МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Гомельский государственный технический университет

имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01-12 Информационные системы и  
технологии (в игровой индустрии)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Распределенные информационные системы»

на тему: «Сетевое приложение для распределенной обработки изображений с использованием *RESTful* и протокола *TCP*»

Исполнитель: студент группы ИТИ-41

Карпенко Д.Е.

Руководитель: преподаватель

Комраков В.В.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

по защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2024

СОДЕРЖАНИЕ

[Введение 3](#_Toc184110572)

[1 Аналическийобзор средств разработки 5](#_Toc184110573)

[1.1 Выбор языка программирования 5](#_Toc184110574)

[1.2 Обзор протокола TCP/IPиUDP 9](#_Toc184110575)

[1.3 Перечень используемых технологий 12](#_Toc184110576)

[2 Алгоритмический разбор задачи 13](#_Toc184110577)

[2.1 Архитектура приложения 13](#_Toc184110578)

[2.2 Цикл работы распределенного приложения 16](#_Toc184110579)

[2.3 Алгоритм сжатия квантованием 17](#_Toc184110580)

[2.4 Алгоритм распределения нагрузки на кластеры 19](#_Toc184110581)

[3 Распределенное приложение «сжатия изображений квантованием» 21](#_Toc184110582)

[3.1 Главные серверные классы 21](#_Toc184110583)

[3.3 Главные классы блока вычисления 24](#_Toc184110584)

[3.4 Сравнение работы алгоритма сжатия в многопотоке и однопотоке. 25](#_Toc184110585)

[3.5 Апробация приложения 26](#_Toc184110586)

[Заключение 28](#_Toc184110587)

[Список используемой литературы 29](#_Toc184110588)

[Приложение А 30](#_Toc184110589)

# **ВВЕДЕНИЕ**

Современные технологии обработки данных и вычислительных ресурсов требуют более эффективных и гибких решений, особенно когда речь идет о больших объемах данных и сложных вычислительных задачах. Одной из таких задач является обработка изображений, которая активно используется в различных областях, таких как медицинская диагностика, компьютерное зрение, научные исследования и цифровая обработка изображений. В связи с этим возникает необходимость разработки распределенных систем, способных выполнять сложные задачи обработки изображений с минимальными временными затратами.

Распределенные системы позволяют распределить вычислительную нагрузку между несколькими машинами (узлами), что особенно актуально в контексте обработки изображений, где традиционные методы, работающие на одной машине, могут оказаться неэффективными. Такой подход позволяет не только ускорить процесс обработки, но и повысить отказоустойчивость системы, а также гибко масштабировать вычислительные мощности в зависимости от нагрузки.

Одним из ключевых элементов таких систем является сетевое приложение, которое координирует передачу данных и выполнение задач обработки между различными узлами. Для реализации такого приложения важно выбрать подходящие технологии и протоколы. В данной работе будет рассмотрена разработка сетевого приложения для распределенной обработки изображений, использующего архитектурные принципы *REST* и протокол *TCP*.

*REST* (*Representational* *State* *Transfer*) является популярным архитектурным стилем для создания распределенных систем, особенно в случае сетевых приложений. Одной из ключевых особенностей REST является его простота и стандартизация, что делает его легко масштабируемым и совместимым с различными платформами и технологиями. *RESTful* *API* использует стандартные *HTTP* методы (*GET*, *POST*, *PUT*, *DELETE*) для управления ресурсами, обеспечивая простое взаимодействие между клиентами и серверами.

Для передачи данных между узлами системы распределенной обработки критически важно обеспечить надежность и целостность данных. Протокол TCP (*Transmission* *Control* *Protocol*) является одним из ключевых протоколов стека *TCP*/*IP*, обеспечивающим гарантированную доставку данных. TCP контролирует последовательность пакетов и корректность их передачи, что особенно важно при обработке изображений, где даже небольшая ошибка в данных может привести к искажению результата.

*TCP* также позволяет устанавливать надежные соединения между клиентом и сервером, обеспечивая контроль за передачей данных, что делает его предпочтительным выбором для приложений, где критичны точность и надежность, например, в распределенной обработке изображений.

Основной целью разработки сетевого приложения для распределенной обработки изображений является создание гибкой и масштабируемой системы, способной обрабатывать изображения параллельно на нескольких узлах с использованием распределенных вычислительных ресурсов. Приложение должно обеспечивать следующие функции:

– возможность разделения задачи обработки изображений на несколько узлов для параллельного выполнения;

– возможность добавления новых узлов в систему без изменения архитектуры приложения;

– способность системы продолжать работу даже при выходе из строя отдельных узлов;

– с использованием протокола *TCP* для обеспечения корректной передачи изображений и результатов их обработки между узлами;

– использование *RESTful* *API* для взаимодействия между клиентами и серверами, что упрощает интеграцию с внешними системами.

Таким образом, разработка сетевого приложения с использованием *RESTful* архитектуры и протокола *TCP* для распределенной обработки изображений представляет собой комплексную задачу, решающую множество актуальных проблем современных вычислительных систем.

# **1 АНАЛИЧЕСКИЙ****ОБЗОР СРЕДСТВ РАЗРАБОТКИ**

## **1.1 Выбор языка программирования**

*C*# – объектно-ориентированный язык программирования. С момента создания язык *C*# обогатился функциями для поддержки новых рабочих нагру­зок и современными рекомендациями по разработке ПО. В основном *C*# – объектно-ориентированный язык. Вы определяете типы и их поведение.

Вот лишь несколько функций языка C#, которые позволяют создавать надежные и устойчивые приложения. Сборка мусора автоматически освобож­дает память, занятую недостижимыми неиспользуемыми объектами. Типы, до­пускающие значение *null*, обеспечивают защиту от переменных, которые не ссылаются на выделенные объекты. Обработка исключений предоставляет структурированный и расширяемый подход к обнаружению ошибок и восста­новлению после них. Имеются Лямбда-выражения. Синтаксис *LINQ* создает общий шаблон для работы с данными из любого источника. Поддержка языков для асинхронных операций предоставляет синтаксис для создания распреде­ленных систем. В *C*# имеется Единая система типов. Все типы *C*# наследуются от базового класса *object*. Все типы используют общий набор операций, а зна­чения любого типа можно хранить, передавать и обрабатывать схожим обра­зом. *C*# поддерживает как ссылочные типы, так и значимые типы. *C*# позволяет динамически выделять объекты и хранить упрощенные структуры в стеке. *C*# поддерживает универсальные методы и типы, обеспечивающие повышенную безопасность типов и производительность. *C*# предоставляет итераторы, кото­рые позволяют разработчикам классов коллекций определять пользователь­ские варианты поведения для клиентского кода.

*C*# подчеркивает управление версиями, чтобы обеспечить совместимость программ и библиотек с течением времени. Вопросы управления версиями су­щественно повлияли на такие аспекты разработки *C*#, как раздельные модифи­каторы *virtual* и *override*, правила разрешения перегрузки методов и поддержка явного объявления членов интерфейса.

Программы *C*# выполняются в .*NET*, виртуальной системе выполнения, вызывающей общеязыковую среду выполнения (*CLR*) и набор библиотек клас­сов. Название исполняющей среды – «общеязыковая исполняющая среда» (*common language runtime*, *CLR*) – говорит само за себя: это исполняющая среда, которая подходит для разных языков программирования [1, с. 22]. Среда *CLR* – это реализация общеязыковой инфраструктуры языка (*CLI*), являющейся международным стандартом, от корпорации Майкрософт. *CLI* является основой для создания сред выполнения и разработки, в которых языки и библиотеки прозрачно работают друг с другом.

*IL* – не зависящий от процессора машинный язык, разработанный *Microsoft* после консультаций с несколькими коммерческими и академиче­скими организациями, специализирующимися на разработке языков и компи­ляторов. *IL* – язык более высокого уровня в сравнении с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для со­ здания и инициализации объектов, вызова виртуальных методов и непосред­ственного манипулирования элементами массивов[1, c. 29].

Исходный код, написанный на языке *C*# компилируется в [промежуточный язык (*IL*)](https://learn.microsoft.com/ru-ru/dotnet/standard/managed-code), который соответствует спецификациям *CLI*. Код на языке *IL* и ре­сурсы, в том числе растровые изображения и строки, сохраняются в сборке, обычно с расширением .*dll*.

При выполнении программы *C*# сборка загружается в среду *CLR*. Среда *CLR* выполняет *JIT*-компиляцию из кода на языке *IL* в инструкции машинного языка. Среда *CLR* выполняет другие операции такие как, автоматическую сборку мусора, обработку исключений и управление ресурсами. Код, выполня­емый средой *CLR*, иногда называют «управляемым кодом». «Неуправляемый код» компилируется на машинный язык, предназначенный для конкретной платформы.

Обеспечение взаимодействия между языками является ключевой особен­ностью .*NET*. Код *IL*, созданный компилятором *C*#, соответствует специфика­ции общих типов (*CTS*). Код *IL*, созданный из кода на *C*#, может взаимодей­ствовать с кодом, созданным из версий .*NET* для языков *F*#, *Visual* *Basic*, *C*++. Существует более 20 других языков, совместимых с *CTS*. Одна сборка может содержать несколько модулей, написанных на разных языках .*NET*, и все типы могут ссылаться друг на друга, как если бы они были написаны на одном языке.

В дополнение к службам времени выполнения .*NET* также включает рас­ширенные библиотеки. Эти библиотеки поддерживают множество различных рабочих нагрузок. Они упорядочены по пространствам имен, которые предо­ставляют разные полезные возможности: от операций файлового ввода и вывода до управления строками и синтаксического анализа *XML*, от платформ веб-приложений до элементов управления *Windows Forms* и *Windows Presentation*. Обычно приложение *C*# активно используют библиотеку классов .*NET* для решения типовых задач.

В *C*# существуют две разновидности типов ссылочные типы и значимые типы. Переменные значимого типа содержат непосредственно данные, а в пе­ременных ссылочных типов хранятся ссылки на нужные данные, которые име­нуются объектами. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому может случиться так, что операции над одной пере­менной затронут объект, на который ссылается другая переменная.

*Java* – это популярный язык программирования высокого уровня, разработанный компанией *Sun Microsystems* (в настоящее время принадлежит корпорации *Oracle*) в середине 1990-х годов. Он был разработан, чтобы быть простым, безопасным и переносимым, то есть он может работать на любой платформе с установленной виртуальной машиной *Java* (*JVM*).

Данный ЯП(язык программирования) является объектно-ориентированным языком, что означает использование объектов и классов для организации кода. В нем также реализовано автоматическое управление памятью, что снижает риск утечки памяти и облег­чает разработчикам написание надежных и масштабируемых программ.

Одним из ключевых достоинств *Java* является ее платформенная незави­симость. После компиляции *Java*-программы в байткод она может быть запу­щена на любой платформе с установленной *JVM*, включая *Windows*, *Mac* *OS* *X*, *Linux* и многие другие операционные системы. Это делает *Java* популярным выбором для создания приложений, которые должны работать на самых разных устройствах.

*Java* имеет широкий спектр применения, от настольных приложений до веб-приложений корпоративного уровня и разработки мобильных приложений для *Android*. Он также широко используется при разработке внутренних систем и приложений на стороне сервера благодаря своей производительности, мас­штабируемости и безопасности.

*Java* имеет встроенный сборщик мусора, который автоматически управ­ляет памятью, возвращая объекты, на которые программа больше не ссылается. По функциональности сборщик мусора в *Java* похож на сборщик мусора в *C*#, но есть некоторые различия в его работе.

Сборщик мусора в *Java* использует процесс под названием *mark and sweep* для идентификации и возврата мусорных объектов. Этот процесс включает в себя следующие шаги: пометить: сборщик мусора обходит все достижимые объекты и помечает их как живые. Зачистка: сборщик мусора просматривает кучу и разблокировывает все блоки памяти, которые не помечены как живые.

Виртуальная машина *Java* – ‘то программа, которая выполняет байтовый код. Виртуальная машина находится в пакете *JDK* или *JRE*. Еще раз напоми­наю, что для разработки программ нужен пакет *JDK*, потому что с помощью пакета *JRE* можно только вьполнять программы *Java* [5, c. 37].

*JDK* (*Java Development Кit*) – набор инструментов разработки *Java* содер­жащий виртуальную машину, компилятор и другие средства разработки. *SDK* (*SoftwareDevelopment Кit*) – пакет инструментов разработки программного обеспечения. Это устаревшее название пакета *JDK*. На других платформах тер­мин "*SDK*" продолжает широко применяться. *JRE* (*Java Runtime Environment*) – средавыполнения *Java*. Это, собственно, и есть виртуальная машина *Java* [5, с. 44].

*JRE* – это платформенно-независимая виртуальная машина, то есть она предназначена для выполнения *Java*-программ на любой платформе, имеющей совместимую реализацию. *JRE* скрывает различия между разными аппаратными средствами и операционными системами, предоставляя согласованную среду выполнения для *Java*-программ.

*C*++ – это мощный язык программирования, который широко ис­пользуется в различных приложениях, включая системное программирование, разработку игр и высокопроизводительные вычисления.

*C*++ является объектно-ориентированным языком, что означает, что он позволяет разработчикам моделировать концепции и сущности реального мира в виде объектов. Объекты инкапсулируют данные и поведение и могут быть использованы для организации и структурирования кода. Этот язык поддерживает шаблоны, которые являются эффективным инструментом для обобщённого программирования. Они позволяют разработчикам создавать код, способный работать с различными типами данных, исключая необходимость дублирования.

