МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
Гомельский государственный технический университет имени П.О.Сухого

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

специальность 1-40 05 01-01 Информационные системы и технологии
(в проектировании и производстве)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту
по дисциплине «Распределённые информационные системы»

на тему «Master-slave система на основе TCP для применения оператора Робертса к изображению»

Исполнитель: студент гр. ИТИ-41
Ковшаров Г. Ю.
Руководитель: доцент
Комраков В.В.

Дата проверки: _____
Дата допуска к защите: _____
Дата защиты: _____
Оценка работы: _____

Подписи членов комиссии
по защите курсового проекта: _____

Гомель 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность исследования обусловлена постоянным ростом объёмов визуальных данных, требующих быстрого и эффективного анализа. Современные системы обработки изображений используются в областях компьютерного зрения, робототехники, автоматизированного контроля качества и медицинской диагностики, где скорость и точность обработки являются критически важными параметрами. Увеличение вычислительных нагрузок приводит к необходимости применения распределённых моделей организации вычислительного процесса, позволяющих разделять задачи между несколькими узлами и выполнять их параллельно. Использование подобных моделей обеспечивает масштабируемость и повышение производительности, что особенно важно при реализации методов выделения структур и контуров изображения. Среди широко распространённых средств обнаружения границ значительное место занимает оператор Робертса, характеризующийся простотой реализации и низкой вычислительной сложностью. Эти свойства делают его подходящим для включения в состав распределённых систем, ориентированных на обработку данных в режиме реального времени.

Целью курсовой работы является разработка распределённой системы обработки изображений, обеспечивающей параллельное выполнение вычислений с применением оператора Робертса. Для достижения поставленной цели предполагается провести анализ современных информационных технологий и программных решений, применяемых в сфере обработки изображений, изучить особенности алгоритма выделения границ оператором Робертса и сформировать подробную постановку задачи с указанием типов входных данных и ожидаемых результатов. Предусматривается построение функциональной схемы программного комплекса, определение структуры основных модулей и характера их взаимодействия, а также реализация программной системы, способной выполнять распределённую обработку изображений. Завершающим этапом является тестирование разработанной программы и её верификация, сопровождаемые оформлением и анализом полученных экспериментальных данных.

Предложенный подход направлен на создание эффективного программного решения, использующего преимущества параллельной обработки и обеспечивающего применение классического метода выделения границ в условиях распределённой вычислительной среды.

1 АНАЛИТИЧЕСКИЙ ОБЗОР МЕТОДОВ ОБРАБОТКИ ИЗОБРАЖЕНИЙ И СЕТЕВЫХ АРХИТЕКТУР

1.1 Обзор поставленной задачи

Разрабатываемая программная система предназначена для обработки изображений, и необходимости обеспечения устойчивой производительности. Задача заключается в создании решения, способного распределять обработку данных между несколькими вычислительными ресурсами, обеспечивая ускорение выполнения операций и повышение эффективности использования доступных мощностей. Для достижения этих целей предполагается организация параллельной обработки, позволяющей разделять исходные данные на фрагменты и выполнять вычисления одновременно на нескольких узлах или потоках.

Требования к системе включают необходимость поддержки стабильного обмена данными между компонентами, надёжности при передаче и обработке информации, а также способности корректно функционировать при переменной нагрузке. В процессе разработки необходимо учитывать особенности работы с изображениями, так как данные данного типа требуют тщательного контроля структуры, формата и последовательности выполнения операций. Важным аспектом является обеспечение масштабируемости, позволяющей адаптировать систему к увеличению количества входных данных без существенного снижения производительности

Особое внимание уделяется выбору механизмов, обеспечивающих распределение вычислительных задач. Для эффективного управления нагрузкой требуется применение алгоритмов планирования, определяющих порядок и последовательность выполнения операций в многопоточной или многозадачной среде. От корректного выбора алгоритма зависит равномерность распределения вычислительных ресурсов и общая скорость обработки данных. Используемая модель взаимодействия компонентов системы также должна обеспечивать согласованность обмена информацией и отсутствие задержек, связанных с некорректной синхронизацией процессов.

Реализация программного комплекса предполагает выбор соответствующего набора технологий и средств разработки, обеспечивающих поддержку сетевого взаимодействия, параллельного выполнения операций и обработки изображений. При этом необходимо учитывать существующие архитектурные решения, позволяющие организовать обмен данными между узлами системы, а также особенности протоколов передачи, обеспечивающих требуемый уровень надёжности и скорости.

1.2 Обзор средств реализации

Эффективная обработка большого объёма изображений требует применения современных программных и аппаратных средств, способных обеспечивать

высокую производительность, надёжность и масштабируемость. Выбор подходящих технологий и инструментов играет ключевую роль в организации параллельной обработки данных, управлении вычислительными потоками и реализации сетевого взаимодействия между компонентами системы.

Современные информационные системы, предназначенные для обработки данных, таких как изображения, требуют применения специализированных программных и аппаратных средств, обеспечивающих высокую производительность, надёжность и масштабируемость. Такие системы должны эффективно обрабатывать большие объёмы данных, распределять вычислительные задачи между узлами и предоставлять пользователю оперативную информацию о ходе выполнения операций. Для реализации системы применяются инструменты, обеспечивающие выполнение поставленных задач, с использованием языка программирования *C#* и платформы *.NET Core*.

Методы реализации распределённых систем обработки данных классифицируются по степени автоматизации:

– ручные методы, не предполагающие использование программных средств;

– автоматические системы, полностью исключающие участие человека;

– автоматизированные системы, сочетающие программно-аппаратные средства с возможностью контроля со стороны пользователя.

Для реализации *master-slave* системы, направленной на обработку изображений, предпочтение отдаётся автоматизированным системам, которые обеспечивают баланс между автоматизацией процессов и возможностью контроля со стороны разработчиков или пользователей. Такие системы требуют применения программных инструментов, способных поддерживать сетевое взаимодействие, обработку изображений и эффективное распределение задач, что делает выбор технологий важным этапом проектирования.

Одной из перспективных платформ для реализации подобных систем является *.NET Core*, представляющая собой кроссплатформенную среду разработки, созданную для построения современных приложений. Данная платформа поддерживает разработку программного обеспечения для различных операционных систем, таких как *Windows, Linux* и *macOS*, что обеспечивает гибкость при развертывании системы. *.NET Core* предоставляет мощную среду выполнения, которая позволяет эффективно компилировать и исполнять программы, обеспечивая высокую производительность и оптимизацию ресурсов. Платформа поддерживает множество языков программирования, однако в контексте данного проекта основное внимание уделяется языку *C#*, который широко используется для создания сложных приложений благодаря своим возможностям и интеграции с экосистемой *.NET Core*.

Язык программирования *C#* рассматривается как один из ключевых инструментов для реализации системы благодаря своим преимуществам: строгая типизация, поддержка объектно-ориентированного подхода, возможности для асинхронного программирования и высокая производительность. *C#* позволяет создавать структурированный и читаемый код, что упрощает разработку систем,

включающих сложные сетевые взаимодействия и обработку данных. Кроме того, язык обладает гибкостью, позволяя реализовать как серверные, так и клиентские компоненты системы, что делает его подходящим для создания *master-slave* архитектуры. Возможности *C#* включают управление потоками, обработку исключений и работу с данными, что позволяет эффективно решать задачи распределения задач и обработки изображений.

