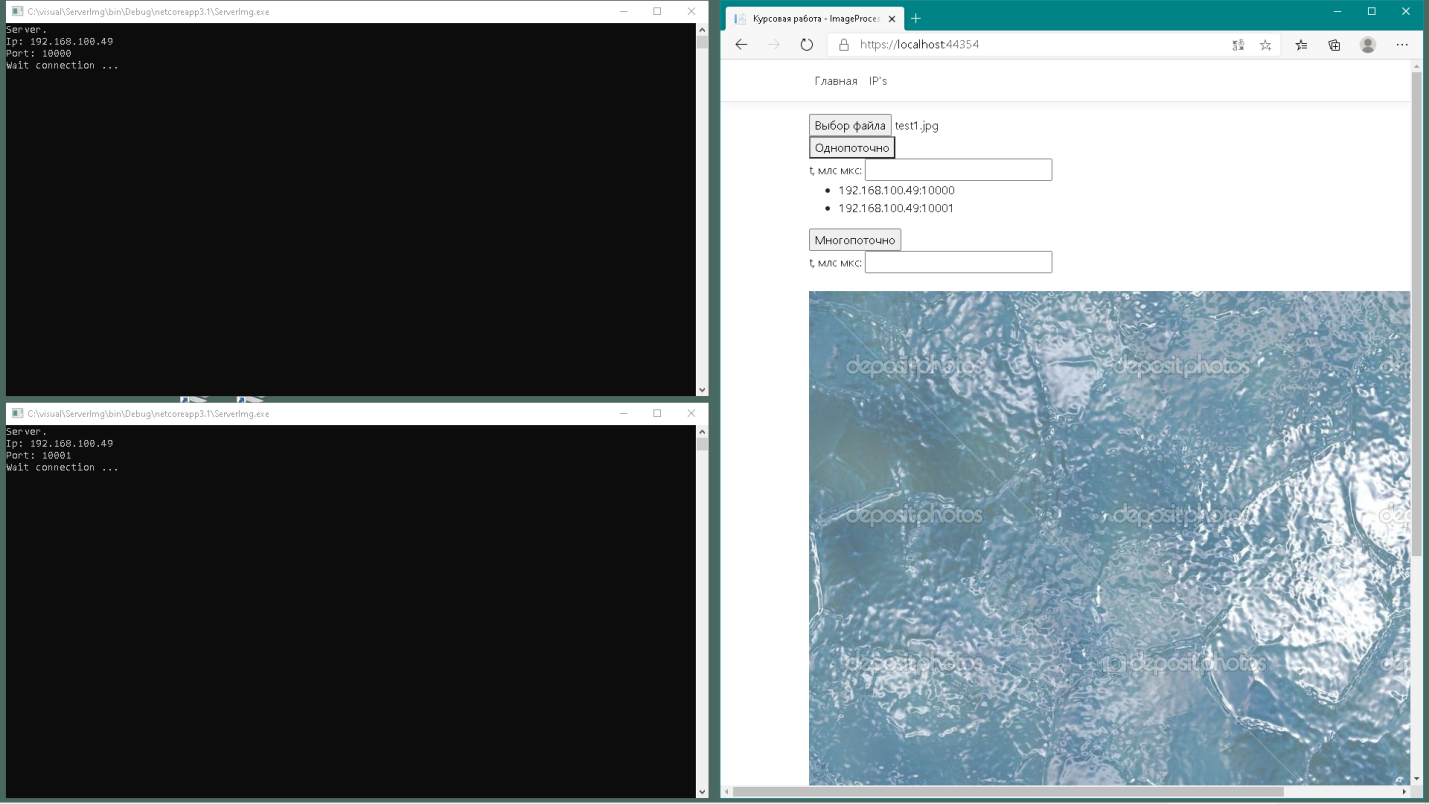


Рисунок 3.6 – Выбор изображения для обработки

Далее необходимо нажать кнопку «Однопоточно», чтобы программа применила фильтр для изображения и вывела полученный результат. После выполнения всех вышесказанных действий и некоторого времени ожидания обработки, полученный результат показан на рисунке 3.6.