*C*++ дает разработчикам контроль над управлением памятью, что может быть обоюдоострым мечом. С одной стороны, это позволяет эффективно и низкоуровневое манипулировать памятью, но с другой стороны, это может приве­сти к утечкам памяти и другим проблемам при неправильном обращении. *C*++ использует указатели для прямого манипулирования памятью и создания эф­фективных структур данных. Однако указатели могут быть сложны в правиль­ном использовании и могут привести к таким проблемам, как ошибки нулевого указателя и утечки памяти.

*C*++ поддерживает множественное наследование, что означает класс наследуется от нескольких базовых классов. Это может быть полезно для со­здания сложных иерархий объектов, но также может привести к таким пробле­мам, как бриллиантовое наследование и двусмысленность.

В *C*++ нет встроенного сборщика мусора, как в некоторых других языках, таких как *Java* или *C*#. Однако существуют некоторые библиотеки и фрейм­ворки, которые обеспечивают функциональность, подобную сборщику мусора в *C*++. Эти библиотеки не являются частью стандартной библиотеки *C*++, но их можно использовать для автоматического управления памятью и облегче­ния части бремени ручного управления памятью.

Одним из примеров такой библиотеки является сборщик мусора *Boehm-Demers-Weiser* (*BDW*), который представляет собой стороннюю библиотеку, обеспечивающую сборку мусора для программ на *C* и *C*++. Сборщик мусора *BDW* использует стандартный алгоритм сборки мусора, что означает, что он мо­жет собирать объекты, которые явно не помечены как мусор. Это может сде­лать библиотеку более простой в использовании, но также может привести к некоторым накладным расходам и ограничениям производительности. Другим примером является библиотека *C*++ *Garbage* *Collection* (*CGC*), которая явля­ется еще одной сторонней библиотекой, обеспечивающей сборку мусора для программ на *C*++. *CGC* использует гибридный подход, сочетающий подсчет ссылок и сборку мусора, что в некоторых случаях может обеспечить лучшую производительность, чем чистая сборка мусора.

Стоит отметить, что хотя эти библиотеки могут облегчить управление па­мятью в *C*++, они не являются идеальной заменой полноценному сборщику мусора, как в некоторых других языках. Разработчикам *C*++ по-прежнему необходимо знать о проблемах управления памятью и быть осторожными, чтобы избежать утечек памяти, висячих указателей и других проблем, которые могут возникнуть при работе с ручным управлением памятью.

*Python* ­ это высокоуровневый язык программирования, который был разработан Гвидо ван Россумом и впервые выпущен в 1991 году. Он стал одним из самых популярных языков программирования благодаря своей простоте, читабельности и универсальности. *Python* поддерживает несколько парадигм программирования, включая объектно-ориентированное, процедурное и функциональное программирование, что делает его подходящим для решения широкого спектра задач. Динамические языки обычно медленнее, чем статические, но их скорость повышается, поскольку интерпретаторы становятся более оптимизированными. Долгое время динамические языки использовались для коротких программ (сценариев), которые часто предназначались для того, чтобы подготовить данные для обработки более длинными программами, написанными на статических языка[4, с. 36].

*Python* интуитивно понятен и легко читаем, что позволяет разработчикам быстро осваивать язык и сосредотачиваться на решении задач, а не на сложности кода. Это делает *Python* идеальным выбором для начинающих программистов и образовательных учреждений. Данный язык программирования используется в различных областях, включая веб-разработку, научные вычисления, анализ данных, искусственный интеллект, автоматизацию и многое другое. Это делает его одним из наиболее универсальных языков программирования.

*Python* поставляется с обширной стандартной библиотекой, которая включает в себя модули для работы с сетью, файлами, базами данных, текстом и многими другими задачами. Это позволяет разработчикам использовать готовые решения, ускоряя процесс разработки.

*Python* поддерживается на различных операционных системах, включая *Windows*, *macOS* и *Linux*, что позволяет разработчикам создавать кроссплатформенные приложения.

Рассматриваемый язык программирования имеет обширные библиотеки для машинного обучения, такие как *TensorFlow*, *Keras* и *PyTorch*, делают *Python* одним из наиболее популярных языков для разработки моделей искусственного интеллекта и машинного обучения. *Python* часто используется для написания скриптов, которые автоматизируют повседневные задачи, такие как обработка данных, управление файлами и взаимодействие с *API*.

## **1.2 Обзор протокола *TCP*/*IP* и *UDP***

Значительная часть технологии *TCP*/*IP* направлена на решение следующих задач адресации. Задача согласованного использования адресов различного типа. Включает отображение адресов разных типов, например, преобразование сетевого *IP*-адреса в локальный, доменного имени – в *IP*-адрес. Обеспечение уникальности адресов. В зависимости от типа адреса требуется обеспечивать однозначность адресации в пределах компьютера, подсети, корпоративной сети или Интернета. Конфигурирование сетевых интерфейсов и сетевых приложений[2, c. 564].

*TCP*/*IP* имеет многослойную архитектуру, состоящую из нескольких уровней, каждый из которых выполняет свои функции в процессе передачи данных. Эта модульная структура делает протокол гибким и легко масштабируемым.

*TCP* обеспечивает надежную, ориентированную на соединение передачу данных с контрольной суммой и проверкой целостности. *TCP* гарантирует, что данные будут переданы в правильной последовательности и без потерь.

Сетевой уровень отвечает за маршрутизацию пакетов между устройствами через множество сетей. Протокол *IP* (*Internet* *Protocol*) определяет уникальные адреса для устройств *IP*-адреса и управляет маршрутизацией данных через узлы сети.

Канальный уровень взаимодействует с физической сетью (*Ethernet*, *Wi*-*Fi* и другие) и отвечает за передачу пакетов данных на уровне сети. На этом уровне происходит физическая передача данных между узлами. *TCP* является одним из ключевых компонентов стека *TCP*/*IP* и отвечает за управление передачей данных, обеспечивая их надежность и целостность.

Установление соединения *TCP* использует процесс трехстороннего рукопожатия (*three*-*way* *handshake*) для установления соединения между клиентом и сервером. Этот процесс гарантирует, что обе стороны готовы к передаче данных. *TCP* разбивает передаваемые данные на небольшие пакеты, называемые сегментами, и передает их по сети. Это позволяет обрабатывать большие объемы данных более эффективно.

Каждый сегмент данных снабжен уникальным номером последовательности, который позволяет восстановить исходный порядок данных на принимающей стороне. *TCP* также отправляет подтверждения (ACK) для сегментов, которые были успешно доставлены.

*TCP* использует алгоритмы управления потоком (*flow* *control*) и перегрузкой (*congestion* *control*), чтобы предотвратить перегрузку сети и поддерживать эффективную передачу данных.

Данный протокол обеспечивает надежную передачу данных, но из-за дополнительных механизмов контроля и подтверждений имеет более высокие накладные расходы по сравнению с *UDP*.

Адрес IP содержит раздел сети и раздел узла, но форматы этих разделов не одинаковы для всех *IP*-адресов. Число бит адреса, отведенное под обозначение сети, и, соответственно, число бит, отведенное под обозначение узла, изменяются в зависимости от длины префикса адреса. Длина префикса определяется битовой маской адреса[3, c. 46].

*IP* ­ это основа адресации и маршрутизации данных в сети. Существует две версии *IP* *IPv*4 и *IPv*6. *IPv*4 старейшая и наиболее используемая версия *IP*, использующая 32-битные адреса. Однако из-за ограниченного количества доступных адресов (около 4,3 миллиарда) возникла необходимость перехода на *IPv*6. *IPv*6: более современная версия *IP*, использующая 128-битные адреса, что позволяет создавать практически неограниченное количество уникальных адресов. В *IPv*6 встроены улучшенные механизмы безопасности и более эффективные схемы маршрутизации.

Основная функции *IP* это маршрутизация. *IP* управляет маршрутизацией данных от отправителя к получателю через множество сетей. Он использует *IP*-адреса для определения конечного пункта назначения пакетов. Если пакет данных слишком велик для передачи через определенный сегмент сети, *IP* может разделить его на несколько фрагментов и передать по сети, а на принимающей стороне собрать в единое целое.

В отличие от *TCP*, *IP* не гарантирует доставку пакетов. Пакеты могут быть потеряны, повреждены или доставлены не по порядку. Решение этих проблем возлагается на транспортные протоколы, такие как *TCP*.

*TCP* и *IP* работают вместе для обеспечения эффективной передачи данных. *IP* занимается маршрутизацией пакетов между сетями, а *TCP* следит за тем, чтобы пакеты были доставлены в правильном порядке, без ошибок и потерь. *TCP* использует *IP* для перемещения данных через сеть, но берет на себя управление доставкой на уровне соединения. Несмотря на то, что *TCP* и *IP* работают на разных уровнях, их взаимодействие является ключом к надежной передаче данных в сети. Преимущества *TCP*/*IP*:

*– TCP*/*IP* легко масштабируется и поддерживает как малые локальные сети, так и глобальные сети, такие как интернет;

– протоколы *TCP*/*IP* поддерживают работу с различными типами сетей и оборудованием, что делает их универсальными;

– *TCP* обеспечивает проверку ошибок, подтверждение доставки и восстановление порядка данных, что делает его надежным для передачи критически важных данных;

– *TCP*/*IP* является стандартом для большинства современных сетевых технологий, что облегчает интеграцию и взаимодействие между системами.

Дополнительные механизмы, такие как управление потоком и перегрузкой, увеличивают накладные расходы, что делает *TCP* менее эффективным в условиях низкой латентности. Для задач, требующих минимальной задержки, таких как потоковое видео или онлайн-игры, более легковесный протокол *UDP* может оказаться предпочтительнее.

*TCP*/*IP* стал основой современной сетевой инфраструктуры благодаря своей гибкости, надежности и поддержке масштабируемости. Несмотря на появление новых технологий, *TCP*/*IP* остается ключевым стандартом для передачи данных в сетях различного масштаба. Протоколы *TCP* и *IP* отлично дополняют друг друга, обеспечивая как надежность передачи данных, так и эффективную маршрутизацию.

*UDP* протокол используется для «связи без установки соединения» (*connectionless* *communication*)[9]. Один узел сети просто отсылает пакеты, адресуя их другому узлу[9]. Отправитель не знает ничего о том, готов ли получатель к приёму пакетов, и вообще, существует ли этот получатель[9]. Отправитель также не ждёт какого-либо подтверждения о том, что получатель принял предыдущие пакеты[9].

В отличие от *TCP*, *UDP* не обеспечивает надежности передачи и не гарантирует порядок доставки, что делает его быстрым и подходящим для задач, где важна минимальная задержка. Основные характеристики *UDP*:

* бесподключительный протокол;
* нет гарантий доставки;
* нет гарантий порядка;
* высокая скорость передачи;
* малый размер заголовка.

*UDP* не требует установления соединения перед передачей данных. Данные отправляются напрямую, что снижает затраты на соединение. Так же протокол не подтверждает доставку пакетов, поэтому если данные потерялись в процессе передачи, протокол не будет повторно их отправлять. Пакеты (датаграммы) могут приходить в произвольном порядке, так как маршрутизация может различаться для каждого пакета.

Данный минус является одновременно и плюсом, *UDP* более быстрый чем, к примеру *TCP*. Так же заголовок у *UDP* составляет 8 байт, когда у *TCP* от 20 до 40 байт. Структура заголовка *UDP*:

* порт отправителя;
* порт получателя;
* длина заголовка и данных;
* контрольная сумма для проверки целостности данных.

Данные (полезная нагрузка) следуют за заголовком. Максимальный размер пакета *UDP* составляет 65 535 байт (включая заголовок), но на практике он ограничивается размером *MTU* (обычно 1500 байт).

## **1.3 Перечень используемых технологий**

При рассмотрении предлагаемых программных решений применимо задаче, выделяется ряд инструментов, наиболее подходящих к использованию в игровом приложении.

Список предлагаемых инструментов включал в себя:

* сетевые протоколы: *TCP/IP, UDP*;
* языки программирования: *C*# .*NET*, *Java, С++, Python*.

В течении всего аналитического обзора сравнивались языки программиро­вания и графические библиотеки.

В качестве сетевого протокола был выбран *TCP/IP*, для создания сервера, клиента и вычислительных узлов выбран ЯП(язык программирования) *C*#, из-за простоты нахождения информации и интеграции разных компонентов, наличием готовых шаблонов. В качесте протокола передачи выбран *TCP*/*IP*, за его лёгкость использования и га0рантии доставки пакетов данных до адресата.

Список выбранных инструментов включает в себя:

* *TCP/IP* как протокол соединения;
* *C*# .*NET* как язык программирования, на котором будет разработано игровое приложение.

# **2 АЛГОРИТМИЧЕСКИЙ РАЗБОР ЗАДАЧИ**

## **2.1 Архитектура приложения**

Архитектура сетевого приложения состоит из *front*-*end* и *back*-*end*. *Front*-*end* предоставляет пользовательский интерфейс для взаимодействия с программным продуктом. Эта часть выполняет функции отображения и управления, предоставляя пользователю доступ к функциям и данным приложения. *Front*-*end* также может обрабатывать часть бизнес-логики, если это необходимо для улучшения пользовательского опыта. *Back*-*end* отвечает за обработку данных, хранение информации и выполнение бизнес-логики. В этой части приложения находятся серверы и базы данных, а также реализуются функции безопасности и аутентификации. Приложения бывают разных типов:

– клиент-серверные;

– одноранговые;

– облачные;

– микросервисные.