Технологии обработки изображений, которые могут быть применены в системе, включают специализированные библиотеки, предназначенные для выполнения операций, таких как загрузка, фильтрация и анализ изображений. Одной из ключевых задач системы является применение оператора Робертса, который используется для выделения границ на изображении. Оператор Робертса основан на вычислении градиента яркости в соседних пикселях, что позволяет выявлять контуры объектов на изображении. Данный метод отличается относительной простотой и вычислительной эффективностью, что делает его подходящим для распределённых систем, где задачи обработки распределяются между несколькими узлами. Библиотеки обработки изображений, совместимые с *C#* и *.NET Core*, предоставляют широкий набор инструментов для работы с различными форматами изображений, такими как *JPEG, PNG* и *BMP*, а также для выполнения операций фильтрации и преобразования. Эти библиотеки оптимизированы для высокой производительности, что позволяет эффективно обрабатывать изображения даже на системах с ограниченными ресурсами.

Для управления асинхронной обработкой задач и сетевых операций могут применяться технологии асинхронного программирования, встроенные в *C#* и *.NET Core*. Асинхронное программирование позволяет выполнять несколько операций одновременно, минимизируя время ожидания и повышая производительность системы. Например, управляющий узел может одновременно принимать изображения от клиента, распределять задачи по подчинённым узлам и отправлять сообщения о прогрессе обработки, что особенно важно при работе с большими объёмами данных. Возможности асинхронного программирования включают поддержку параллельного выполнения задач, что делает их подходящими для реализации систем, требующих высокой производительности.

Для работы с изображениями на стороне клиента и потенциального отображения результатов обработки могут использоваться технологии, поддерживающие загрузку, обработку и сохранение изображений. Такие технологии позволяют работать с различными форматами изображений и обеспечивают базовые функции для их преобразования. В случае необходимости отображения результатов обработки или прогресса могут быть рассмотрены инструменты для создания простых графических интерфейсов, хотя для упрощения реализации системы предпочтение может быть отдано консольным приложениям. Эти технологии обеспечивают гибкость в работе с данными и позволяют адаптировать клиентскую часть системы под конкретные требования.

Для мониторинга работы системы и отладки могут применяться технологии логирования, которые позволяют фиксировать информацию о сетевых опе-

рациях, распределении задач и обработке изображений. Логирование предоставляет возможность отслеживать этапы выполнения операций, анализировать производительность системы и выявлять возможные ошибки. Логи могут сохраняться в различных форматах, таких как текстовые файлы или консольный вывод, что упрощает анализ работы системы на этапе разработки и тестирования.

Альтернативные подходы к реализации системы включают использование других языков программирования, таких как *Python* или *C++*. *Python* широко применяется в задачах обработки изображений благодаря наличию мощных библиотек и простоте разработки. Однако *Python* может уступать *C#* в производительности и строгой типизации, что делает его менее предпочтительным для сложных распределённых систем. *C++*, в свою очередь, обеспечивает высокую производительность и гибкость в управлении ресурсами, но требует более сложного управления памятью, что увеличивает время разработки. *C#* и *.NET Core* представляют собой оптимальный выбор, сочетая производительность, простоту разработки и поддержку кроссплатформенности.

Таким образом, для реализации были выбраны язык программирования *C#* и платформа *.NET Core*, которые предоставляют широкие возможности для организации сетевого взаимодействия, обработки изображений и распределения задач. Эти технологии характеризуются гибкостью, производительностью и поддержкой кроссплатформенной разработки, что делает их перспективными для создания эффективных распределённых систем.

1.3 Обзор моделей взаимодействия в распределённых системах

Организация эффективного взаимодействия компонентов в распределённых системах является важным аспектом при разработке программных комплексов, работающих с большими объёмами данных. Выбор модели взаимодействия определяет, каким образом распределяются задачи между вычислительными узлами, обеспечивается синхронизация потоков, поддерживается обмен данными и достигается требуемый уровень производительности. Различные подходы к построению взаимодействия оказывают прямое влияние на надёжность системы, скорость обработки и возможность масштабирования, поэтому важно рассмотреть существующие модели и оценить их применимость к конкретной вычислительной задаче.

Для реализации распределённых систем, ориентированных на обработку данных, таких как изображения, часто используется *master-slave* архитектура. Данная модель предполагает разделение процессов на две основные группы: управляющий узел (*master*) и подчинённые узлы (*slave*). Управляющий узел отвечает за координацию работы системы, распределение задач между подчинёнными узлами и взаимодействие с клиентом, который инициирует запросы на обработку данных. Подчинённые узлы выполняют вычислительные задачи, такие как обработка изображений, и возвращают результаты управляющему узлу. Взаимодействие между клиентом, *master*-узлом и *slave*-узлами показано на рисунке 1.1.

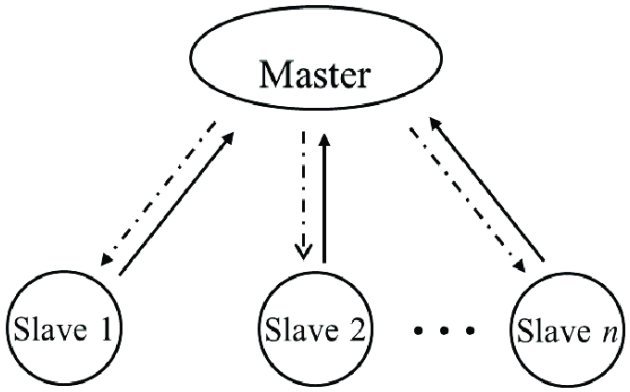

Рисунок 1.1 – Взаимодействие в *master-slave* архитектуре

Master-slave архитектура характеризуется чётким разделением ролей, что позволяет эффективно распределять вычислительные задачи и повышать производительность системы. Управляющий узел принимает запросы от клиента, распределяет их между подчинёнными узлами с использованием определённых алгоритмов и собирает результаты обработки для передачи клиенту. Для распределения задач в подобных системах часто применяются алгоритмы, обеспечивающие равномерную загрузку узлов, такие как алгоритм *Round Robin*. Данный алгоритм предполагает циклическое распределение задач между подчинёнными узлами, что позволяет избежать перегрузки отдельных узлов и обеспечивает сбалансированную работу системы.

При проектировании распределённых систем, таких как *master-slave* архитектура, принято разделять их на три основных уровня:

– уровень взаимодействия с клиентом;

– уровень обработки данных;

– уровень координации и хранения данных.

Уровень взаимодействия с клиентом включает компоненты, обеспечивающие приём запросов от пользователя, таких как отправка изображений для обработки. Этот уровень может включать минималистичные интерфейсы для передачи данных или, при необходимости, графические интерфейсы для отображения результатов обработки. Обычно данный уровень реализуется на стороне клиента, предоставляя пользователю возможность инициировать задачи и получать информацию о их выполнении.

Уровень обработки данных отвечает за выполнение вычислительных задач, таких как применение оператора Робертса для выделения границ на изображении. Этот уровень обычно реализуется на подчинённых узлах, которые получают данные от управляющего узла, выполняют обработку и возвращают результаты. В случае обработки изображений данный уровень включает выполнение операций фильтрации, анализа и преобразования изображений, что требует использования специализированных алгоритмов и инструментов.