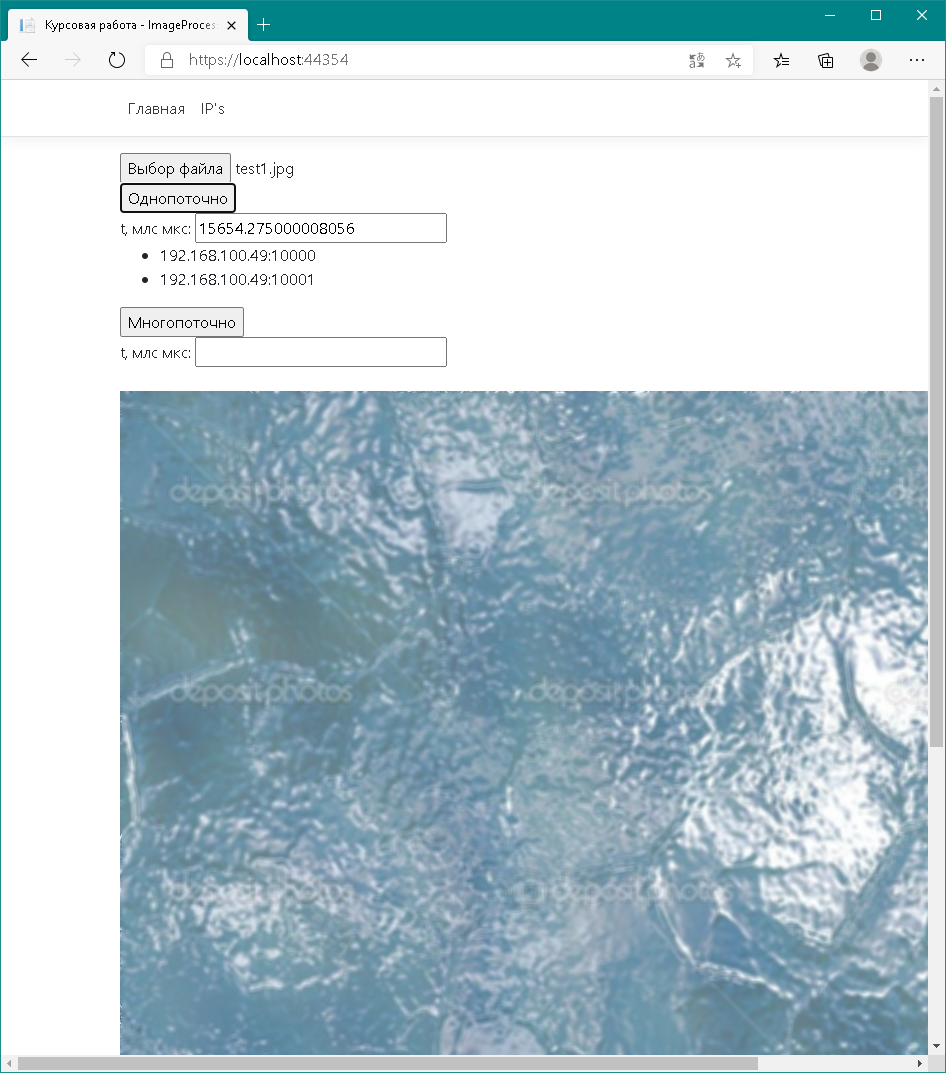


Рисунок 3.6 – Результат обработки однопоточным расчетом

Из рисунка видно, что обработка заняла 15,6 секунды.

***3.2.4*** Многопоточная обработка. Итак, ранее рассмотрели, как ввести ip-адреса серверов и загрузку изображения. Далее необходимо нажать кнопку «Многопоточно», чтобы выполнить расчет на двух серверах. Результат показана на рисунке 3.7.