Клиент-серверные приложениях обрабатывает запросы клиентов. Почтовые серверы, веб-серверы, файловые серверы. Клиентское устройство отправляет запросы на сервер и получает от него данные. Есть два вида клиент-серверных архитектур двух уровневая и многоуровневая. Двухуровневая архитектура, клиентское приложение взаимодействует напрямую с сервером, который обрабатывает запросы и отправляет данные обратно клиенту.

Многоуровневая архитектура: здесь между клиентом и сервером могут быть дополнительные промежуточные уровни (например, уровень приложений, базы данных, уровень бизнес-логики), которые помогают распределить задачи обработки и улучшить масштабируемость и производительность приложения. На рисунке 2.1 изображена двух уровневая и многоуровневая.

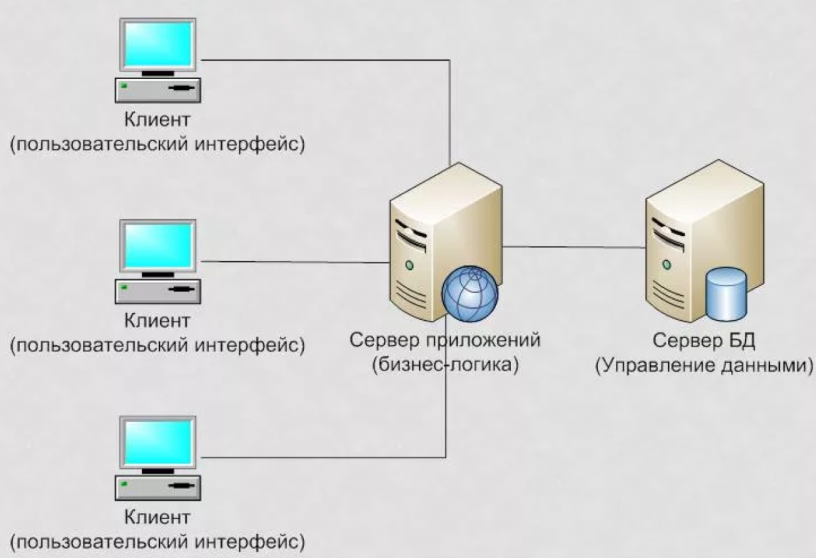


Рисунок 2.1 ­ Двух уровневая структура и много уровневая

Одноранговые (*P*2*P*): в этой модели все узлы сети равноправны и могут как отправлять, так и получать данные. Системы обмена файлами (например, *BitTorrent*), распределенные базы данных. В одноранговой архитектуре все узлы сети равноправны и могут выполнять как функцию отправки, так и функцию получения данных. Такие приложения эффективны для обмена файлами и распределенных вычислений. Примеры: системы обмена файлами (например, *BitTorrent*), распределенные базы данных.

Облачные приложения размещены на удаленных серверах и предоставляются пользователям через интернет. Такие приложения обеспечивают высокую доступность и позволяют работать с данными и файлами независимо от устройства. Примеры: *Google* *Docs*, *Dropbox*, *Office* 365.

Микросервисные приложения делятся на множество мелких сервисов, каждый из которых выполняет свою задачу и взаимодействует с другими сервисами по сети. Эти приложения разделены на множество мелких сервисов, каждый из которых выполняет определенную задачу и взаимодействует с другими сервисами по сети. Каждый микросервис может разрабатываться и разворачиваться независимо, что упрощает масштабирование и улучшает устойчивость к сбоям.

Так же у приложения можно выделить назначение, это так же влияет на архитектуру приложения. Приложения по назначению:

– онлайн редакторы;

– корпоративные системы;

– мессенджеры и средства связи;

– игровые приложения;

– файловые хранилища и обменники.

Способ связи играет важнейшую роль, он определяет каким будет взаимодействие приложения и пользователя. Данный аспект влияет то каким будет *front*-*end* и часть *back*-*end*. Можно выделить три основных способа связи. Синхронный способ требует немедленного ответа от сервера клиенту, асинхронный способ не требует мгновенного отклика от сервера, реального времени и кэшируемые. Реального времени предоставляют информацию происходящую с минимальной задержкой и практически в реальном времени. Кэшируемые или же ойфлайн-доступные это приложения для хранения и обмена файлами.

Клиентская часть приложения отвечает за взаимодействие с пользователем. Она предоставляет интерфейс, через который пользователь может загружать изображения для обработки, выбирать параметры преобразований (например, фильтры или методы улучшения изображения) и получать готовый результат. Реализация клиентской части может варьироваться. Это может быть веб-приложение, доступное через браузер, либо мобильное приложение. Для передачи данных на сервер может использовать протоколы, такие как *REST* *API*, *WebSocket*, *HTTP* что обеспечивает идентичность форматов принимаемых и отправляемых данных.

Центральным узлом архитектуры выступает серверная часть, которая обрабатывает запросы клиентов, управляет хранением данных и выполняет основные операции над изображениями. После получения запроса сервер принимает изображение и передаёт его в модуль обработки. Сервер также отслеживает параметры, указанные пользователем, и в зависимости от них применяет необходимые преобразования. Обработка изображений осуществляется с использованием специализированных библиотек и инструментов.

Важной составляющей архитектуры является модуль обработки изображений, который непосредственно выполняет операции преобразования. Он поддерживает как простые задачи, например изменение размеров или наложение фильтров, так и сложные, такие как распознавание объектов, стилизация изображения или его улучшение с использованием нейросетей. Для ускорения вычислений модуль может использовать аппаратное ускорение с помощью *GPU*, что особенно важно при обработке больших изображений или применении вычислительно затратных алгоритмов.

На рисунке 2.2 изображена архитектура разрабатываемого приложения. Архитектура клиент-серверного приложения для обработки изображений разработана таким образом, чтобы обеспечить оптимальное взаимодействие между клиентами и сервером, а также добиться высокой производительности и минимальных задержек при выполнении операций над изображениями. Эта архитектура включает несколько ключевых компонентов, которые совместно обеспечивают её функциональность.

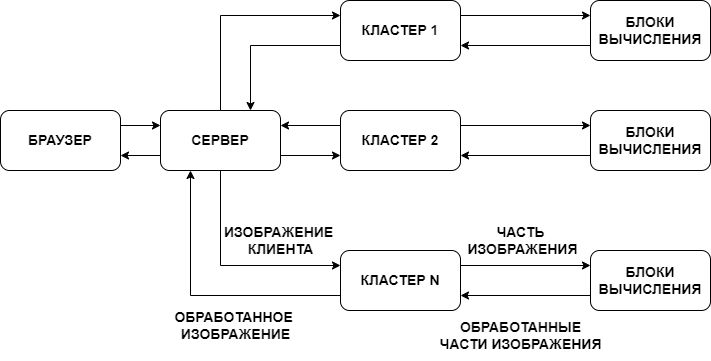


Рисунок 2.2 ­ Архитектура разрабатываемого приложения

Рассмотрим основные компоненты этой архитектуры и этапы работы приложения, которое включает обработку изображения по запросу клиента.

Обработка изображения происходит путем сжатия изображения путем квантования. Назначение данного приложения можно выделить из выше упомянутых предложений, а именно это онлайн-редактор с синхронным типом связи.

## **2.2 Цикл работы распределенного приложения**

Цикл работы распределенного ­ это последовательность шагов, через которые проходят компоненты приложения для выполнения запросов пользователя и обеспечения обмена данными между узлами системы. На рисунке 2.3 изображен цикл работы.

Этот цикл можно разбить на несколько основных этапов:

– отправка запроса клиентом;

– передача запроса в узлы;

– обработка запроса сервером (или узлом);

– координация и синхронизация;

– сбор и агрегирование результатов;

– передача ответа клиенту;

– отображение данных пользователю.



Рисунок 2.3 ­ Цикл работы

Пользователь инициирует действие, например, отправку сообщения, просмотр видео или выполнение сложного вычисления, с помощью интерфейса приложения. Клиентское приложение формирует запрос к системе, включающий:

– тип запроса;

– идентификаторы сессии;

– параметры запроса.

После формирования запроса клиентское приложение передает его по сети к распределенной системе. Запрос может проходить через несколько промежуточных узлов, таких как балансировщики нагрузки, которые распределяют запросы между доступными узлами для оптимального использования ресурсов. При сложных операциях узел может разделить запрос на несколько параллельных задач для более быстрой обработки. Если данные распределены между узлами (шардирование или репликация), узлы синхронизируют результаты обработки для поддержания согласованности.

Когда узлы завершили свои части задачи, необходимо собрать все результаты для передачи клиенту. Если данные запрашиваются с нескольких узлов, агрегируются частичные результаты. Данные, полученные от нескольких источников, объединяются в единый ответ. Если часть узлов не смогла выполнить операцию, система может предпринять меры для восстановления данных или уведомить клиента о частичной неудаче. После обработки запроса сервер формирует ответ для клиента:

– форматирование данных;

– код состояния;

– отправка ответа.

На клиенте данные, полученные от сервера, обрабатываются и отображаются пользователю.

## **2.3 Алгоритм сжатия квантованием**

Алгоритм сжатия изображений путем квантования является эффективным методом, который позволяет уменьшить объем данных, представляющих изображение, за счет сокращения числа цветов или оттенков, используемых для кодирования пикселей. Основная идея квантования – это представить цвета более грубо, что позволяет снизить объем передаваемых данных, сохраняя при этом приемлемое качество. Этот метод особенно популярен в сжатии изображений, например, в формате *JPEG*. Этапы сжатия квантованием:

– преобразование цвета изображения;

– разделение изображения на блоки;

– дискретное косинусное преобразование (ДКП);

– квантование коэффициентов, на рисунке 2.4 пример матрицы коэффицентов;

– удаление нулевых коэффициентов;

– обратное ДКП;

– сборка итогового изображения.

Изображение выглядит как текст, Шрифт, белый, число

Автоматически созданное описание

Рисунок 2.4 ­ Матрица квантования

Часто изображения хранятся в формате *RGB*, где каждый цвет кодируется тремя значениями (красный, зеленый, синий). Однако для квантования иногда удобнее работать с другим цветовыми моделями, такими как *YCbCr*, где разделяются яркостная и цветовая составляющие. Например, *Y* обозначает яркость (светлоту), а *Cb* и *Cr* ­ цветовую разницу. Такое разделение позволяет проще обрабатывать и сжимать компоненты изображения, так как яркость воспринимается человеческим глазом лучше, чем цветовые различия.

Изображение разбивается на маленькие блоки, например, 8x8 или 16x16 пикселей, и каждый блок обрабатывается отдельно. Это облегчает работу алгоритма, так как мелкие детали, такие как шум, можно удалить в каждом блоке без значительных потерь. Такое разделение позволяет также легче применять методы частотного анализа, как ДКП, к каждому блоку.

Дискретное косинусное преобразование – это преобразование Фурье, аналогичное дискретному преобразованию Фурье (ДПФ), но использующее только действительные числа[8]. ДКП применяется к каждому блоку, переводя его из пространственного представления в частотное. ДКП преобразует изображение так, что большинство значений, передающих крупные детали, находятся в начале матрицы, а мелкие детали и шум переходят в конец. На рисунке 2.4 изображена матрица квантования.

Данный метод позволяет отделить высокочастотные (детали, текстуры и шум) компоненты изображения от низкочастотных (основная структура, фон), которые более значимы для восприятия. Формула(2.1) ДКП задана данной функцией:

– это коэффициенты дискретного косинусного преобразования для индекса *k*. Эти коэффициенты представляют собой спектральные компоненты преобразованного сигнала.

– это символ суммы, указывающий на то, что необходимо просуммировать все элементы от *n* = 0, до *n* = *N* - 1, где *N* – длина исходного сигнала.

– это значение входного сигнала в точке *n*. Это данные, которые мы преобразуем с помощью *DCT*.

– это косинусная функция, которая зависит от индекса n, индекса преобразования *k* и длины сигнала *N*. Эта часть формулы отвечает за преобразование данных в спектральные компоненты на основе частот.

*K* = 0,1,2,…, *N*-1 – индексы частот, для которых вычисляются коэффициенты . Эти индексы могут быть разными в зависимости от числа коэффициентов в результате преобразования.

Частотное представление удобно для квантования, так как можно отбросить менее важные компоненты (высокочастотные) с меньшими визуальными потерями.

После ДКП получается матрица коэффициентов для каждого блока изображения, где каждый коэффициент имеет различный уровень значимости для визуального качества.

Каждый коэффициент матрицы делится на определенное значение и округляется. Чем выше значение, на которое делится коэффициент, тем сильнее «сжимается» соответствующая деталь.

Более низкочастотные компоненты квантуются с меньшим шагом, чтобы сохранить основные элементы изображения. Высокочастотные компоненты, в которых меньше важной информации, квантуются с более крупными шагами, что позволяет «отбросить» больше данных, не влияя значительно на качество.

Таблица квантования – это массив чисел, который задает, каким образом будут квантоваться различные компоненты. Чем более агрессивная таблица квантования, тем выше сжатие, но тем больше потери качества.