Уровень координации и хранения данных отвечает за управление задачами и координацию работы системы. Этот уровень реализуется на управляющем

узле, который принимает запросы от клиента, распределяет задачи между подчинёнными узлами и собирает результаты. В *master-slave* архитектуре координация задач может осуществляться с использованием алгоритмов распределения, таких как *Round Robin*, которые обеспечивают равномерную загрузку подчинённых узлов. В некоторых системах на этом уровне также могут храниться промежуточные данные, например, информация о состоянии обработки или результаты, хотя в данном случае основное внимание уделяется координации, а не хранению данных.

Архитектура *master-slave* допускает различные варианты организации уровней в зависимости от требований системы. Например, уровень взаимодействия с клиентом может быть реализован как на стороне клиента, так и частично на стороне управляющего узла, если требуется удалённое управление представлением данных. Уровень обработки данных может быть полностью сосредоточен на подчинённых узлах или разделён между *master*- и *slave*-узлами для выполнения предварительной обработки.

Гибкость *master-slave* архитектуры позволяет адаптировать её под конкретные задачи. Например, для обработки изображений с применением оператора Робертса требуется эффективное распределение задач между подчинёнными узлами, чтобы обеспечить высокую производительность и минимизировать время обработки. Алгоритм *Round Robin*, используемый для распределения задач, способствует равномерной загрузке узлов, что позволяет избежать узких мест в системе.

Помимо *master-slave* архитектуры, в распределённых системах широко применяются модели *peer-to-peer* и клиент-серверного взаимодействия. В модели *peer-to-peer* каждый узел системы выполняет одновременно функции клиента и сервера, что позволяет равномерно распределять вычислительные задачи и исключает зависимость от одного управляющего узла. Такая модель повышает отказоустойчивость системы, однако требует более сложных механизмов синхронизации и управления состоянием узлов, что может увеличивать накладные расходы при обработке больших объёмов данных.

Клиент-серверная модель, в отличие от *master-slave*, подразумевает наличие центрального сервера, который обрабатывает запросы клиентов и может напрямую координировать вычислительные задачи. Этот подход упрощает управление системой и обеспечивает централизованный контроль, однако при увеличении числа клиентов или объёма данных возможна перегрузка сервера, что снижает масштабируемость и производительность по сравнению с *master-slave*.

Сравнительный анализ этих моделей показывает, что *master-slave* сохраняет преимущества в простоте организации, предсказуемости распределения задач и балансировке нагрузки с использованием алгоритмов планирования, таких как *Round Robin*. Именно эти свойства делают её наиболее подходящей для реализации систем, ориентированных на параллельную обработку больших объёмов изображений с требованиями к высокой производительности и надёжности.

1.4 Обзор алгоритмов планирования

Эффективное распределение вычислительных задач между узлами или потоками в распределённых системах напрямую зависит от применяемых алгоритмов планирования. От их корректного выбора зависит равномерность загрузки вычислительных ресурсов, общая производительность системы и минимизация времени обработки данных. Алгоритмы планирования определяют порядок выполнения задач, приоритеты обработки, а также способы реагирования на изменения нагрузки или отказ отдельных компонентов. В условиях обработки большого объёма изображений использование оптимальных алгоритмов позволяет избежать перегрузки отдельных узлов и обеспечивает стабильное выполнение операций, что является критически важным для поддержания высокой производительности и надёжности системы.

Алгоритм *Round Robin* представляет собой один из наиболее распространённых подходов к распределению задач в распределённых системах, таких как *master-slave* архитектура. Его основная идея заключается в циклическом распределении задач между доступными узлами, что обеспечивает равномерную загрузку и предотвращает перегрузку отдельных компонентов системы. В контексте обработки изображений данный алгоритм позволяет распределять задачи, связанные с применением оператора Робертса, между подчинёнными узлами, что способствует эффективному использованию вычислительных ресурсов и сокращению общего времени обработки.

Принцип работы алгоритма *Round Robin* основан на последовательном назначении задач каждому подчинённому узлу в заранее определённом порядке. Например, если в системе имеется несколько *slave*-узлов, задачи распределяются по очереди: первая задача передаётся первому узлу, вторая – второму, третья – третьему и так далее, пока не будет достигнут последний узел, после чего цикл начинается заново. Такой подход обеспечивает равномерное распределение нагрузки, что особенно важно в системах, где задачи имеют схожую вычислительную сложность, например, при обработке изображений с применением оператора Робертса. Схема работы алгоритма *Round Robin* представлена на рисунке 1.2.

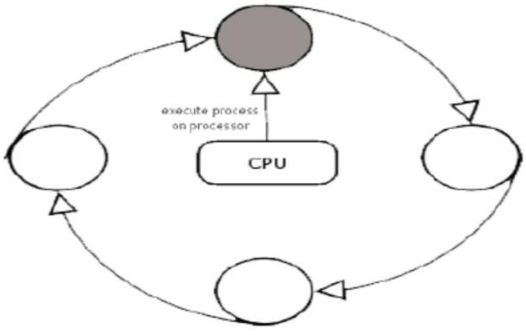

Рисунок 1.2 – Принцип работы алгоритма *Round Robin*

Алгоритм *Round Robin* характеризуется простотой реализации и высокой эффективностью в системах с большим количеством задач. Его использование позволяет избежать ситуации, при которой один узел обрабатывает значительно больше задач, чем другие, что может привести к задержкам и снижению производительности. В *master-slave* архитектуре, ориентированной на обработку изображений, алгоритм *Round Robin* может быть применён для распределения изображений, отправленных клиентом, между подчинёнными узлами, каждый из которых выполняет обработку с использованием оператора Робертса. Это обеспечивает сбалансированную работу системы и минимизирует время ожидания результатов.

Одним из ключевых преимуществ алгоритма *Round Robin* является его способность адаптироваться к динамическим изменениям в системе. Например, если один из подчинённых узлов становится недоступным, алгоритм может быть настроен на исключение этого узла из цикла распределения, что повышает отказоустойчивость системы. Кроме того, алгоритм не требует сложных вычислений для определения, какой узел должен получить следующую задачу, что делает его вычислительно эффективным и подходящим для систем, работающих в реальном времени.

В контексте обработки изображений алгоритм *Round Robin* может быть адаптирован для учёта различных факторов, таких как размер изображения или его сложность. Например, если изображения имеют разное разрешение, алгоритм может быть дополнен механизмами, учитывающими вычислительную мощность подчинённых узлов, чтобы обеспечить более сбалансированное распределение. Однако в большинстве случаев стандартная реализация *Round Robin*, основанная на циклическом распределении, оказывается достаточной для достижения равномерной загрузки.

Помимо алгоритма *Round Robin*, в распределённых системах широко применяются такие подходы к планированию задач, как планирование по приоритету, *shortest job first* (*SJF*) и планирование на основе динамической оценки загрузки узлов. Алгоритмы с приоритетами позволяют отдавать предпочтение задачам, требующим срочного выполнения, однако менее приоритетные задачи могут значительно задерживаться, что снижает предсказуемость обработки всего потока данных. Алгоритм *SJF* назначает выполнение задач с наименьшим ожидаемым временем обработки, что оптимизирует среднее время выполнения, но требует точного прогнозирования длительности каждой задачи, что затруднительно в реальных условиях. Динамическое распределение нагрузки оценивает текущую загруженность узлов и перенаправляет задачи на менее загруженные узлы, обеспечивая адаптивное распределение, но при этом накладные расходы на мониторинг и расчёт нагрузки могут снижать общую производительность.