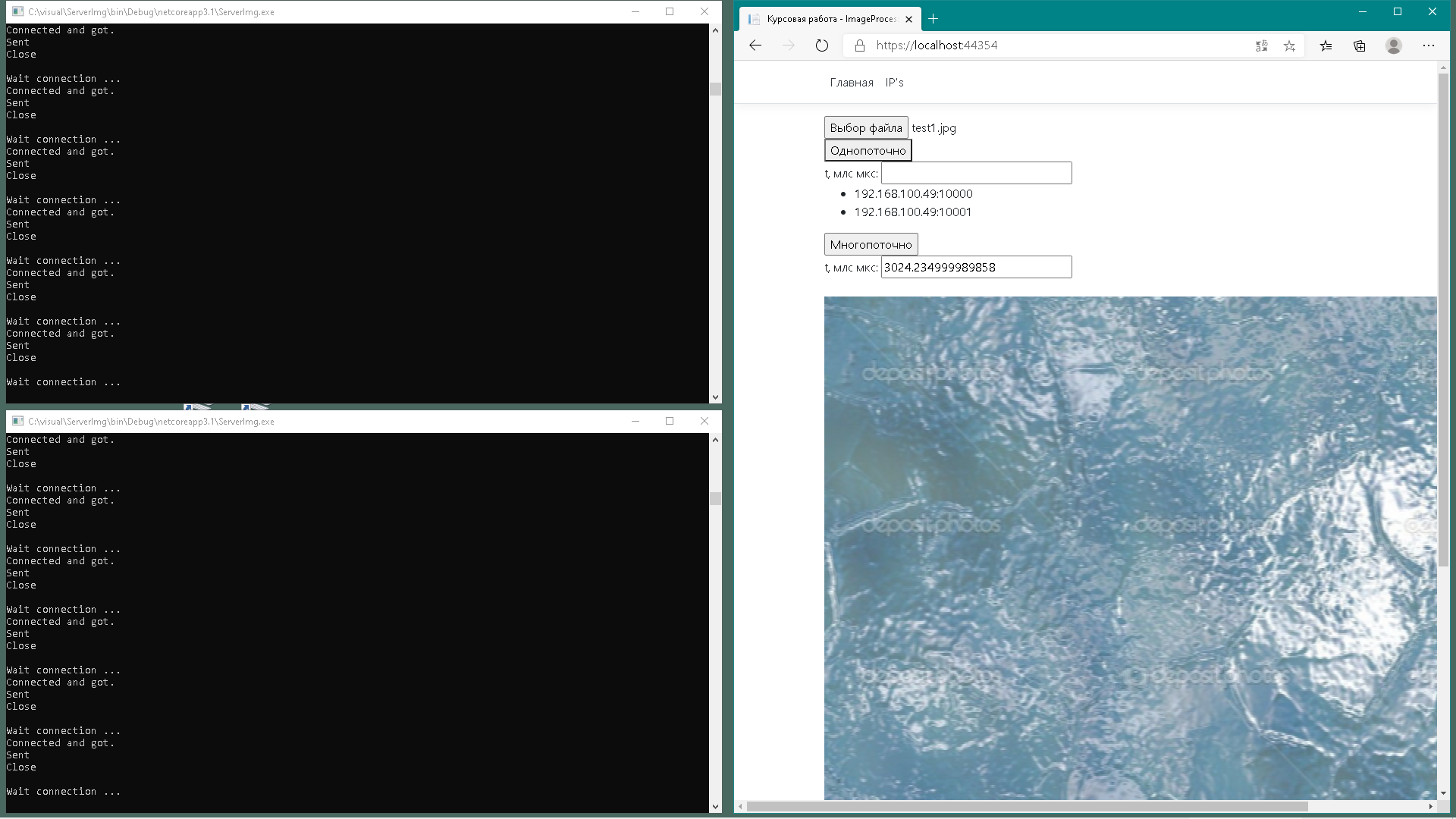


Рисунок 3.7 – Результат многопоточной обработки

Обработка заняла 3 секунды. При сравнении необходимо учесть тот факт, что однопоточная обработка выполнялась программой на javascript, а распределенная обработка на C#.

## 3.3 Вывод проделанной работы

Функция fetch JavaScript отправляет/получает запрос объёмом максимум 16 кб, что установили документально. Обработка изображения при помощи JavaScript гораздо дольше, чем обработка кодом, полученного компиляцией C#. На JavaScript реализована однопоточная обработка, которая заняла 15,6с. Серверная обработка реализована на C#. В распределенном решении суммарная обработка изображения заняла 3с. Не учитывая, тот факт, что JavaScript отработал медленнее, явно можно увидеть более быструю распределенную обработку большого изображения – в 5 раз быстрее.

Заключение

Распределенные системы состоят из автономных компьютеров, которые работа­ют совместно, представая в виде единой связной системы. Их важное преимуще­ство состоит в том, что они упрощают интеграцию различных приложений, рабо­тающих на разных компьютерах, в единую систему и ускоряют вычислительные процессы конечной системы.

В результате выполнения данной курсовой работы был разработан сетевой программный комплекс для распределенной обработки большого изображения.

Приложение состоит из двух программных частей и имеет простой и понятный графический интерфейс и руководство пользователя, что позволяет пользователю быстро освоиться в среде вычислений.