После квантования многие высокочастотные коэффициенты могут стать нулевыми. Эти нулевые значения можно просто игнорировать, что позволяет существенно уменьшить объем данных. После квантования и сжатия данные преобразуются обратно с помощью обратного ДКП, восстанавливая каждый блок изображения. Этот процесс воссоздает пиксели изображения, но с потерей некоторых мелких деталей, которые были отброшены при квантовании. После обратного преобразования блоки собираются обратно в единое изображение. Если изображение изначально было преобразовано в другую цветовую модель, его можно вернуть в исходный формат *RGB*.

## **2.4 Алгоритм распределения нагрузки на кластеры**

Распределение нагрузки на кластеры – это важный аспект оптимизации вычислительных ресурсов, который позволяет эффективно использовать вычислительные мощности системы и минимизировать время отклика. Оптимальный алгоритм распределения зависит от задач, архитектуры системы и особенностей нагрузки. Рассмотрим несколько распространённых подходов к распределению нагрузки с объяснением их работы:

– *Round* *Robin* (Циклический обход);

– *Least* *Connections* (Минимум соединений);

– *Least* *Load* (Минимальная загрузка);

– *Weighted* *Round* *Robin* (Взвешенный *Round* *Robin*);

– *Adaptive* *Load* *Balancing* (Адаптивное балансирование нагрузки);

– *Consistent* *Hashing* (Консистентное хеширование);

– *Machine* *Learning*-*Based* *Load* *Balancing* (Балансировка на основе машинного обучения);

Алгоритм *Round* *Robin* – алгоритм распределения нескольких задач или/и нагрузки распределённой вычислительной системы — методом перебора и упорядочения её элементов по круговому циклу[6]. Это простой метод, который подходит, если кластеры обладают одинаковой производительностью и нагрузка на них однородна.

*Least* *Connections* (Наименьшее количество соединений) – это алгоритм балансировки нагрузки, который призван выбирать сервер с наименьшим активным соединением из числа доступных серверов[8]. Это позволяет учитывать загруженность кластеров и направлять задачи туда, где нагрузка минимальна.

Алгоритм *Least* *Load* направляет задачи на кластер с минимальной загрузкой. Загрузка может измеряться в процентах от общего количества ресурсов (*CPU*, память, сеть и т.д.), доступных на кластере.

В алгоритме *Weighted* *Round* *Robin* данном алгоритме каждому кластеру присваивается вес в зависимости от его производительности. Кластеры с более высокой производительностью получают больше задач.

Адаптивное балансирование – это динамический метод, который учитывает не только загрузку, но и другие метрики производительности (например, отклик или задержку). Задачи перенаправляются на менее загруженные кластеры в зависимости от текущей ситуации.

Этот алгоритм позволяет распределять задачи между кластерами на основе хеширования, что хорошо подходит для распределённых систем хранения данных. При добавлении/удалении кластеров алгоритм минимизирует изменения распределения.

*Machine* *Learning*-*Based* *Load* *Balancing* этот метод использует модели машинного обучения для прогнозирования нагрузки и распределения задач с учётом истории и текущих данных о производительности кластеров.

Для данного проекта оптимальным решением будет использовать *Round* *Robin*. На рисунке 2.5 изображен алгоритм *Round* *Robin* для данного проекта.

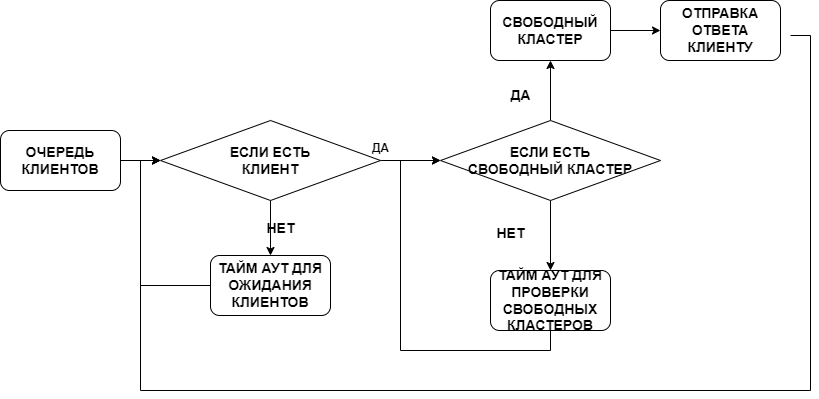


Рисунок 2.5 ­ Алгоритм работы *Round* *Robin*

*Round* *Robin* будет последовательно распределять задачи сжатия между кластерами, при этом задачи, которые не могут быть сразу обработаны, будут помещаться в очередь. Очередь поможет хранить поступающие задачи до тех пор, пока они не смогут быть обработаны, что повышает пропускную способность системы и минимизирует простои.

# **3 РАСПРЕДЕЛЕННОЕ ПРИЛОЖЕНИЕ «СЖАТИЯ ИЗОБРАЖЕНИЙ КВАНТОВАНИЕМ»**

## **3.1 Главные серверные классы**

Сервер является сердцем в любом многопользовательском приложении, он обеспечивает обработку клиентов и их запросов. В данном случае сервер принимает запросы клиентов на сжатие картинок и отправляет их на кластеры.

Рассматриваемые классы:

* *HTTPServer*;
* *FileHandler*;
* *ClasterManager*;
* *Claster*.

*HTTPServer* – класс представляет *HTTP*-сервер, способный обрабатывать запросы клиентов, выполнять загрузку и передачу файлов. Поддерживает работу с *HTTP*-методами *GET* и *POST* и реализует асинхронную обработку соединений.

Методы класса:

* *HTTPServer*(*int* *port*);
* *Start*();
* *HandleClient*(*TcpClient* *client*).

*HTTPServer*(*int* *port*) – конструктор, инициализирующий сервер на заданном порту. *Start*() – метод запускает сервер, начинает прослушивание *TCP*-соединений и обрабатывает входящие запросы. *HandleClient*(*TcpClient* *client*) – асинхронно обрабатывает соединение с клиентом, интерпретирует *HTTP*-запросы и формирует соответствующий *HTTP*-ответ.

*FileHandler* – класс предоставляет функциональность для обработки файлов, загружаемых на сервер через *HTTP*, а также для отправки файлов и *HTML*-форм клиенту. Основное применение включает загрузку, чтение, и извлечение данных из *HTTP*-запросов.

Методы класс:

* *HandleUpload*(*NetworkStream* *stream*);
* *ExtractBoundaryFromHeaders*(*string* *headers*);
* *SendFile*(*NetworkStream* *stream*, *string* *path*);
* *GetHtmlForm*();
* *GetContentType*(*string* *filePath*).

*HandleUpload*(*NetworkStream* *stream*) – асинхронно обрабатывает загружаемый файл, читая поток данных, извлекая содержимое файла и возвращая его в виде массива байт. *ExtractBoundaryFromHeaders*(*string* *headers*) – извлекает границу (*boundary*) из *HTTP*-заголовков запроса. *SendFile*(*NetworkStream* *stream*, *string* *path*) – отправляет файл клиенту через *NetworkStream*, устанавливая корректный *HTTP*-ответ и заголовки, включая *Content*-*Type*. *GetHtmlForm*() – возвращает *HTML*-форму для загрузки файлов с использованием метода *POST*. *GetContentType*(*string* *filePath*) – определяет *MIME*-тип файла на основе его расширения.

*ClasterManager* – класс отвечает за управление кластерами, обеспечивая обработку подключений от кластерных узлов и отправку данных для обработки. Реализует серверный сокет для приема соединений и взаимодействует с объектами типа *Claster*.

Методы класса:

* *ClasterManager*();
* *Send*(*byte*[] *data*);
* *ClasterCycle*().

*ClasterManager*() – конструктор, инициализирующий серверный сокет и запускающий цикл обработки кластеров. *Send*(*byte*[] *data*) – асинхронно отправляет данные свободному кластерному узлу для обработки и возвращает результат. *ClasterCycle*() – запускает цикл ожидания подключений от новых кластерных узлов и добавляет их в список \_*clasters*.

*Claster* – класс представляет кластерный узел, обрабатывающий данные, отправленные сервером, и возвращающий результат. Поддерживает асинхронное взаимодействие с сервером с использованием сокетов.

Методы класса:

* *Claster*(*Socket* *clasterSocket*);
* *SendData*(*byte*[] *bytes*);
* *ClasterRecive*();
* *SendToClasterData*(*byte*[] *dataSend*).

*Claster*(*Socket* *clasterSocket*) – конструктор, инициализирующий кластерный узел и запускающий поток для получения данных. *SendData*(*byte*[] *bytes*) – асинхронно отправляет данные в кластер и ожидает результата. *ClasterRecive*() – Получает данные от кластера и уведомляет ожидающие потоки. *SendToClasterData*(*byte*[] *dataSend*) – отправляет данные в кластер, включая их размер.

**3.2 Главные классы кластера**

Кластер является ключевым компонентом распределённого приложения, обеспечивая его масштабируемость, производительность и отказоустойчивость. Кластер позволяет распределять задачи между несколькими узлами, что снижает нагрузку на отдельные компоненты системы. Это обеспечивает равномерное использование ресурсов и улучшает общую производительность. Главные классы кластера:

* *ClusterServer*;
* *ClasterUnit*;
* *Pocket*.

*ServerCluster* – класс отвечает за координацию взаимодействия между сервером и вычислительными блоками (*ClusterUnit*) в рамках кластерной архитектуры. Основные задачи: приём данных от сервера, распределение их между вычислительными блоками, сбор результатов и отправка итоговых данных обратно на сервер.

Методы класса:

* *ServerCluster*();
* *CycleOfServer*();
* *DesideTask*(*MemoryStream* *memoryStream*);
* *SendImageToUnits*();
* *RecivePocketsFromUnits*(*byte*[] *dataSend*);
* *AddPocket*(*Pocket* *pocket*);
* *ClasterListener*().

*ServerCluster*() – конструктор, инициализирующий сокеты, подключение к серверу и запуск потоков для обработки запросов. *CycleOfServer*() – метод получает данные от сервера, разделяет задачи и распределяет их по вычислительным блокам. *DesideTask*(*MemoryStream* *memoryStream*) – метод делит изображение на части и формирует пакеты для вычислительных блоков. *SendImageToUnits*() – метод отправляет сформированные пакеты с изображениями вычислительным блокам. *RecivePocketsFromUnits*(*byte*[] *dataSend*) – метод передаёт объединённые результаты работы вычислительных блоков обратно на сервер. *AddPocket*(*Pocket* *pocket*) – метод добавляет пакет с результатами от вычислительного блока в список. Если получены все результаты, объединяет их в единое изображение и отправляет серверу. *ClasterListener*() – метод ожидает подключения новых вычислительных блоков и добавляет их в кластер.

*ClusterUnit* – класс представляет отдельный вычислительный блок (*unit*) в составе кластера. Он обрабатывает отправляемые сервером данные, выполняет вычисления и возвращает результаты.

Методы класса:

– *ClusterUnit*(*Socket socket*, *ServerCluster serverCluster*);

* *CileOfListen*();
* *SendDataToUnit*(*Pocket* *pocket*).

*ClusterUnit*(*Socket* *socket*, *ServerCluster* *serverCluster*) – конструктор, инициализирующий вычислительный блок и запускающий поток для приёма данных. *CileOfListen*() – метод обрабатывает данные, возвращаемые вычислительным блоком, и отправляет их на сервер кластера. *SendDataToUnit*(*Pocket* *pocket*) – метод отправляет данные в вычислительный блок для обработки.

*Pocket* – класс представляет собой контейнер для хранения данных, используемый в распределённых вычислениях. Каждый объект *Pocket* содержит часть сообщения, которая может быть передана между компонентами кластера. Данный метод не имеет методов, он имеет только два конструктора. Конструкторы класса:

* *Pocket*();
* *Pocket*(*int* *partOfMesage*, *byte*[] *partOfData*).

*Pocket*() – cоздаёт пустой объект *Pocket*. Удобен для десериализации, когда поля объекта будут заполнены позже. *Pocket*(*int* *partOfMesage*, *byte*[] *partOfData*) – cоздаёт объект *Pocket* с заданным номером части и её данными. *partOfMesage* – номер или идентификатор части сообщения. *partOfData* – данные в виде массива байт, относящиеся к этой части. Позволяет инициализировать объект с данными для передачи между компонентами кластера.

## **3.3 Главные классы блока вычисления**

Блок вычисления является непосредственной единицей вычисления в кластере. Подход с блоками позволяет делать масштабируемые, отказоустойчивые приложения. Основные классы блока вычисления:

* *Unit*;
* *Pocket*;
* *CompressIMG*.

*Unit*() – конструктор, инициализирующий узел, подключающий его к кластеру и запускающий процесс приёма данных. *Listen*() – метод который асинхронно обрабатывает входящие данные, выполняет сжатие и отправляет результаты обратно в кластер. *SendLengthAndData*(*byte*[] *dataSend*) – метод отправляет размер данных и их содержимое в кластер.

*CompressIMG* – класс реализует алгоритм сжатия изображений на основе дискретного косинусного преобразования (*DCT*) с использованием квантования и обратного преобразования. Он разделяет изображение на блоки размером 8×8, применяет преобразования и восстанавливает данные.