Сравнение этих алгоритмов с *Round Robin* показывает, что именно циклическое распределение задач обеспечивает наилучший баланс между простотой реализации, равномерной загрузкой узлов и высокой эффективностью для систем с большим количеством однородных задач, таких как обработка изображе-

ний с применением оператора Робертса. *Round Robin* не требует предварительных оценок времени выполнения и сложных вычислений, что делает его особенно подходящим для распределённых систем реального времени.

1.5 Обзор протоколов передачи данных

Эффективная работа распределённых систем невозможна без корректной организации передачи данных между компонентами. Выбор протоколов передачи напрямую влияет на надёжность, скорость обмена информацией и способность системы масштабироваться при увеличении объёма обрабатываемых данных. В условиях обработки изображений, где передача больших массивов информации между узлами является критически важной, протоколы должны обеспечивать минимальные задержки, контроль ошибок и устойчивость к потерям пакетов. Кроме того, протоколы передачи данных определяют подход к синхронизации процессов, управлению потоками и обеспечению согласованности состояния узлов, что делает их ключевым элементом архитектуры распределённых систем.

1.5.1 Протоколы *TCP* (*Transmission Control Protocol*) и *UDP* (*User Datagram Protocol*) представляют собой ключевые средства передачи данных в распределённых системах, таких как *master-slave* архитектура, предназначенная для обработки изображений. В отличие от протоколов, ориентированных на веб-приложения, таких как *HTTP*, *TCP* и *UDP* обеспечивают низкоуровневое взаимодействие между компонентами системы, что делает их подходящими для задач, требующих надёжной передачи данных или высокой скорости обмена сообщениями. В контексте *master-slave* системы, ориентированной на обработку изображений с применением оператора Робертса, протокол *TCP* может использоваться для надёжной передачи изображений между клиентом, управляющим узлом (*master*) и подчинёнными узлами (*slave*), а протокол *UDP* – для оперативного мониторинга прогресса обработки. Принципы работы протоколов *TCP* и *UDP* в распределённых системах показаны на рисунках 1.3.

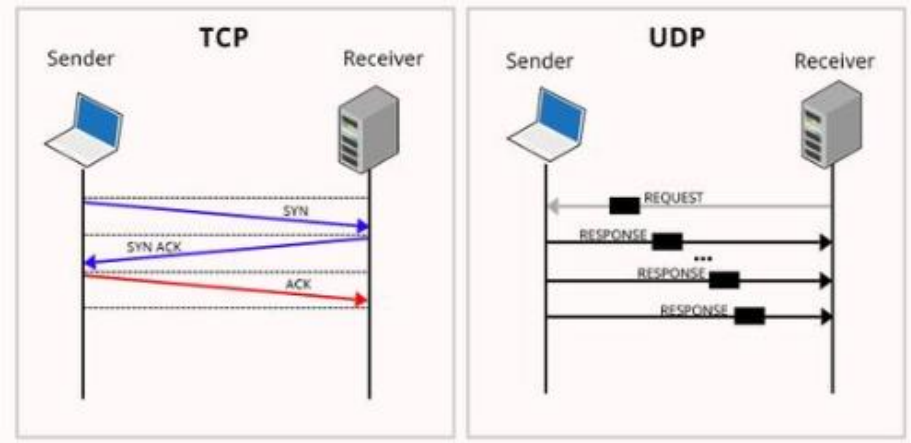

Рисунок 1.3 – Принципы работы протоколов *TCP* и *UDP*

Протокол *TCP* характеризуется установлением соединения между отправителем и получателем, что обеспечивает надёжную передачу данных. Процесс начинается с трёхэтапного рукопожатия, в ходе которого клиент и сервер обмениваются управляющими сообщениями для установления соединения. После этого данные передаются в виде сегментов, которые доставляются в правильном порядке, с подтверждением получения и повторной передачей в случае потерь. Такой подход делает *TCP* особенно подходящим для задач, где важна целостность данных, например, при передаче изображений от клиента к управляющему узлу и от него к подчинённым узлам для обработки с использованием оператора Робертса. Надёжность *TCP* позволяет избежать искажений данных, что критически важно для сохранения качества изображений.

1.5.2 В отличие от *TCP*, протокол *UDP* не предусматривает установления соединения и работает по принципу передачи датаграмм. Данные отправляются без предварительного согласования, что обеспечивает минимальные задержки, но не гарантирует доставку или правильный порядок получения сообщений. Это делает *UDP* подходящим для задач, где скорость передачи приоритетнее надёжности, таких как отправка сообщений о прогрессе обработки от подчинённых узлов к клиенту через управляющий узел. Например, в *master-slave* системе *UDP* может использоваться для передачи информации о проценте выполненной обработки изображений, что позволяет клиенту получать оперативные обновления в реальном времени.

1.5.3 *TCP* и *UDP* поддерживают передачу данных в различных форматах, что делает их универсальными для распределённых систем. В случае *TCP* данные передаются в виде потока байтов, что позволяет обрабатывать файлы изображений, такие как *JPEG* или *PNG*, без потери их структуры. *UDP*, в свою очередь, передаёт данные в виде отдельных датаграмм, что подходит для отправки коротких сообщений, таких как уведомления о состоянии обработки. Оба протокола могут быть адаптированы для различных сценариев, включая передачу больших объёмов данных или небольших сообщений, что делает их перспективными для использования в *master-slave* архитектуре.

Комбинированное использование *TCP* и *UDP* в *master-slave* системе позволяет оптимизировать взаимодействие между компонентами. *TCP* обеспечивает надёжную передачу изображений для обработки, что критически важно для сохранения их целостности, а *UDP* позволяет оперативно информировать клиента о прогрессе выполнения задач. Такой подход делает систему гибкой и эффективной, сочетая надёжность и скорость в зависимости от требований конкретной задачи.

1.5.4 Помимо обработки изображений, *TCP* и *UDP* применяются в различных областях, таких как потоковая передача мультимедиа, системы мониторинга

и сетевые приложения. Например, в системах видеонаблюдения *TCP* может использоваться для передачи видеопотока, а *UDP* – для отправки метаданных о состоянии системы. В игровых приложениях *UDP* часто применяется для передачи данных об игровых событиях, где скорость важнее надёжности.

Альтернативные протоколы, такие как *HTTP* или *gRPC*, также могут быть рассмотрены для распределённых систем, но их использование менее распространено в задачах обработки изображений. *HTTP*, ориентированный на веб-приложения, предполагает большие накладные расходы из-за необходимости отправки заголовков с каждым запросом, что делает его менее эффективным для передачи больших файлов, таких как изображения. *gRPC*, в свою очередь, предоставляет высокопроизводительный фреймворк для вызова удалённых процедур, но требует сложной настройки, что может быть избыточным для задач, связанных с обработкой изображений. *TCP* и *UDP*, напротив, обеспечивают низкоуровневый доступ к сетевым операциям, что делает их более подходящими для *master-slave* системы.

В современных распределённых системах *TCP* и *UDP* остаются одними из наиболее распространённых протоколов благодаря их универсальности и эффективности. Их комбинированное использование позволяет создавать гибкие системы, способные обрабатывать большие объёмы данных и предоставлять оперативную информацию о состоянии обработки.