Для создания программного комплекса было использовано следующее:

– протокол IP из стека протоколов TCP/IP;

– базовые знания языков программирования С#, Javascript, HTML, CSS;

– базовые знания шаблона проектирования Asp.Net MVC Core;

– среда разработки Microsoft Visual Community 2019.

В процессе разработки сетевого приложения была написана программная реализация для сервера и клиента. Основные функции сервера: получение части изображения от клиента, фильтрация изображения и отправка клиенту. Функция клиента – деление изображения на части, отправки частей серверам, сбор фильтрованной информации от серверов, формирование общего изображения и вывод изображения.

Клиент разработан по шаблону Asp.Net MVC Core. Сервер разработан как консольное приложение.

Все поставленные задачи по созданию клиент-серверного приложения успешно решены, а именно: разработано клиент-серверное приложение, использующее протокол TCP.

Поэтому данный курсовой проект даёт возможность глубже изучить пройденный теоретический материал, позволяет закрепить навыки решения поставленных задач в короткие сроки и обучиться поиску необходимой для этого информации, а также применить на практике все теоретические знания, полученные по данной дисциплине.

Список использованных источников

1. Структура стека протоколов TCP/IP [Электронный ресурс]: Информационный портал – Электронные данные. – Режим доступа: www.agpu. net/fakult/ipimif/fpiit/kafinf/umk/el\_lib/calc\_system/lab\_work\_net/kulgin\_3.htm– Дата доступа: 14.10.2020.

2. Классическая архитектура «клиент-сервер» [Электронный ресурс]. – Режим доступа: http://www.webbcare.org/lectures/poks/Lecture1.pdf – Дата доступа: 13.10.2020.

3. Сокет (программный интерфейс) [Электронный ресурс]: Информационный портал – Википедия. – Режим доступа: https://ru.wikipedia.org/ wiki/Сокет\_(программный\_интерфейс) – Дата доступа: 14.10.2020.

4. Варианты построения серверных и клиентских приложений [Электронный ресурс]. – Режим доступа: http://life-prog.ru/ view\_ shpargalkiCompStroi.php?id=105 – Дата доступа: 13.10.2020.

5. Рихтер Дж., CLR via С#. Программирование на платформе Microsoft .NET Framework 4.0 на языке С#. 3-е изд./ Дж. Рихтер. – СПб.: Питер, 2020.– 709с.

6. Объектно-ориентированное программирование // Написание сетевого приложения [Электронный ресурс]. – 2014. – Режим доступа: http://ru.wikipedia.org/wiki/ООП. – Дата доступа: 23.10.2020.

7. Джонс, Э. Программирование в сетях Microsoft Windows/ Э. Джонс. – М.: И.Д. Вильямс, 2004. – 752 с.

8. Портал Life-Prog [Электронный ресур]: портал. – Электрон. дан. М., 2014. Режим доступа: http://www.life-prog.ru. Дата: 01.11.2020.

9. UNIX разработка сетевых приложений, создание сокетов / Р.Стивенс [и др.]. – К.: ДиаСофт, 2001. – 736 с.

10. Протоколы TCP и UDP [Электронный ресурс]. – Режим доступа: http://professorweb.ru/my/csharp/web/level1/1\_4.php – Дата доступа: 13.10.2020.

Приложение А

(обязательное)

Листинг приложения

HomeController.cs

using System;

using System.Diagnostics;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Logging;

using ImageProcessing.Models;

using System.Net;

using System.Net.Sockets;

using Microsoft.AspNetCore.Http;

using System.Collections.Generic;

namespace ImageProcessing.Controllers

{

public class HomeController : Controller

{

private readonly ILogger<HomeController> \_logger;

private static int count = 0;

private static int[] ports = null;

private static string[] ips = null;

private static Socket[] sockets = null;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

public IActionResult Index()

{

getip();

return View();

}

[HttpPost]

[RequestSizeLimit(40000000)]

public IActionResult Make()

{

// Получаем массив для обработки.

int LEN = 4 \* 1000 \* 4;

byte[] msg = new byte[LEN];

Request.Body.ReadAsync(msg, 0, LEN);

// Отслылаем серверам массив по частям.

for (int i = 0; i < count; i++)

{

byte[] r = new byte[LEN / count];

Array.Copy(msg, i \* LEN / count, r, 0, LEN / count);

sockets[i] = SendMessageFromSocket(ips[i], ports[i], r);

}

byte[] R = new byte[LEN];

// Получаем обработанные части от серверов.