Методы класса:

* *CompressImageAsync*(*byte*[] *buffer*);
* *ProcessBlock*(*byte*[] *pixelData*, *int* *startX*, *int* *startY*, *int* *stride*, *int* *bytesPerPixel*);
* *AddPadding*(*Bitmap* *original*);
* *QuantizeBlock*(*double*[,] *block*);
* *DequantizeBlock*(int[,] *quantizedBlock*);
* *DCT*(*double*[,] *block*);
* *IDCT*(*double*[,] *block*).

*CompressImageAsync*(*byte*[] *buffer*) – асинхронно выполняет сжатие изображения, возвращая сжатое изображение в виде массива байт. *ProcessBlock*(*byte*[] *pixelData*, *int* *startX*, *int* *startY*, *int* *stride*, *int* *bytesPerPixel*) – метод обрабатывает блок изображения (8×8), выполняя *DCT*, квантование и обратное преобразование. *AddPadding*(*Bitmap* *original*) – метод Добавляет отступы к изображению для выравнивания его размеров до кратных 8. *QuantizeBlock*(*double*[,] *block*) – метод квантует блок данных с использованием матрицы квантования. *DequantizeBlock*(*int*[,] *quantizedBlock*) – метод деквантует блок данных, восстанавливая исходные значения. *DCT*(*double*[,] *block*) – метод выполняет дискретное косинусное преобразование для блока. *IDCT*(*double*[,] *block*) – метод выполняет обратное дискретное косинусное преобразование для блока.

## **3.4 Сравнение работы алгоритма сжатия в многопотоке и однопотоке.**

Целью данного сравнения является скорость обработки изображения. В тесте будут использоваться малые и большие изображения. В качестве малого изображения будет использоваться картинка с такими характеристиками. Ширина 709 пикселей высота 109 пикселей. В качестве большого изображения будет использоваться картинка с такими характеристиками. Ширина 4096 пикселя высота 6400 пикселей. Для каждого режима обработки, однопоточного и многопоточного, будет 5 прогонов. Результаты будут записаны в две таблице. На рисунке 3.1 изображены тестовые картинки.

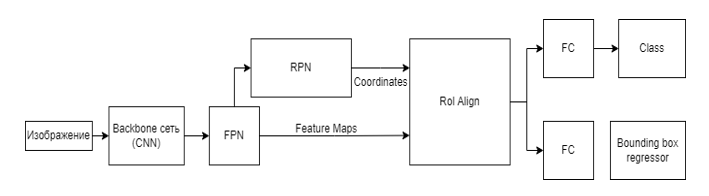


Рисунок 3.1 – Изображения для теста производительности

В левой части изображения 3.1 расположена картинка высокого разрешения в правой низкого. Как можно увидеть из таблицы 3.1, многопоток медленнее чем однопоток. Это является закономерным следствием из-за малой нагрузки. Процессор больше тратит времени на создание потоков, чем на выполнение задачи. Так же ему приходится выделять время выполнения для потоков, что тоже дает задержки.

Таблица 3.1 – Время сжатия для малого разрешения изображения

|  |  |
| --- | --- |
| Однопоток | Многопоток |
| 0.8297566 | 1.0863765 |
| 0.7777555 | 1.3783560 |
| 0.7601087 | 1.4804316 |
| 0.7450012 | 1.7064037 |
| 0.7324589 | 1.5132574 |

В таблице 3.2 напротив, многопоток уже выбирается в лидеры. Хоть и в относительных показателях это на уровне с разницей много потока. Но в абсолютных это примерно 40 секунд. Что говорит о том что на больших разрешениях многопоток существенно ускоряет процесс обработки изображения.

Таблица 3.2 – Время сжатия для большого разрешения изображения

|  |  |
| --- | --- |
| Однопоток | Многопоток |
| 2:24.2671009 | 42.1933648 |
| 2:19.0601231 | 37.5231847 |
| 2:29.2021233 | 39.8217463 |
| 2:21.6870009 | 44.4789123 |
| 2:18.0091929 | 42.6894729 |

После двух тестов можно подвести итог сравнения. Многопоток более оптимален в задаче сжатия чем однопоток, т.к. время ожидания у много потока больше только на малых изображениях. Разница составляет около 0.2 секунды, что в рамках сети не будет ощущаться как большая задержка. Разница на целых 1.5 минуты это уже более ощутима чем на 0.2 секунды.

## **3.5 Апробация приложения**

На рисунке 3.2 изображен *HTML* страница, на которой пользователь может выбрать изображение для сжатия. Данная страница состоит из двух кнопок. Первая кнопка позволяет выбрать изображение для сжатия, вторая необходима для отправки на сервер изображения.

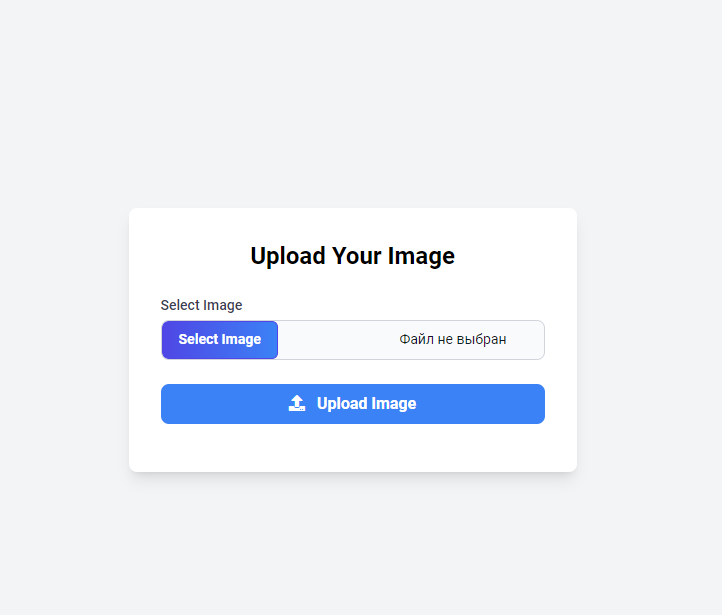


Рисунок 3.2 – *HTML* страница пользователя

На рисунке 3.3 изображено часть сжатого изображения и не сжатого. В качестве примера будет использовано малое и большое изображение. Как видно есть небольшие артефакты при сжатии. Т.к. само сжимаемое изображение было не большого разрешения и с не большими артефактами.

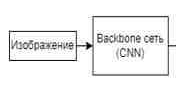


Рисунок 3.3 – Сжатое изображение А и не сжатое изображение Б

На рисунке 3.4 изображена часть изображения более высокого разрешения и не сжатое изображение. Если рассматривать картинку, не приближая отдельные части, то понять, где сжатая картинка, а где не сжатая будет сложно.

****

Рисунок 3.4 – Сжатая картинка A и не сжатая картинка Б высокого разрешения

Обратите внимание на линии они стали более ступенчатыми – там можно заметить небольшие квадратики, которые являются результатом сжатия с использованием дискретного косинусного преобразования.

Первоначальный размер изображения с низким разрешением составлял 16 килобайт, а после сжатия уменьшился до 12 килобайт. Для изображения с высоким разрешением исходный вес был 1.18 мегабайта, а стал 0.853 мегабайта. Показатель сжатия можно регулировать, изменяя степень сжатия через более интенсивную матрицу квантования. Однако, при этом возрастёт количество артефактов на выходном изображении, что негативно скажется на его качестве.

# **ЗАКЛЮЧЕНИЕ**

В результате проделанной работы было разработано клиент-серверное приложение для сжатия изображений, использующее протокол *HTTP* для взаимодействия между клиентом и сервером. Приложение было создано на языке *C*# в среде разработки *Visual* *Studio*, а для отображения информации использовался браузер. Важным компонентом приложения являются кластеры, которые напрямую участвуют в процессе сжатия изображений, что позволяет распределять нагрузку равномерно и повышает масштабируемость приложения.

В процессе разработки был проведен аналитический обзор инструментов и технологий, необходимых для реализации данного приложения. В частности, было уделено внимание выбору подходящего алгоритма сжатия изображений, а также инфраструктуре, обеспечивающей эффективное взаимодействие между сервером и клиентом через протокол *HTTP*.

Для обеспечения масштабируемости и повышения производительности были внедрены кластеры, которые позволяют эффективно обрабатывать и сжимать изображения, распределяя нагрузку между несколькими узлами. Это решение способствует ускорению процесса сжатия и уменьшению времени отклика приложения.

Архитектура приложения была спроектирована с учетом принципов объектно-ориентированного программирования (ООП). Для реализации сетевого взаимодействия между клиентом и сервером использовались сокеты, что обеспечивало надежную и быструю передачу данных. Важным аспектом разработки стало обеспечение удобства работы с интерфейсом через браузер, что позволяет пользователям легко загружать и просматривать сжатые изображения.

Данное приложение предоставляет удобное средство для сжатия изображений с использованием современных технологий и подходов, таких как кластеризация и клиент-серверная архитектура, и рассчитано на широкую аудиторию пользователей, которым важно быстро и эффективно уменьшить размер изображений без значительной потери качества.

# **СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Frame work 2.0 на языке C#. Мастер класс / Дж. Рихтер. – СПб. Питер , 2008. – 656 с.

2. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 3-е издание. – СПб.: Питер, 2006. – 958 с.

3. Крэйг Хант. TCP/IP. Сетевое администрирование, 3-е издание. – Пер. с англ. – СПб: Символ Плюс, 2004. – 816 с.

4. Любанович Б., Простой Python. Современный стиль программиро-вания. / Любанович, Б. – СПб.: Питер, 2016. – 480 с

5. Берд, Барри. Jаvа для чайников / Берд Барри. – 5-е изд. – 2013. – 368 с.

6. Round-robin (алгоритм) [Электронный ресурс]. – Режим доступа: <https://ru.wikipedia.org/wiki/Round-robin_(алгоритм)> – Дата доступа: 12.10.2024

7. Сравнение алгоритмов балансировки нагрузки: Round Robin vs. Least Connections vs. IP Hash [Электронный ресурс]. Режим доступа: <https://habr.com/ru/companies/otus/articles/770248/> – Дата доступа: 12.10.2024

8. Дискретное косинусное преобразование [Электронный ресурс]. Режим доступа: <https://translated.turbopages.org/proxy_u/en-ru.ru.e9c1cc56-675022c4-54bb5944-74722d776562/https/en.wikipedia.org/wiki/Discrete_cosine_transform> – Дата доступа: 25.09.2024

9. TCP и UDP, или Два столпа Интернета [Электронный ресурс]. Режим доступа: <https://habr.com/ru/articles/711578/> – Дата доступа: 25.09.2024

# **ПРИЛОЖЕНИЕ А**

Листинг программы

(обязательное)

**Файл Program.cs:**

using HTTP;

namespace Program

{

class Program

{

public static void Main(string[] args)

{

HTTPServer server = new HTTPServer(4000);

server.Start();

}

}

}

**Файл HTTPServer.cs:**

using System.Net;

using System.Net.Sockets;

namespace HTTP

{

public class HTTPServer

{

private TcpListener listener;

private bool isRunning = false;

private Claster.ClasterManager manager;

private UserManager.UserManager userManager;

public HTTPServer(int port)

{

listener = new TcpListener(IPAddress.Any, port);

manager = new Claster.ClasterManager();

userManager = new UserManager.UserManager(1);

}

public void Start()

{

isRunning = true;

listener.Start();

//\_listener.Start();

Console.WriteLine($"Server started on port {((IPEndPoint)listener.LocalEndpoint).Port}");

while (isRunning)

{

TcpClient client = listener.AcceptTcpClient();

client.SendBufferSize = 65536;

client.ReceiveBufferSize = 65536;

client.Client.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.KeepAlive, true);

userManager.AddUser(client);

Thread thread = new Thread(() => HandleClient(client));

thread.Start();

}

}

private async void HandleClient(TcpClient client)

{

if (client.Connected)

{

using (NetworkStream stream = client.GetStream())

{

// Чтение и обработка запроса

HTTPRequest request = HTTPRequest.Parse(stream);

HTTPResponse response = new HTTPResponse();

if (request.Method == "GET")

{

if (request.Path == "/")

{

response.SendHtml(stream, FileHandler.GetHtmlForm());

userManager.RemoveUser(client);

client.Close();

}

else if (request.Path.StartsWith("/uploads/"))

{

FileHandler.SendFile(stream, request.Path);

userManager.RemoveUser(client);

client.Close();

}

else

{

response.SendError(stream, "404 Not Found");

userManager.RemoveUser(client);

client.Close();

}

}

else if (request.Method == "POST" && request.Path == "/")

{

var buffer = await FileHandler.HandleUpload(request);

var buffer\_1 = userManager.SendData(manager, client, buffer).Result;

if (buffer\_1 != null)

{

response.SendFile(stream, buffer\_1, FileHandler.GetHtmlForm());

client.Close();

}

else

{

response.SendError(stream, "Query is full, please try anywere.");

client.Close();

}

}

}

}

}

}

}

**Файл HTTPResponse.cs:**

using System.Net.Sockets;

using System.Text;

namespace HTTP

{

public class HTTPResponse

{

public void SendHtml(NetworkStream stream, string content)