1.6 Оператор Робертса и его аналоги в обработке изображений

Для обработки изображений в распределённых системах, таких как *master-slave* архитектура, применяются различные алгоритмы выделения границ, одним из которых является оператор Робертса. Данный оператор используется для обнаружения границ на изображении путём вычисления градиента яркости в соседних пикселях, что позволяет выявлять контуры объектов. Однако существуют и другие алгоритмы, такие как операторы Собеля, Прюитт и Кэнни, которые также применяются для аналогичных задач. Эти методы имеют свои преимущества и недостатки, что делает их подходящими для различных сценариев обработки изображений.

Оператор Робертса представляет собой один из простейших методов выделения границ, основанный на использовании двух свёрточных ядер размером 2x2. Эти ядра анализируют изменения яркости в диагональных направлениях, что позволяет выявлять границы объектов на изображении. Простота вычислений делает оператор Робертса особенно подходящим для распределённых систем, где задачи обработки распределяются между подчинёнными узлами с использованием алгоритма *Round Robin*. В *master-slave* архитектуре, ориентированной на обработку изображений, оператор Робертса может быть применён для быстрого выделения границ, что минимизирует вычислительную нагрузку на каждый узел и сокращает общее время обработки.

В сравнении с оператором Робертса, оператор Собеля использует свёрточные ядра размером 3x3, что позволяет учитывать изменения яркости не только в

диагональных, но и в горизонтальном и вертикальном направлениях. Это делает оператор Собеля более точным в выявлении границ, особенно на изображениях с высоким уровнем шума или сложной текстурой. Однако увеличенный размер ядер приводит к большей вычислительной сложности, что может быть недостатком в распределённых системах с ограниченными ресурсами. Оператор Собеля часто применяется в задачах, где требуется более детализированное выделение границ, например, в системах компьютерного зрения или медицинской диагностики.

Оператор Прюитт, подобно оператору Собеля, также использует ядра размером 3x3, но с другим набором коэффициентов, что обеспечивает схожую точность при меньшей чувствительности к шуму. Данный метод может быть полезен в системах, где изображения содержат умеренный уровень шума, но его вычислительная сложность аналогична оператору Собеля, что делает его менее предпочтительным для задач, требующих высокой скорости обработки. В *master-slave* архитектуре оператор Прюитт может быть применён для обработки изображений, если требуется баланс между точностью и устойчивостью к шуму.

Оператор Кэнни представляет собой более сложный подход к выделению границ, включающий несколько этапов: сглаживание изображения для подавления шума, вычисление градиента, подавление немаксимумов и пороговую обработку для определения границ. Этот метод считается одним из наиболее точных, так как позволяет выявлять границы даже на изображениях с высоким уровнем шума или низким контрастом. Однако оператор Кэнни требует значительных вычислительных ресурсов, что может быть ограничивающим фактором в распределённых системах, где задачи распределяются между подчинёнными узлами. В таких системах применение оператора Кэнни может быть оправдано для обработки изображений высокого качества, например, в медицинских или научных приложениях.

В сравнении с оператором Робертса, операторы Собеля и Прюитт обеспечивают более точное выделение границ за счёт использования ядер большего размера, но требуют больше вычислительных ресурсов. Оператор Кэнни, в свою очередь, превосходит их по точности, но его сложность делает его менее подходящим для систем, где приоритет отдаётся скорости обработки. В *master-slave* архитектуре, использующей алгоритм *Round Robin* для распределения задач, оператор Робертса может быть предпочтительным благодаря своей простоте, что позволяет эффективно распределять задачи между подчинёнными узлами и минимизировать время обработки.

Применение операторов выделения границ, таких как Робертса, Собеля, Прюитт или Кэнни, зависит от требований конкретной задачи. В системах компьютерного зрения, например, для распознавания объектов или анализа сцен, оператор Кэнни может быть предпочтительным благодаря своей точности. В медицинской диагностике, где важна устойчивость к шуму, операторы Собеля или Прюитт могут обеспечивать более надёжные результаты. В задачах, где приоритет отдаётся скорости обработки, как в *master-slave* системах с ограниченными

ресурсами, оператор Робертса предоставляет оптимальный баланс между производительностью и качеством.

Перспективы использования оператора Робертса и его аналогов включают их интеграцию с современными технологиями обработки изображений. Например, комбинация оператора Робертса с методами машинного обучения может повысить точность выделения границ при сохранении вычислительной эффективности. В *master-slave* системах возможно внедрение гибридных подходов, где подчинённые узлы применяют различные операторы в зависимости от типа изображения или требований задачи. Кроме того, интеграция с облачными платформами позволяет масштабировать вычислительные ресурсы, что делает возможным использование более сложных операторов, таких как Кэнни, в системах с высокой нагрузкой.

Таким образом, оператор Робертса представляет собой эффективный инструмент для выделения границ в распределённых системах благодаря своей простоте и низкой вычислительной сложности. Его аналоги, такие как операторы Собеля, Прюитт и Кэнни, предоставляют более точные результаты, но требуют больших ресурсов, что делает их менее подходящими для систем, где приоритет отдаётся скорости. Выбор оператора зависит от требований задачи, и в *master-slave* архитектуре оператор Робертса может быть оптимальным решением для обработки изображений, обеспечивая баланс между производительностью и качеством.

2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОЙ ИНФОРМАЦИОННОЙ СИСТЕМЫ

2.1 Архитектура распределённой информационной системы

В рамках курсового проектирования разрабатывается программный комплекс для распределённой обработки изображений, реализующий выделение границ объектов с помощью оператора Робертса. Для реализации системы была выбрана клиент-серверная архитектура на основе модели распределённых вычислений *Master-Slave* (Мастер-Рабочий).

В основе архитектуры лежит централизованная модель координации, при которой весь процесс обработки управляется одним выделенным узлом-координатором (мастер-узлом), а непосредственные вычисления делегируются произвольному количеству рабочих узлов. Такая организация позволяет достигать практически линейного роста производительности при увеличении числа задействованных рабочих машин и обеспечивает простоту развёртывания и масштабирования системы в условиях локальной сети или облачной инфраструктуры.

Для реализации системы был выбран язык *C#* с использованием платформы *.NET* 8. В отличие от монолитных приложений, данная разработка представляет собой комплекс из трёх независимых типов приложений:

1) *clientApp* – графическое приложение на базе *Windows Presentation Foundation* (*WPF*);

2) *masterNode* – консольное приложение, выполняющее роль координатора и балансировщика нагрузки;

3) *slaveNode* – консольное приложение, выполняющее непосредственные вычисления.

Архитектура распределённой системы представлена на рисунке 2.1.