// Составляем общий массив R для отправки.

for (int i = 0; i < count; i++)

{

byte[] r = ReceiveMessageFromSocket(sockets[i]);

Array.Copy(r, 0, R, i \* LEN / count, LEN / count);

}

Response.ContentType = "application/octet-stream";// Request.ContentType;

Response.Body.WriteAsync(R, 0, R.Length);

return Ok();

}

private void getip()

{

ViewBag.Param = "";

if (count != 0)

{

List<String> s = new List<string>();

for (int i = 0; i < count; i++)

{

s.Add(ips[i] + ":" + ports[i]);

}

ViewBag.Param = s;

}

}

public IActionResult Privacy()

{

getip();

return View();

}

[HttpPost]

public IActionResult Privacy2(String list)

{

string[] L = list.Split(',');

count = L.Length;

ports = new int[count];

ips = new string[count];

sockets = new Socket[count];

for (int i = 0; i < count; i++)

{

string[] R = L[i].Split(':');

ports[i] = int.Parse(R[1]);

ips[i] = R[0];

}

getip();

return Redirect("Privacy");

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

private Socket SendMessageFromSocket(string ip, int port, byte[] msg)

{

// Соединяемся с удаленным устройством

IPAddress ipAddr = IPAddress.Parse(ip);

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

Socket sender = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

// Соединяем сокет с удаленной точкой

sender.Connect(ipEndPoint);

// Отправляем данные через сокет

int bytesSent = sender.Send(msg);

return sender;

}

private byte[] ReceiveMessageFromSocket(Socket sender)

{

// Получаем ответ от сервера

// Буфер для входящих данных

byte[] bytes = new byte[4 \* 1000 \* 4 / count];

// Получаем ответ от сервера

int bytesRec = sender.Receive(bytes);

// Освобождаем сокет

sender.Shutdown(SocketShutdown.Both);

sender.Close();

return bytes;

}

}

}

JavaScript:

// Объект загрузки файла.

const file = document.getElementById("file-id");

// Изображение.

const img = document.getElementById("img-id");

// Кнопка обработать.

const but = document.getElementById("but");

// Время

const t1= document.getElementById("t1");

const t2 = document.getElementById("t2");

function fun() {

// Создаем канвас.

const canvas = document.createElement("canvas");

var ctx = canvas.getContext("2d");

canvas.width = img.width;

canvas.height = img.height;

// Копируем изображение в канвас.

ctx.drawImage(img, 0, 0);

// Получаем данные о точках.

var imgData = ctx.getImageData(0, 0, img.width, img.height);

(async () => {

var time = performance.now();

var LEN = 4 \* 1000 \* 4;

for (var j = 0; j < 1000 \* 1000 \* 4; j += LEN) {

// Копируем массив байт.

var data = imgData.data.slice(j, j + LEN - 1);

const res = await fetch('Home/Make', { method: 'POST', body: data });

const arr = await res.arrayBuffer();

var byteArray = new Uint8Array(arr);

for (var i = 0; i < byteArray.byteLength; i++) {

imgData.data[j + i] = byteArray[i];

}

};

time = performance.now() - time;

t2.value = '' + time;

// Подмена данных о точках.

ctx.putImageData(imgData, 0, 0);

// Подмена изображения.

img.src = canvas.toDataURL("image/png");

})();

}; // fun

// Событие загрузки файла.

file.addEventListener("change", function () {

var reader = new FileReader();

reader.onload = function (e) {

img.src = e.target.result;

}

reader.readAsDataURL(this.files[0]);

});

// Событие нажатия кнопки "Обработать".

but.addEventListener("click", function () {

// Создаем канвас.

const canvas = document.createElement("canvas");

var ctx = canvas.getContext("2d");

canvas.width = img.width;

canvas.height = img.height;

// Копируем изображение в канвас.

ctx.drawImage(img, 0, 0);

// Получаем данные о точках.

var imgData = ctx.getImageData(0, 0, img.width, img.height);

// Копируем массив байт.

var data = imgData.data.slice();

// Радиус окрестности точки.

var s = 2;

var time = performance.now();

// По всем точкам изображения.

for (var i = s; i < img.height - s; i++) {

for (var j = s; j < img.width - s; j++) {

// Количество точек в окрестности.

var mn = Math.pow(2 \* s + 1, 2);

// По окрестности точки (i,j).