{

string response = "HTTP/1.1 200 OK\r\n" +

"Content-Type: text/html\r\n" +

"Content-Length: " + content.Length + "\r\n" +

"\r\n" +

content;

byte[] buffer = Encoding.ASCII.GetBytes(response);

stream.Write(buffer, 0, buffer.Length);

}

public async Task SendHtmlAsync(NetworkStream stream, string content)

{

// Преобразуем строку HTML в байты с использованием кодировки UTF-8

byte[] contentBytes = Encoding.UTF8.GetBytes(content);

// Создаем HTTP-ответ с заголовками

string responseHeaders = "HTTP/1.1 200 OK\r\n" +

"Content-Type: text/html; charset=utf-8\r\n" +

"Content-Length: " + contentBytes.Length + "\r\n" +

"\r\n";

// Преобразуем заголовки в байты

byte[] headerBytes = Encoding.ASCII.GetBytes(responseHeaders);

// Отправляем заголовки

await stream.WriteAsync(headerBytes, 0, headerBytes.Length);

// Отправляем контент (сам HTML)

await stream.WriteAsync(contentBytes, 0, contentBytes.Length);

}

public void SendFile(NetworkStream stream, byte[] fileBytes, string contentType)

{

string headers = "HTTP/1.1 200 OK\r\n" +

"Content-Type: " + contentType + "\r\n" +

"Content-Length: " + fileBytes.Length + "\r\n" +

"\r\n";

byte[] headerBytes = Encoding.ASCII.GetBytes(headers);

stream.Write(headerBytes, 0, headerBytes.Length);

stream.Write(fileBytes, 0, fileBytes.Length);

}

public void SendError(NetworkStream stream, string errorMessage)

{

string response = "HTTP/1.1 404 Not Found\r\n" +

"Content-Type: text/html\r\n" +

"Content-Length: " + errorMessage.Length + "\r\n" +

"\r\n" +

errorMessage;

byte[] buffer = Encoding.ASCII.GetBytes(response);

stream.Write(buffer, 0, buffer.Length);

}

}

}

**Файл HTTPRequest.cs:**

using System.Net.Sockets;

using System.Text;

namespace HTTP

{

public class HTTPRequest

{

public string Method;

public string Path;

public string Boundary;

public byte[] HTTPRequestBytes;

public static HTTPRequest Parse(NetworkStream stream)

{

byte[] buffer = new byte[1024 \* 4]; // 4 kB buffer

int bytesRead;

using (MemoryStream memoryStream = new MemoryStream())

{

bool avb = true;

try

{

DateTime lastReceiveTime = DateTime.Now;

const int TIMEOUT\_MS = 1000; // таймаут 1 секунда

while (true)

{

if (stream.DataAvailable)

{

bytesRead = stream.Read(buffer, 0, buffer.Length);

memoryStream.Write(buffer, 0, bytesRead);

lastReceiveTime = DateTime.Now;

}

else

{

// Проверяем таймаут

if ((DateTime.Now - lastReceiveTime).TotalMilliseconds > TIMEOUT\_MS)

{

break;

}

Thread.Sleep(10); // небольшая пауза чтобы не грузить процессор

}

// Проверка на отключение сокета

if (!stream.Socket.Connected)

{

break;

}

}

}

catch (IOException ioEx)

{

Console.WriteLine("Ошибка ввода-вывода: " + ioEx.Message);

}

catch (ObjectDisposedException)

{

Console.WriteLine("Соединение закрыто");

}

catch (Exception ex)

{

Console.WriteLine("Произошла ошибка: " + ex.Message);

}

Console.WriteLine("\nЧтение из потока окончено");

HTTPRequest request = new HTTPRequest();

request.HTTPRequestBytes = memoryStream.ToArray();

// Преобразуем MemoryStream в строку запроса

string requestString = Encoding.UTF8.GetString(memoryStream.ToArray());

// Разбиваем запрос на строки

string[] lines = requestString.Split(new[] { "\r\n" }, StringSplitOptions.None);

// Проверяем, что есть хотя бы одна строка для заголовка

if (lines.Length > 0)

{

string[] requestLineParts = lines[0].Split(' ');

if (requestLineParts.Length >= 2)

{

request.Method = requestLineParts[0];

request.Path = requestLineParts[1];

}

}

// Ищем границу для multipart/form-data

foreach (string line in lines)

{

if (line.StartsWith("Content-Type: multipart/form-data; boundary="))

{

request.Boundary = line.Substring(line.IndexOf("boundary=") + 9);

break; // Граница найдена, дальнейший поиск не нужен

}

}

return request;

}

}

}

}

**Файл FileHandler.cs:**

using HTTP;

using System.Net.Sockets;

using System.Text;

public static class FileHandler

{

public static async Task<byte[]> HandleUpload(HTTPRequest request)

{

string uploadDirectory = AppDomain.CurrentDomain.BaseDirectory + "uploads";

// Создание директории для загрузки файлов, если ее нет

if (!Directory.Exists(uploadDirectory))

{

Directory.CreateDirectory(uploadDirectory);

}

byte[] buffer = new byte[1024];

using (MemoryStream memoryStream = new MemoryStream())

{

// Преобразуем прочитанные байты в строку

byte[] requestBytes = request.HTTPRequestBytes;

string requestString = Encoding.UTF8.GetString(requestBytes);

// 1. Поиск заголовков

int contentStart = requestString.IndexOf("\r\n\r\n") + 4;

if (contentStart == 3) // IndexOf вернул -1, добавление +4 дало 3

throw new InvalidOperationException("HTTP headers not properly terminated.");

string headers = requestString.Substring(0, contentStart);

// 2. Извлечение boundary

string boundary = ExtractBoundaryFromHeaders(headers);

if (string.IsNullOrEmpty(boundary))

throw new InvalidOperationException("Boundary not found in Content-Type header.");

// Добавляем "--" к началу boundary для корректного поиска

string fullBoundary = boundary;

// 3. Поиск имени файла

string contentDispositionHeader = "Content-Disposition:";

int contentDispositionIndex = requestString.IndexOf(contentDispositionHeader);

if (contentDispositionIndex == -1)

throw new InvalidOperationException("Content-Disposition header not found.");

int filenameStart = requestString.IndexOf("filename=\"", contentDispositionIndex);

if (filenameStart == -1)

throw new InvalidOperationException("Filename not found in Content-Disposition header.");

filenameStart += 10; // Пропускаем "filename=\""

int filenameEnd = requestString.IndexOf("\"", filenameStart);

if (filenameEnd == -1)

throw new InvalidOperationException("Invalid filename format.");

string fileName = requestString.Substring(filenameStart, filenameEnd - filenameStart);

Console.WriteLine($"Extracted filename: {fileName}");

// 4. Поиск начала данных файла

int fileDataStart = requestString.IndexOf("\r\n\r\n", contentDispositionIndex) + 4;

if (fileDataStart == 3)

throw new InvalidOperationException("File data not properly terminated.");

// 5. Поиск следующего boundary

int boundaryIndex = requestString.IndexOf(fullBoundary, fileDataStart);

if (boundaryIndex == -1)

throw new InvalidOperationException("Boundary not found in request body.");

// Поиск заголовка Content-Length

string contentLengthHeader = "Content-Length:";

int contentLengthIndex = requestString.IndexOf(contentLengthHeader);

if (contentLengthIndex == -1)

throw new InvalidOperationException("Content-Length header not found.");

// Индекс начала числа в заголовке

int contentLengthStart = contentLengthIndex + contentLengthHeader.Length;

int contentLengthEnd = requestString.IndexOf("\r\n", contentLengthStart);

if (contentLengthEnd == -1)

contentLengthEnd = requestString.Length; // если это последний заголовок

// Извлекаем значение Content-Length и парсим его

string contentLengthStr = requestString.Substring(contentLengthStart, contentLengthEnd - contentLengthStart).Trim();

if (!int.TryParse(contentLengthStr, out int contentLength))

throw new InvalidOperationException("Invalid Content-Length value.");

// Удаляем "\r\n" перед boundary

int fileDataLength = int.Parse(contentLengthStr); // Учитываем \r\n перед boundary

if (fileDataLength < 0)

throw new InvalidOperationException("Invalid file data length.");

byte[] actualFileBytes = requestBytes.Skip(fileDataStart).Take(fileDataLength).ToArray();

return actualFileBytes;

}

}

private static string ExtractBoundaryFromHeaders(string headers)

{

int boundaryIndex = headers.IndexOf("boundary=");

if (boundaryIndex > -1)

{

int boundaryStart = boundaryIndex + 9; // длина строки "boundary="

int boundaryEnd = headers.IndexOf("\r\n", boundaryStart);

return headers.Substring(boundaryStart, boundaryEnd - boundaryStart);

}

return null;

}

public static void SendFile(NetworkStream stream, string path)

{

string filePath = Path.Combine(Environment.CurrentDirectory, path.TrimStart('/'));

if (File.Exists(filePath))

{

byte[] fileBytes = File.ReadAllBytes(filePath);

HTTPResponse response = new HTTPResponse();

string contentType = GetContentType(filePath);

response.SendFile(stream, fileBytes, contentType);

}

else

{

HTTPResponse response = new HTTPResponse();

response.SendError(stream, "404 Not Found");

}

}

public static string GetHtmlForm()

{

return "<html lang=\"en\">\r\n<head>\r\n <meta charset=\"UTF-8\">\r\n <meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">\r\n <title>Image Upload Form</title>\r\n <script src=\"https://cdn.tailwindcss.com\"></script>\r\n <link rel=\"stylesheet\" href=\"https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.3/css/all.min.css\"></link>\r\n <link href=\"https://fonts.googleapis.com/css2?family=Roboto:wght@400;500;700&display=swap\" rel=\"stylesheet\">\r\n <style>\r\n body {\r\n font-family: 'Roboto', sans-serif;\r\n }\r\n .file-input::-webkit-file-upload-button {\r\n visibility: hidden;\r\n }\r\n .file-input::before {\r\n content: 'Select Image';\r\n display: inline-block;\r\n background: linear-gradient(to right, #4f46e5, #3b82f6);\r\n border: 1px solid #4f46e5;\r\n border-radius: 0.375rem;\r\n padding: 0.5rem 1rem;\r\n outline: none;\r\n white-space: nowrap;\r\n cursor: pointer;\r\n color: white;\r\n font-weight: 700;\r\n font-size: 0.875rem;\r\n transition: background 0.3s ease;\r\n }\r\n .file-input:hover::before {\r\n background: linear-gradient(to right, #3b82f6, #4f46e5);\r\n }\r\n .file-input:active::before {\r\n background: linear-gradient(to right, #3b82f6, #4f46e5);\r\n }\r\n </style>\r\n</head>\r\n<body class=\"bg-gray-100 flex items-center justify-center min-h-screen\">\r\n <div class=\"bg-white p-8 rounded-lg shadow-lg w-full max-w-md\">\r\n <h2 class=\"text-2xl font-bold mb-6 text-center\">Upload Your Image</h2>\r\n <form action=\"http://localhost:4000\" method=\"POST\" enctype=\"multipart/form-data\" class=\"space-y-6\">\r\n <div>\r\n <label for=\"image\" class=\"block text-sm font-medium text-gray-700\">Select Image</label>\r\n <div class=\"mt-1 flex items-center\">\r\n <input type=\"file\" name=\"image\" id=\"image\" accept=\"image/\*\" class=\"file-input block w-full text-sm text-gray-900 border border-gray-300 rounded-lg cursor-pointer bg-gray-50 focus:outline-none focus:border-blue-500\">\r\n </div>\r\n </div>\r\n <div class=\"flex items-center justify-between\">\r\n <button type=\"submit\" class=\"w-full bg-blue-500 text-white font-bold py-2 px-4 rounded-lg hover:bg-blue-600 focus:outline-none focus:ring-2 focus:ring-blue-500 focus:ring-opacity-50 transition duration-300 ease-in-out transform hover:scale-105\">\r\n <i class=\"fas fa-upload mr-2\"></i> Upload Image\r\n </button>\r\n </div>\r\n </form>\r\n </div>\r\n</body>\r\n</html>";

}

private static string GetContentType(string filePath)

{

string extension = Path.GetExtension(filePath).ToLower();

return extension switch

{

".jpg" => "image/jpeg",

".jpeg" => "image/jpeg",

".png" => "image/png",

".gif" => "image/gif",

\_ => "application/octet-stream"

};

}

}

**Файл UserManager.cs:**

using HTTP.Claster;

using System.Net.Sockets;

namespace HTTP.UserManager

{

public class UserManager

{

private List<TcpClient> \_users = new List<TcpClient>();

private int \_maxCountOfUsers;

private int \_curCount => \_users.Count;

public UserManager(int maxCountUsers)

{

\_maxCountOfUsers = maxCountUsers;

}

public void AddUser(TcpClient user)

{

lock (\_users)

{

\_users.Add(user);

}

}

public void RemoveUser(TcpClient user)

{

lock (\_users)

{

\_users.Remove(user);

}

}

public Task<byte[]> SendData(ClasterManager clasterManager, TcpClient tcpClient, byte[] bytes)

{

while (true)

{

if (!clasterManager.FreeClasters)

{

if (\_maxCountOfUsers < \_curCount)

{

lock (this)

{

Monitor.Wait(this);

}

}

else

{

return null;

}

}

else

{

break;

}

}

var buffer = clasterManager.Send(bytes);

lock (this)

{

Monitor.Pulse(this);

}

lock (\_users)