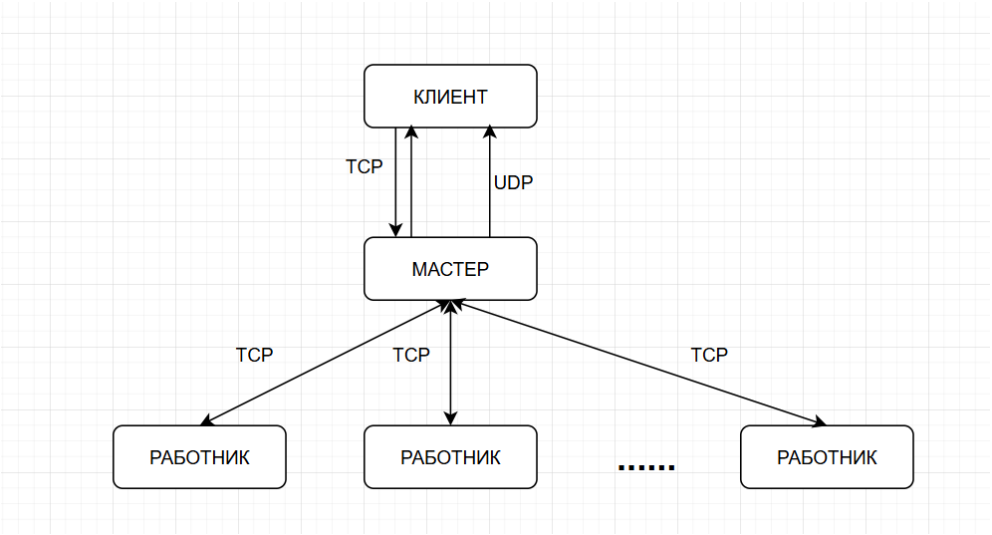

Рисунок 2.1 – Архитектура распределённой системы

Такое разделение позволяет запускать вычислительные узлы (*Slave*) на разных физических машинах в локальной сети, обеспечивая истинную параллельность вычислений.

Архитектура обладает выраженной горизонтальной масштабируемостью: добавление новых рабочих узлов выполняется без остановки системы и не требует изменения конфигурации клиента или мастера. Отказ любого из рабочих узлов автоматически обрабатывается мастером путём исключения его из списка активных участников и перераспределения невыполненных фрагментов между оставшимися узлами.

2.1.1 Клиентский узел (*ClientApp*). Клиентский узел представляет собой точку входа пользователя в систему. Графический интерфейс (*GUI*), разработанный с использованием паттерна *MVVM*, позволяет пользователю:

– выбирать изображения из файловой системы;

– визуализировать исходное изображение и результат обработки;

– инициировать отправку задачи на *Master*-узел;

– получать информацию о текущем статусе обработки в реальном времени.

Сетевое взаимодействие клиента:

В отличие от простейших реализаций, клиент использует гибридную схему сетевого взаимодействия:

TCP-канал: используется для передачи «тяжелых» данных (самого изображения) и получения результата. Это гарантирует, что биты изображения не будут потеряны или повреждены при передаче.

UDP-канал: используется для прослушивания сообщений о прогрессе обработки (например, «В очереди», «Обрабатывается», «Завершено»). Потеря одного пакета с прогрессом не критична для работы системы, поэтому использование *UDP* снижает нагрузку на основной канал связи.

Передача данных осуществляется не в текстовом формате, а с использованием кастомного бинарного протокола, что значительно уменьшает размер передаваемых пакетов и исключает накладные расходы на сериализацию строк. Структура полезной нагрузки для изображения представлена в таблице 2.1.

Таблица 2.1 – Структура бинарного сообщения с изображением

Поле	**Тип данных**	**Описание**
ImageId	*int*	Уникальный идентификатор задачи
NameLength	*int*	Длина имени файла в байтах
FileName	*string*	Имя файла изображения
Width	*int*	Ширина изображения в пикселях
Height	*int*	Высота изображения в пикселях
Format	*int*	Код формата
DataLength	*int*	Размер массива байтов изображения
PixelData	*byte[]*	Непосредственно байты изображения

2.1.2 Серверный узел координации (*MasterNode*). *Master*-узел является центральным элементом системы, отвечающим за маршрутизацию данных и балансировку нагрузки. Он функционирует как асинхронный многопоточный сервер. Функции *Master*-узла:

– *TCP*-сервер для *Slave*-узлов: принимает входящие подключения от вычислительных узлов, регистрирует их в пуле доступных ресурсов и отслеживает состояние соединения;

– *TCP*-сервер для клиентов: принимает задачи от клиентов. Для каждого клиента создается отдельный поток обработки, который удерживает соединение открытым до момента получения результата от *Slave*-узла;

– *UDP*-уведомления: асинхронная отправка датаграмм статуса на адрес клиента;

– планировщик задач: реализует логику распределения задач. Для балансировки нагрузки используется алгоритм *Round Robin*. Задачи от клиентов поступают в очередь. Планировщик хранит список подключенных *Slave*-узлов и индекс следующего узла. При поступлении задачи планировщик перебирает список *Slave*-узлов начиная с текущего индекса. Если выбранный *Slave* свободен, задача назначается ему, и индекс смещается. Если *Slave* занят, проверяется следующий. Такой подход обеспечивает равномерную загрузку вычислительных мощностей без простоев, даже если задачи поступают неравномерно.

2.1.3 *Slave*-узел – это консольное приложение, выполняющее ресурсоемкую операцию обработки изображения. При запуске узел инициирует *TCP*-соединение с *Master*-узлом и переходит в режим ожидания команд.

Алгоритм работы *Slave*-узла:

– чтение заголовка;

– получение данных: на основе длины из заголовка считывается тело сообщения с изображением;

– обработка: применяется оператор Робертса;

– возврат результата: обработанное изображение сериализуется в бинарный формат и отправляется обратно в тот же *TCP*-сокет.

Асинхронная архитектура – все операции ввода-вывода выполняются асинхронно с использованием *async*/await, что обеспечивает отзывчивость интерфейса и эффективное использование потоков.

Данная архитектура является оптимальной для демонстрации принципов распределённых вычислений, поскольку она легко масштабируется путём добавления дополнительных *Slave* узлов и обеспечивает наглядную визуализацию процесса распределённой обработки данных.

2.2 Алгоритм выделения границ оператором Робертса

Для обработки изображений был выбран алгоритм выделения границ на основе перекрёстного дифференциального оператора Робертса. Данный алгоритм не имеет смысла размещать на серверном *Master*-приложении, так как оно

выступает лишь в роли координатора, распределяющего задачи между вычислительными узлами и управляющего сетевыми потоками, но не занимается непосредственно ресурсоемкой обработкой растровых данных. Поэтому алгоритм реализации оператора Робертса будет размещён исключительно в классе вычислительного *Slave*-узла и инкапсулирован в сервисе *ImageProcessor*.

Когда вычислительный узел получает *TCP*-пакет с изображением от *Master*-узла, он десериализует данные, запускает метод выделения границ и возвращает результат в виде объекта сообщения. После завершения обработки узел выполняет кодирование полученного результата в формат *PNG* или *JPEG* для оптимизации сетевого трафика и отправляет обработанное изображение обратно через установленное *TCP*-соединение на *Master*-узел, который в свою очередь пересылает его соот-ветствующему клиенту.

Оператор Робертса является одним из старейших и наиболее быстрых алгоритмов выделения границ в компьютерном зрении. Основное преимущество данного метода заключается в компактности ядра свёртки (матрица размером 2×2) и высокой скорости вычислений по сравнению с более слож-ными операторами, такими как Собель или Кэнни. Это делает его идеальным выбором для дмонстрации распределённых вычислений, где критически важно минимизировать время обработки на одном узле для обеспечения высокой пропускной способности системы.

Алгоритм основан на аппроксимации модуля градиента изображения с использованием дискретных разностей по диагоналям. В реализованной системе обработка происходит в два этапа. Сначала исходное цветное изображение преобразуется в оттенки серого, так как выделение границ бази-руется на анализе перепадов яркости, а не цветности. Для этого используется стандартная формула пересчета яркости:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B , \qquad (2.1)$$

где *R, G, B* – значения красного, зелёного и синего каналов соответственно.