var sr = 1.0;

var sg = 1.0;

var sb = 1.0;

// Применяем среднегармонический фильтр.

for (var p = i - s; p <= i + s; p++) {

for (var q = j - s; q <= j + s; q++) {

var r = data[p \* img.width \* 4 + q \* 4 + 0];

var g = data[p \* img.width \* 4 + q \* 4 + 1];

var b = data[p \* img.width \* 4 + q \* 4 + 2];

sr \*= r;

sg \*= g;

sb \*= b;

}

}

var fr = Math.pow (sr, 1.0/mn);

var fg = Math.pow(sg, 1.0 / mn);

var fb = Math.pow(sb, 1.0 / mn);

imgData.data[i \* img.width \* 4 + j \* 4 + 0] = fr;

imgData.data[i \* img.width \* 4 + j \* 4 + 1] = fg;

imgData.data[i \* img.width \* 4 + j \* 4 + 2] = fb;

// imgData.data[i \* img.width \* 4 + i \* 4 + 3] - Альфа-канал (0-255; 0 - прозрачный, 255 - полностью видимый).

}

}

time = performance.now() - time;

t1.value = '' + time;

// Подмена данных о точках.

ctx.putImageData(imgData, 0, 0);

// Подмена изображения.

img.src = canvas.toDataURL("image/png");

});

Приложение Б

(обязательное)

Графическая схема алгоритма распределенной обработки изображений на основе усредняющего среднегеометрического фильтра

using System;

using System.ComponentModel.DataAnnotations;

using System.Net;

using System.Net.Sockets;

namespace ServerImg

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Server.");

var s = 2; // Радиус окрестности пиксела.

Console.Write("Ip: ");

IPAddress ipAddr = IPAddress.Parse(Console.ReadLine());

Console.Write("Port: ");

int port = int.Parse(Console.ReadLine());

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

// Создаем сокет Tcp/Ip

Socket sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

int H = 4;

int W = 1000;

// Назначаем сокет локальной конечной точке и слушаем входящие сокеты

try

{

sListener.Bind(ipEndPoint);

sListener.Listen(10);

// Начинаем слушать соединения

while (true)

{

Console.WriteLine("Wait connection ...");

// Программа приостанавливается, ожидая входящее соединение

Socket handler = sListener.Accept();

// Мы дождались клиента, пытающегося с нами соединиться

byte[] bytes = new byte[W \* H \* 4];

int bytesRec = handler.Receive(bytes);

Array.Resize(ref bytes, bytesRec);

H = bytesRec / (4 \* 1000);

// Показываем данные на консоли

Console.Write("Connected and got.\n");

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ФИЛЬТР ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Копируем массив байт.

byte[] data = (byte[])bytes.Clone();

// Радиус окрестности точки.

// По всем точкам изображения.

for (var i = 0; i < H; i++)

{

for (var j = 0; j < W; j++)

{

var i0 = (i - s < 0 ? 0 : i - s);

var j0 = (j - s < 0 ? 0 : j - s);

var i1 = (i + s >= H ? H - 1 : i + s);

var j1 = (j + s >= W ? W - 1 : j + s);

// Количество точек в окрестности.

double mn = (i1 - i0 + 1) \* (j1 - j0 + 1);

// По окрестности точки (i,j).

double sr = 1.0;

double sg = 1.0;

double sb = 1.0;

// Применяем среднегармонический фильтр.

for (var p = i0; p <= i1; p++)

{

for (var q = j0; q <= j1; q++)

{

double r = data[p \* W \* 4 + q \* 4 + 0];

double g = data[p \* W \* 4 + q \* 4 + 1];

double b = data[p \* W \* 4 + q \* 4 + 2];

sr \*= r;

sg \*= g;

sb \*= b;

}

}

double fr = Math.Pow (sr, 1.0 / mn);

double fg = Math.Pow(sg, 1.0 / mn);

double fb = Math.Pow(sb, 1.0 / mn);

bytes[i \* W \* 4 + j \* 4 + 0] = (byte)fr;

bytes[i \* W \* 4 + j \* 4 + 1] = (byte)fg;

bytes[i \* W \* 4 + j \* 4 + 2] = (byte)fb;

}

}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Отправляем ответ клиенту

handler.Send(bytes);

Console.Write("Sent\n");

handler.Shutdown(SocketShutdown.Both);

handler.Close();

Console.Write("Close\n\n");

}

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

Console.ReadLine();

}

}

}

}