{

\_users.Remove(tcpClient);

}

return buffer;

}

}

}

**Файл ClasterManager.cs:**

using System.Net;

using System.Net.Sockets;

namespace HTTP.Claster

{

public class ClasterManager

{

int \_numOfUsers = 0;

int \_maxUsers = 2;

object \_lock = new object();

List<Claster> \_clasters = new List<Claster>();

Socket \_clasterListener;

public bool FreeClasters

{

get

{

return \_clasters.Any(c => c.IsFree);

}

}

public ClasterManager()

{

//Серверный сокет

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

IPEndPoint IP = new IPEndPoint(IPEndPoint.Parse("127.0.0.1").Address, 4100);

\_clasterListener.Bind(IP);

Thread thread = new Thread(ClasterCycle);

thread.Start();

}

public async Task<byte[]> Send(byte[] data)

{

foreach (var item in \_clasters)

{

if (item.IsFree)

{

return await item.SendData(data);

}

}

return null;

}

private void ClasterCycle()

{

while (true)

{

\_clasterListener.Listen(1000);

Socket socket = \_clasterListener.Accept();

Claster claster = new Claster(socket);

\_clasters.Add(claster);

Console.WriteLine("Cluster " + socket.RemoteEndPoint + " connect to server");

}

}

}

}

**Файл Claster.cs:**

using System.Net.Sockets;

using System.Text;

namespace HTTP.Claster

{

public class Claster

{

private byte[] \_dataRecv;

private Socket ClasterSocket;

public bool IsFree => \_isFree;

private bool \_isFree = true;

private object \_monitor = new object();

public Claster(Socket clasterSocket)

{

ClasterSocket = clasterSocket;

Thread thread1 = new Thread(ClasterRecive);

thread1.Start();

}

public async Task<byte[]> SendData(byte[] bytes)

{

SendToClasterData(bytes);

lock (\_monitor)

{

Monitor.Wait(\_monitor);

}

return \_dataRecv;

}

private void ClasterRecive()

{

while (true)

{

if (ClasterSocket.Available > 0)

{

// Получаем размер данных

byte[] sizeBuffer = new byte[4];

int totalRead = 0;

while (totalRead < 4)

{

int read = ClasterSocket.Receive(sizeBuffer, totalRead, 4 - totalRead, SocketFlags.None);

if (read == 0) break;

totalRead += read;

}

int expectedSize = BitConverter.ToInt32(sizeBuffer, 0);

Console.WriteLine($"Expecting to receive {expectedSize} bytes");

// Получаем сами данные

using (MemoryStream memoryStream = new MemoryStream())

{

byte[] buffer = new byte[8192]; // больший размер буфера для эффективности

totalRead = 0;

while (totalRead < expectedSize)

{

int read = ClasterSocket.Receive(buffer, 0,

Math.Min(buffer.Length, expectedSize - totalRead),

SocketFlags.None);

if (read == 0) break; // соединение закрыто

memoryStream.Write(buffer, 0, read);

totalRead += read;

}

\_dataRecv = memoryStream.ToArray();

string str = Encoding.UTF8.GetString(\_dataRecv);

lock (\_monitor)

{

Monitor.PulseAll(\_monitor);

}

}

}

}

}

private void SendToClasterData(byte[] dataSend)

{

Console.WriteLine("Send to claster size of data = " + dataSend.Length);

// Сначала отправляем размер данных

byte[] sizeBytes = BitConverter.GetBytes(dataSend.Length);

ClasterSocket.Send(sizeBytes);

// Затем отправляем сами данные

int totalSent = 0;

while (totalSent < dataSend.Length)

{

int sent = ClasterSocket.Send(dataSend, totalSent,

dataSend.Length - totalSent, SocketFlags.None);

totalSent += sent;

}

}

}

}

**Файл Program.cs:**

using Cluster.ClusterTools;

namespace Program

{

class Program

{

public static void Main(string[] args)

{

ServerCluster serverCluster = new ServerCluster();

}

}

}

**Файл Pocket.cs:**

namespace Cluster.PocketOfData

{

public class Pocket

{

public int PartOfMesage;

public byte[] PartOfData;

public Pocket() { }

public Pocket(int partOfMesage, byte[] partOfData)

{

PartOfData = partOfData;

PartOfMesage = partOfMesage;

}

}

}

**Файл ServerCluster.cs:**

using Cluster.PocketOfData;

using Newtonsoft.Json;

using System.Drawing;

using System.Drawing.Imaging;

using System.Net;

using System.Net.Sockets;

namespace Cluster.ClusterTools

{

public class ServerCluster

{

List<ClusterUnit> \_clusterUnits = new List<ClusterUnit>();

List<Pocket> \_pocketsToSend = new List<Pocket>();

List<Pocket> \_pocketsToRecive = new List<Pocket>();

int sizeOfPockets => \_clusterUnits.Count;

//Для общения с ClusterUnit

Socket \_serverClusterSocket;

//Для общения с сервером

Socket \_serverSocket;

public ServerCluster()

{

string pathToConfig = AppDomain.CurrentDomain.BaseDirectory + "MetaData\\Configure.json";

string json = File.ReadAllText(pathToConfig);

var dictionary = JsonConvert.DeserializeObject<Dictionary<string, string>>(json);

//Создания сокета кластера, для общения кластера и сокета

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

IPEndPoint IP = new IPEndPoint(IPEndPoint.Parse(dictionary["IPadres"]).Address, int.Parse(dictionary["PORT"]));

\_serverSocket.Bind(IP);

IP = new IPEndPoint(IPEndPoint.Parse(dictionary["SERVERIP"]).Address, int.Parse(dictionary["SERVERPORT"]));

while (true)

{

try

{

\_serverSocket.Connect(IP);

Console.WriteLine("Cluster connect to server with IP:PORT = " + \_serverSocket.LocalEndPoint);

break;

}

catch (Exception ex)

{

Console.WriteLine($"Error: {ex.Message}");

Thread.Sleep(100);

}

}

//Присоеденение к серверу кластера вычислительных блоков

IP = new IPEndPoint(IPEndPoint.Parse(dictionary["SERVERUNITIP"]).Address, int.Parse(dictionary["SERVERUNITPORT"]));

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

\_serverClusterSocket.Bind(IP);

\_serverClusterSocket.Listen(12);

Console.WriteLine("Cluster start listen on IP:PORT = " + \_serverClusterSocket.LocalEndPoint);

Thread threadCluster = new Thread(ClasterListener);

threadCluster.Start();

Thread threadServer = new Thread(CycleOfServer);

threadServer.Start();

}

private void CycleOfServer()

{

while (true)

{

if (\_serverSocket.Available > 0)

{

// Получаем размер данных

byte[] sizeBuffer = new byte[4];

int totalRead = 0;

while (totalRead < 4)

{

int read = \_serverSocket.Receive(sizeBuffer, totalRead, 4 - totalRead, SocketFlags.None);

if (read == 0) break;

totalRead += read;

}

int expectedSize = BitConverter.ToInt32(sizeBuffer, 0);

Console.WriteLine($"Expecting to receive {expectedSize} bytes");

// Получаем сами данные

using (MemoryStream memoryStream = new MemoryStream())

{

byte[] buffer = new byte[8192]; // больший размер буфера для эффективности

totalRead = 0;

while (totalRead < expectedSize)

{

int read = \_serverSocket.Receive(buffer, 0,

Math.Min(buffer.Length, expectedSize - totalRead),

SocketFlags.None);

if (read == 0) break; // соединение закрыто

memoryStream.Write(buffer, 0, read);

totalRead += read;

}

Console.WriteLine($"Actually received {totalRead} bytes");

if (totalRead == expectedSize)

{

memoryStream.Position = 0; // сбрасываем позицию в начало

DesideTask(memoryStream);

}

else

{

Console.WriteLine("Error: Incomplete data received");

}

}

}

}

}

//Разделение изображения на части

private void DesideTask(MemoryStream memoryStream)

{

using (Bitmap bitmap = new Bitmap(memoryStream))

{

int partHeight = bitmap.Height / \_clusterUnits.Count;

for (int i = 0; i < \_clusterUnits.Count; i++)

{

using (Bitmap partBitmap = bitmap.Clone(new Rectangle(0, i \* partHeight, bitmap.Width, partHeight), bitmap.PixelFormat))

{

// Конвертируем часть изображения в массив байтов

byte[] imageBytes;

using (MemoryStream ms = new MemoryStream())

{

partBitmap.Save(ms, ImageFormat.Png);

imageBytes = ms.ToArray();

}

\_pocketsToSend.Add(new Pocket(i, imageBytes));

}

}

SendImageToUnits();

}

}

//Отправка изображений на блоки вычисления

private void SendImageToUnits()

{

for (int i = 0; i < \_clusterUnits.Count; i++)

{

\_clusterUnits[i].SendDataToUnit(\_pocketsToSend[i]);

}

Console.WriteLine("Send to unit's block of IMG, num of block's = " + \_clusterUnits.Count);

\_pocketsToSend.Clear();

}

//Прием изображеня из блоков вычисления

private void RecivePocketsFromUnits(byte[] dataSend)

{

// Сначала отправляем размер данных

byte[] sizeBytes = BitConverter.GetBytes(dataSend.Length);

\_serverSocket.Send(sizeBytes);

// Затем отправляем сами данные

int totalSent = 0;

while (totalSent < dataSend.Length)

{

int sent = \_serverSocket.Send(dataSend, totalSent,

dataSend.Length - totalSent, SocketFlags.None);

totalSent += sent;

}

}

//Обработка пакетов.

public void AddPocket(Pocket pocket)

{

\_pocketsToRecive.Add(pocket);

if (\_pocketsToRecive.Count == \_clusterUnits.Count)

{

using (MemoryStream memoryStream = new MemoryStream())

{

List<Bitmap> bitmaps = new List<Bitmap>();

int index = 0;

int addedParts = 0;

while (true)

{

foreach (var poc in \_pocketsToRecive)

{

if (index == poc.PartOfMesage)

{

using (MemoryStream memory = new MemoryStream(poc.PartOfData))

{

bitmaps.Add(new Bitmap(memory));

}

//memoryStream.Write(poc.PartOfData);

index++;

break;

}

}

if (index == \_pocketsToRecive.Count)

{

break;

}

}

int height = bitmaps.Sum(x => x.Height);

int width = bitmaps[0].Width;

Bitmap combinedBitmap = new Bitmap(width, height);

using (Graphics g = Graphics.FromImage(combinedBitmap))

{

// Очищаем фон (например, делаем его белым)

g.Clear(Color.White);

int offsetY = 0;

foreach (var item in bitmaps)

{

g.DrawImage(item, 0, offsetY); // Рисуем текущее изображение

offsetY += item.Height;

}

}

combinedBitmap.Save(memoryStream, ImageFormat.Jpeg);

RecivePocketsFromUnits(memoryStream.ToArray());

foreach (var item in bitmaps)

{

item.Dispose();

}

\_pocketsToRecive.Clear();

}

}

}

private void ClasterListener()

{

while (true)

{

var unitSocket = \_serverClusterSocket.Accept();

\_clusterUnits.Add(new ClusterUnit(unitSocket, this));

Console.WriteLine("Connect to cluster unit block with IP:PORT = " + unitSocket.RemoteEndPoint);

}

}

}

}

**Файл ClusterUnit.cs:**

using Cluster.PocketOfData;

using Newtonsoft.Json;

using System.Net.Sockets;

using System.Text;

namespace Cluster.ClusterTools

{

public class ClusterUnit

{

EventHandler \_complete;

Socket \_unitSocet;

ServerCluster \_serverCluster;

bool \_isFree = true;

public ClusterUnit(Socket socket, ServerCluster serverCluster)

{

\_unitSocet = socket;

\_serverCluster = serverCluster;

Thread thread = new Thread(CileOfListen);

thread.Start();

}

//Прием данных с UnitBlock

private void CileOfListen()

{

while (true)

{

if (\_unitSocet.Available > 0)

{

// Получаем размер данных

byte[] sizeBuffer = new byte[4];

int totalRead = 0;

while (totalRead < 4)

{

int read = \_unitSocet.Receive(sizeBuffer, totalRead, 4 - totalRead, SocketFlags.None);

if (read == 0) break;

totalRead += read;

}

int expectedSize = BitConverter.ToInt32(sizeBuffer, 0);

Console.WriteLine($"Expecting to receive {expectedSize} bytes");

// Получаем сами данные

using (MemoryStream memoryStream = new MemoryStream())

{

byte[] buffer = new byte[8192]; // больший размер буфера для эффективности

totalRead = 0;

while (totalRead < expectedSize)

{

int read = \_unitSocet.Receive(buffer, 0,

Math.Min(buffer.Length, expectedSize - totalRead),

SocketFlags.None);

if (read == 0) break; // соединение закрыто

memoryStream.Write(buffer, 0, read);

totalRead += read;

}

Console.WriteLine($"Actually received {totalRead} bytes");

Pocket pocket = JsonConvert.DeserializeObject<Pocket>(Encoding.UTF8.GetString(memoryStream.GetBuffer()));

\_serverCluster.AddPocket(pocket);

}

}

}

}

public void SendDataToUnit(Pocket pocket)

{

if (\_isFree)

{

var dataSend = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(pocket));

// Сначала отправляем размер данных

byte[] sizeBytes = BitConverter.GetBytes(dataSend.Length);