Далее алгоритм выполняет проход по изображению с использованием скользящего ядра размером 2×2 пикселя. В отличие от фильтров шумоподавления, где окно центрируется относительно пикселя, оператор Робертса использует текущий пиксель как левый верхний угол ядра. Ядра оператора представляют собой две матрицы свёртки, представленные на рисунке 2.2.

$$\begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$$

Рисунок 2.2 – Ядра свёртки

В программной реализации это сводится к вычислению разности значений яркости диагонально смежных пикселей. Для каждой точки изображения вычисляются градиенты, а результирующее значение пикселя (величина градиента) определяется как геометрическая длина вектора:

$$G = \sqrt{(Y_{x,y} - Y_{x+1,y+1})^2 + (Y_{x+1,y} - Y_{x,y+1})^2}\ , \qquad (2.2)$$

где x, y – координаты пикселя, а Y – значение яркости пикселя в точке (x, y).

После завершения обработки всех пикселей исходное и результирующее изображения разблокируются, освобождая системную память. Однако, для снижения нагрузки на пропускную способность сети, результирующий массив пикселей кодируется в эффективный формат (*PNG*) перед отправкой.

На рисунке 2.3 представлена блок-схема алгоритма выделения границ оператором Робертса.

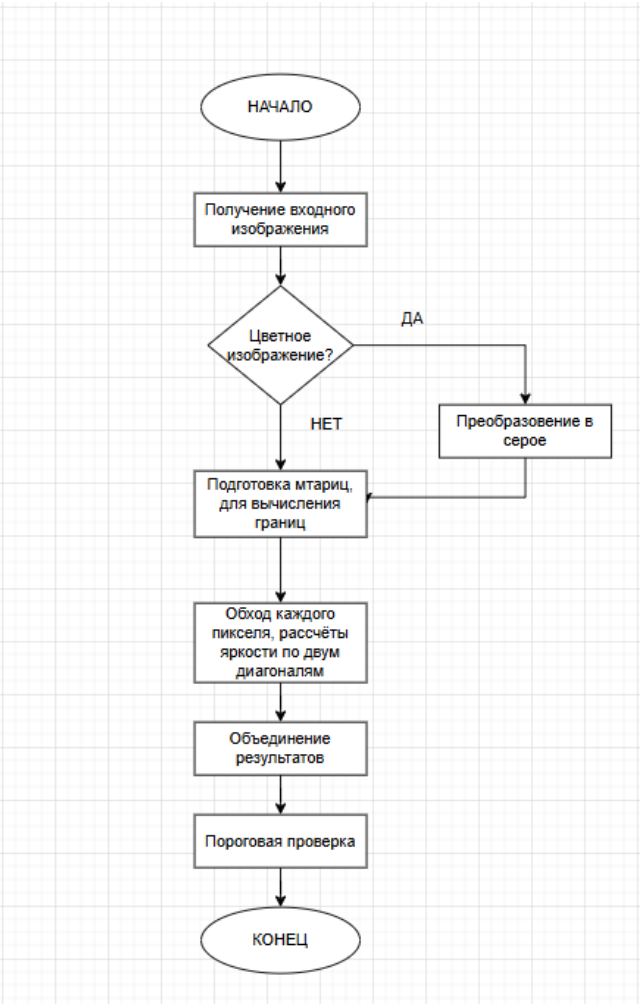

Рисунок 2.3 – Блок-схема алгоритма выделения границ оператором Робертса

Для повышения производительности вычислений был выбран параллельный способ обработки изображения. Изображение разбивается на строки, при

этом каждый поток обрабатывает одну строку изображения. Такой подход позволяет равномерно распределить вычислительную нагрузку между потоками и эффективно использовать многоядерные процессоры. Поскольку вычисление градиента для каждого пикселя зависит только от локального соседства, обработка отдельных строк может выполняться независимо, без необходимости синхронизации между потоками на этапе вычислений. Благодаря этому алгоритм эффективно справляется с выделением контуров на изображениях различных размеров.

2.3 Структура распределённой информационной системы

Разрабатываемый программный продукт представляет собой комплексное решение на платформе *.NET* 8, объединяющее три независимых исполняемых приложения: *ClientApp, MasterNode* и *SlaveNode*, а также общую библиотеку классов *Common*. Такая архитектура позволяет гибко настраивать развёртывание системы: клиентское приложение может работать на машине пользователя, мастер-узел – на выделенном сервере, а вычислительные узлы – на множестве доступных компьютеров в сети.

2.3.1 В основных моделях данных, расположенных в библиотеке *Common,* можно выделить класс *ImageMessage*, который является базовым контейнером для передачи данных по сети. В отличие от текстовых форматов, он спроектирован для бинарной передачи и содержит поле *ImageId* (уникальный временной идентификатор), поля *Width* и *Height* для хранения размеров, поле *Format* для кодировки, имя файла *FileName* и массив байтов *ImageData* с полезной нагрузкой.

Перечисление *MessageType* определяет семантику передаваемых пакетов и включает пять типов: *ClientToMasterImage* для входящих задач, *MasterToSlaveTask* для распределения работы, *SlaveToMasterResult* для возврата результата, *MasterToClientResult* для финального ответа и *MasterToClientProgress* для *UDP*-уведомлений.

Класс *MessageSerializer* заменяет стандартные средства сериализации. Он предоставляет статические методы *SerializeImageMessage* и *DeserializeImageMessage*, использующие *BinaryWriter* и *BinaryReader* для формирования плотного байтового потока без накладных расходов, свойственных *JSON*-формату.

2.3.2 Основным алгоритмическим сервисом выступает статический класс *RobertsOperator* (в пространстве *Common.ImageProcessing*). Метод *ApplyRobertsOperatorFast* реализует алгоритм выделения границ, используя прямой доступ к памяти через механизм *LockBits* и небезопасный код (*unsafe*). Вспомогательные методы *BytesToBitmap* и *BitmapToBytes* обеспечивают трансформацию между сетевым представлением данных и графическими объектами.

Критически важным для сетевой подсистемы является класс *NetworkExtensions* (в пространстве *Common.Networking*). Он реализует метод расширения *ReadExactAsync*, который гарантирует чтение строго определённого количества байт из *NetworkStream*. Это решает проблему фрагментации *TCP*-пакетов, предотвращая ошибки десериализации. В клиентской части класс *ClientService* инкапсулирует гибридную сетевую логику: надёжную отправку задач по *TCP* и параллельное прослушивание *UDP*-порта для обновления прогресса в реальном времени.

2.3.3 В основных узлах системы центральное место занимает *MasterNode*. Его ядром является класс *MasterServer*, запускающий два независимых *TCP*-слушателя. Класс *TaskScheduler* управляет потокобезопасной очередью задач *ConcurrentQueue* и реализует алгоритм балансировки *RoundRobin*. Класс *SlaveHandler* представляет собой цифровую проекцию подключенного вычислителя, отслеживая его доступность и асинхронно маршрутизируя задачи. Класс *ProgressSender* отвечает за отправку *UDP*-пакетов прогресса.