\_unitSocet.Send(sizeBytes);

// Затем отправляем сами данные

int totalSent = 0;

while (totalSent < dataSend.Length)

{

int sent = \_unitSocet.Send(dataSend, totalSent,

dataSend.Length - totalSent, SocketFlags.None);

totalSent += sent;

}

}

}

}

}

**Файл Program.cs:**

using ClusterUnit.UnitClient;

namespace Program

{

class Program

{

public static void Main(string[] args)

{

Unit unit = new Unit();

}

}

}

**Файл Unit.cs:**

using ClusterUnit.ComputeTools;

using ClusterUnit.NetPocket;

using Newtonsoft.Json;

using System.Net;

using System.Net.Sockets;

using System.Text;

namespace ClusterUnit.UnitClient

{

public class Unit

{

Socket \_clasterSocket;

CompressIMG \_compressIMG;

FastCompressIMG \_fastDCTCompressIMG;

public Unit()

{

string path = AppDomain.CurrentDomain.BaseDirectory + "\\Configure.json";

string json = File.ReadAllText(path);

var dictionary = JsonConvert.DeserializeObject<Dictionary<string, string>>(json);

IPEndPoint IP = new IPEndPoint(IPEndPoint.Parse(dictionary["IPadres"]).Address, int.Parse(dictionary["PORT"]));

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

IP = new IPEndPoint(IPEndPoint.Parse(dictionary["CLASTERIP"]).Address, int.Parse(dictionary["CLASTERPORT"]));

while (true)

{

try

{

\_clasterSocket.Connect(IP);

Console.WriteLine("Unit connect to cluster with IP:PORT = " + \_clasterSocket.RemoteEndPoint);

break;

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

Thread.Sleep(100);

}

}

\_compressIMG = new CompressIMG();

\_fastDCTCompressIMG = new FastCompressIMG();

Thread thread = new Thread(Listen);

thread.Start();

}

private async void Listen()

{

while (true)

{

if (\_clasterSocket.Available > 0)

{

// Получаем размер данных

byte[] sizeBuffer = new byte[4];

int totalRead = 0;

while (totalRead < 4)

{

int read = \_clasterSocket.Receive(sizeBuffer, totalRead, 4 - totalRead, SocketFlags.None);

if (read == 0) break;

totalRead += read;

}

int expectedSize = BitConverter.ToInt32(sizeBuffer, 0);

Console.WriteLine($"Expecting to receive {expectedSize} bytes");

// Получаем сами данные

using (MemoryStream memoryStream = new MemoryStream())

{

byte[] buffer = new byte[8192]; // больший размер буфера для эффективности

totalRead = 0;

while (totalRead < expectedSize)

{

int read = \_clasterSocket.Receive(buffer, 0,

Math.Min(buffer.Length, expectedSize - totalRead),

SocketFlags.None);

if (read == 0) break; // соединение закрыто

memoryStream.Write(buffer, 0, read);

totalRead += read;

}

Console.WriteLine($"Actually received {totalRead} bytes");

Pocket pocket = JsonConvert.DeserializeObject<Pocket>(Encoding.UTF8.GetString(memoryStream.GetBuffer()));

Console.WriteLine("Unit recive block of data;\n" + "part of block = " + pocket.PartOfMesage + "\n" + "Num of byte's = " + pocket.PartOfData.Length + "\n");

DateTime dateTime = DateTime.Now;

pocket = new Pocket(pocket.PartOfMesage, \_fastDCTCompressIMG.CompressImageAsync(pocket.PartOfData).Result);

var time = DateTime.Now - dateTime;

Console.WriteLine("\nComplete time: " + time + "\n");

string st = JsonConvert.SerializeObject(pocket);

var bytes = Encoding.UTF8.GetBytes(st);

SendLengthAndData(bytes);

Console.WriteLine("Unit send block of data;\n" + "part of block = " + pocket.PartOfMesage + "\n" + "Num of byte's = " + pocket.PartOfData.Length + "\n");

}

}

}

}

private void SendLengthAndData(byte[] dataSend)

{

// Сначала отправляем размер данных

byte[] sizeBytes = BitConverter.GetBytes(dataSend.Length);

\_clasterSocket.Send(sizeBytes);

// Затем отправляем сами данные

int totalSent = 0;

while (totalSent < dataSend.Length)

{

int sent = \_clasterSocket.Send(dataSend, totalSent,

dataSend.Length - totalSent, SocketFlags.None);

totalSent += sent;

}

}

}

}

**Файл FastCompressIMG.cs:**

using System.Drawing;

using System.Drawing.Imaging;

using System.Runtime.InteropServices;

namespace ClusterUnit.ComputeTools

{

public class FastCompressIMG

{

static readonly int[,] QuantizationMatrix = {

{ 32, 24, 20, 32, 48, 80, 102, 122 },

{ 24, 24, 28, 38, 52, 116, 120, 110 },

{ 28, 26, 32, 48, 80, 114, 138, 112 },

{ 28, 34, 44, 58, 102, 174, 160, 124 },

{ 36, 44, 74, 112, 136, 218, 206, 154 },

{ 48, 70, 110, 128, 162, 208, 226, 184 },

{ 98, 128, 156, 174, 206, 242, 240, 202 },

{ 144, 184, 190, 196, 224, 200, 206, 198 }

};

private double[,,,] cosTable;

private double[,] dctMatrix;

private double[,] idctMatrix;

private int \_countOfThreads = 0;

private int \_numberOfThreads = 8;

private int blockSize = 8;

private List<double[,]> \_dataBlocks = new List<double[,]>();

public FastCompressIMG()

{

double c1 = Math.PI / 16;

cosTable = new double[8, 8, 8, 8];

dctMatrix = new double[8, 8];

idctMatrix = new double[8, 8];

InitializeCosTable();

}

// Предварительные вычисления для косинусов

private void InitializeCosTable()

{

double c1 = Math.PI / 16;

// Заполняем таблицу значений косинусов для всех индексов

for (int u = 0; u < 8; u++)

{

for (int v = 0; v < 8; v++)

{

for (int x = 0; x < 8; x++)

{

for (int y = 0; y < 8; y++)

{

cosTable[u, v, x, y] = Math.Cos((2 \* x + 1) \* u \* c1) \* Math.Cos((2 \* y + 1) \* v \* c1);

}

}

}

}

}

public async Task<byte[]> CompressImageAsync(byte[] buffer)

{

byte[] bytes = new byte[buffer.Length];

using (MemoryStream memoryStream = new MemoryStream(buffer))

{

using (Bitmap original = new Bitmap(memoryStream))

{

int originalWidth = original.Width;

int originalHeight = original.Height;

// Добавляем отступы, если нужно

Bitmap paddedImage = AddPadding(original);

int width = paddedImage.Width;

int height = paddedImage.Height;

// Получаем данные изображения в массив байт для обработки

BitmapData bmpData = paddedImage.LockBits(new Rectangle(0, 0, width, height), ImageLockMode.ReadWrite, paddedImage.PixelFormat);

int bytesPerPixel = Image.GetPixelFormatSize(paddedImage.PixelFormat) / 8;

int stride = bmpData.Stride;

IntPtr ptr = bmpData.Scan0;

byte[] pixelData = new byte[stride \* height];

Marshal.Copy(ptr, pixelData, 0, pixelData.Length);

// Параллельная обработка каждого блока 8x8

Parallel.For(0, width / blockSize, i =>

{

for (int j = 0; j < height / blockSize; j++)

{

ProcessBlock(pixelData, i \* blockSize, j \* blockSize, stride, bytesPerPixel);

}

});

// Копируем обработанные данные обратно

Marshal.Copy(pixelData, 0, ptr, pixelData.Length);

paddedImage.UnlockBits(bmpData);

Rectangle cropArea = new Rectangle(0, 0, originalWidth, originalHeight);

Bitmap cropped = new Bitmap(cropArea.Width, cropArea.Height);

using (Graphics g = Graphics.FromImage(cropped))

{

g.DrawImage(paddedImage, 0, 0, cropArea, GraphicsUnit.Pixel);

}

// Сохраняем изображение в поток памяти и возвращаем его как байтовый массив

using (MemoryStream memory = new MemoryStream())

{

cropped.Save(memory, ImageFormat.Jpeg);

paddedImage.Dispose();

cropped.Dispose();

\_dataBlocks.Clear();

return memory.ToArray();

}

}

}

}

private void ProcessBlock(byte[] pixelData, int startX, int startY, int stride, int bytesPerPixel)

{

double[,] pixelsR = new double[blockSize, blockSize];

double[,] pixelsG = new double[blockSize, blockSize];

double[,] pixelsB = new double[blockSize, blockSize];

// Извлекаем блок пикселей для каждого канала

for (int x = 0; x < blockSize; x++)

{

for (int y = 0; y < blockSize; y++)

{

int pixelIndex = ((startY + y) \* stride) + ((startX + x) \* bytesPerPixel);

pixelsR[x, y] = pixelData[pixelIndex + 2]; // Red

pixelsG[x, y] = pixelData[pixelIndex + 1]; // Green

pixelsB[x, y] = pixelData[pixelIndex]; // Blue

}

}

// Обрабатываем каждый канал отдельно

// Красный канал

double[,] dctBlockR = DCT(pixelsR);

int[,] quantizedBlockR = QuantizeBlock(dctBlockR);

double[,] dequantizedBlockR = DequantizeBlock(quantizedBlockR);

double[,] idctBlockR = IDCT(dequantizedBlockR);

// Зеленый канал

double[,] dctBlockG = DCT(pixelsG);

int[,] quantizedBlockG = QuantizeBlock(dctBlockG);

double[,] dequantizedBlockG = DequantizeBlock(quantizedBlockG);

double[,] idctBlockG = IDCT(dequantizedBlockG);

// Синий канал

double[,] dctBlockB = DCT(pixelsB);

int[,] quantizedBlockB = QuantizeBlock(dctBlockB);

double[,] dequantizedBlockB = DequantizeBlock(quantizedBlockB);

double[,] idctBlockB = IDCT(dequantizedBlockB);

// Копируем обработанные данные обратно в массив пикселей

for (int x = 0; x < blockSize; x++)

{

for (int y = 0; y < blockSize; y++)

{

int pixelIndex = ((startY + y) \* stride) + ((startX + x) \* bytesPerPixel);

pixelData[pixelIndex] = (byte)Math.Clamp(idctBlockB[x, y], 0, 255); // Blue

pixelData[pixelIndex + 1] = (byte)Math.Clamp(idctBlockG[x, y], 0, 255); // Green

pixelData[pixelIndex + 2] = (byte)Math.Clamp(idctBlockR[x, y], 0, 255); // Red

}

}

}

public Bitmap AddPadding(Bitmap original)

{

int newWidth = ((original.Width + 7) / 8) \* 8; // округление до ближайшего числа, кратного 8

int newHeight = ((original.Height + 7) / 8) \* 8;

// Создаем новое изображение с новыми размерами и заполняем его белым цветом

Bitmap paddedImage = new Bitmap(newWidth, newHeight);

using (Graphics g = Graphics.FromImage(paddedImage))

{

g.Clear(Color.White); // Заполняем пустые пиксели белым (или другим цветом)

g.DrawImage(original, 0, 0); // Копируем оригинальное изображение

}

return paddedImage;

}

public int[,] QuantizeBlock(double[,] block)

{

int[,] quantizedBlock = new int[8, 8];

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

// Делим коэффициенты блока на значения матрицы квантования и округляем

quantizedBlock[i, j] = (int)Math.Round(block[i, j] / QuantizationMatrix[i, j]);

}

}

return quantizedBlock;

}

public double[,] DequantizeBlock(int[,] quantizedBlock)

{

double[,] dequantizedBlock = new double[8, 8];

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

// Умножаем на значения матрицы квантования, чтобы восстановить блок

dequantizedBlock[i, j] = quantizedBlock[i, j] \* QuantizationMatrix[i, j];

}

}

return dequantizedBlock;

}

// DCT (Дискретное косинусное преобразование)

public double[,] DCT(double[,] block)

{

double[,] result = new double[8, 8];

for (int u = 0; u < 8; u++)

{

for (int v = 0; v < 8; v++)

{

double sum = 0.0;

for (int x = 0; x < 8; x++)

{

for (int y = 0; y < 8; y++)

{

sum += block[x, y] \* cosTable[u, v, x, y];

}

}

double cu = (u == 0) ? 1.0 / Math.Sqrt(2) : 1.0;

double cv = (v == 0) ? 1.0 / Math.Sqrt(2) : 1.0;

result[u, v] = 0.25 \* cu \* cv \* sum;

}

}

return result;

}

// IDCT (Обратное дискретное косинусное преобразование)

public double[,] IDCT(double[,] block)

{

double[,] result = new double[8, 8];

for (int x = 0; x < 8; x++)

{

for (int y = 0; y < 8; y++)

{

double sum = 0.0;

for (int u = 0; u < 8; u++)

{

for (int v = 0; v < 8; v++)

{

double cu = (u == 0) ? 1.0 / Math.Sqrt(2) : 1.0;

double cv = (v == 0) ? 1.0 / Math.Sqrt(2) : 1.0;

sum += cu \* cv \* block[u, v] \* cosTable[u, v, x, y];

}

}

result[x, y] = 0.25 \* sum;

}

}

return result;

}

}

}