Класс *SlaveNode* реализует вычислительный узел. При запуске класс *SlaveWorker* управляет жизненным циклом приложения, инициируя подключение к мастеру. Класс *SlaveConnectionHandler* (или логика внутри воркера) обеспечивает надёжный приём данных с использованием двухэтапного чтения заголовка и тела сообщения. Класс *ImageProcessor* выступает связующим звеном, вызывая методы оператора Робертса и формируя ответные пакеты.

2.3.4 В пользовательском интерфейсе (*ClientApp*) класс *ClientViewModel* реализует паттерн *MVVM* и управляет состоянием представления. Он содержит свойства конфигурации подключения, свойства данных и состояния интерфейса. Команды *SelectImageCommand* и *SendTaskCommand*, реализованные через *RelayCommand*, обеспечивают взаимодействие пользователя с системой.

Класс *MainWindow* содержит минимальную логику инициализации, а разметка в *MainWindow.xaml* описывает визуальную структуру окна. Конвертер *InverseBooleanConverter* используется для блокировки элементов управления во время выполнения задачи, а *BooleanToVisibilityConverter* управляет отображением индикаторов загрузки.

Вышеописанная архитектура приложения обеспечивает высокую степень модульности и горизонтальную масштабируемость, позволяя изменять алгоритмы обработки или добавлять новые вычис-лительные узлы без остановки работы системы. Использование бинарного протокола поверх *TCP* гарантирует целостность данных и высокую производительность.

3 ВЕРИФИКАЦИЯ И АПРОБАЦИЯ РАСПРЕДЕЛЁННОЙ ИНФОРМАЦИОННОЙ СИСТЕМЫ

3.1 Модульное тестирование распределённой информационной системы

Для обеспечения корректной работы разработанного программного программного комплекса была применена методика модульного тестирования. Основная цель модульного тестирования заключается в проверке отдельных компонентов системы в изоляции, что позволяет убедиться в их правильной работе и соответствии функциональным требованиям. Такой подход способствует выявлению возможных ошибок на ранних стадиях разработки и обеспечивает основу для безопасного внесения изменений и расширения функциональности приложения в дальнейшем.

В рамках проекта был создан отдельный проект для тестирования, интегрированный в решение *Visual Studio*, с использованием фреймворка *xUnit*. Для проверки подвергался ключевой компонент системы, отвечающий за обработку изображений – класс, реализующий оператор Робертса на стороне *Slave*-узла. Выбор именно этого компонента для модульного тестирования обусловлен его критической ролью в процессе обработки изображений, а также тем, что его работа не зависит от сетевого взаимодействия или пользовательского интерфейса, что позволяет проводить проверку в изолированной среде.

3.1.1 Для класса, реализующего оператор Робертса, был подготовлен развернутый набор тестовых случаев, обеспечивающий всестороннюю проверку корректности работы алгоритма в различных условиях применения. Тестирование включало как типичные сценарии, так и граничные ситуации: обработку однотонных изображений, небольших матриц пикселей, изображений с ярко выраженными градиентами и сложными переходами. Отдельное внимание уделялось корректной обработке крайних строк и столбцов, где алгоритм должен формировать чёрные границы в соответствии со спецификой оператора Робертса. Проверялись точность вычисления градиентов, соблюдение допустимого диапазона яркости пикселей и стабильность поведения при минимальных и максимально возможных значениях. Такой комплексный подход позволил убедиться в надёжности и корректной реализации алгоритма.

Результаты выполнения тестов подтвердили устойчивую и корректную работу алгоритма во всех рассмотренных сценариях, включая обработку изображений различного размера, формата и структуры пикселей. Алгоритм надёжно воспроизводит ожидаемое поведение даже в граничных ситуациях, что указывает на его стабильность и корректную реализацию. Примеры успешного прохождения тестов в среде *Visual Studio* представлены на рисунке 3.1 и демонстрируют полноту и точность проверки.

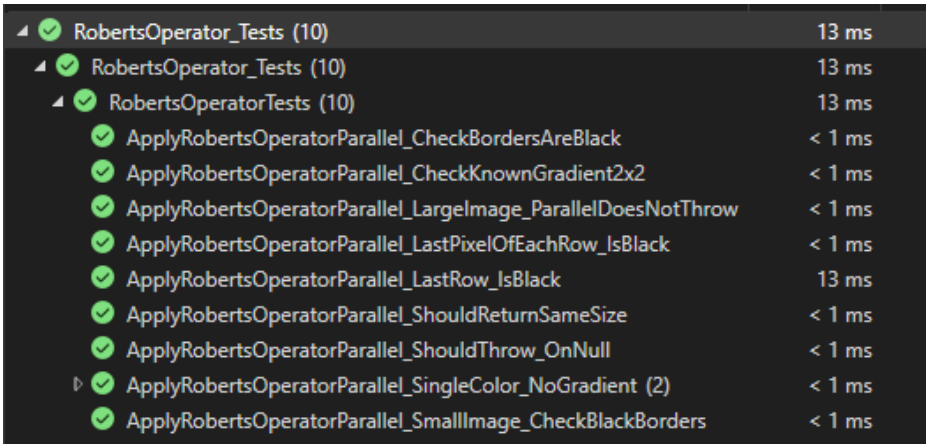

Рисунок 3.1 – Результаты модульного тестирования класса, реализующего оператор Робертса

Применение модульного тестирования позволило подтвердить корректность работы ключевого алгоритма системы в изолированной среде. Это гарантирует, что возможные ошибки в процессе функционирования всего программного комплекса не будут связаны с базовой логикой вычисления градиентов и обработки изображений.

3.2 Нагрузочное тестирование распределённой информационной системы

Нагрузочное тестирование является важным этапом верификации распределённой системы, так как позволяет оценить её производительность, масштабируемость и эффективность использования вычислительных ресурсов. Целью данного тестирования является измерение времени обработки стандартного набора изображений при различном количестве активных *Slave*-узлов, а также расчет ключевых метрик производительности, таких как ускорение и эффективность параллельного выполнения задач.

Для проведения нагрузочного тестирования был подготовлен набор из 100 изображений. Тестирование проводилось в несколько этапов: сначала система запускалась с одним *Slave*-узлом, после чего последовательно увеличивалось количество *Slave*-узлов до двух и четырёх. В каждом случае фиксировалось общее время, затраченное на обработку всего набора изображений.

3.2.1 Замеры времени проводились с момента отправки первого изображения *Master*-узлу до получения всех результатов на клиентской стороне. Для повышения точности результаты измерялись несколько раз, после чего вычислялось среднее значение. Все компоненты системы запускались на одной физической машине, что минимизировало влияние сетевых задержек на измерения.

Для начала сделаем замеры для однопоточной версии алгоритма Робертса, затем сравним их с многопоточной версией версией. Результаты замеров среднего времени выполнения задачи представлены в таблице 3.1.

Таблица 3.1 – Результаты нагрузочного тестирования

Количество Slave-узлов (n)	Среднее время выполнения (Tn), мс
1	5888
6	5030
12	4938

На основе полученных данных были рассчитаны показатели ускорения для различных конфигураций. Ускорение отражает, во сколько раз параллельная обработка выполняется быстрее последовательного выполнения и рассчитывается по формуле 3.1:

$$S_n = \frac{T_1}{T_n}, \quad (3.1)$$

ускорение для потоков: $S6 \approx 1.17$, $S12 \approx 1.19$.

Результаты показывают, что с увеличением количества потоков общее время обработки сокращается. Однако ускорение не является линейным, что объясняется наличием последовательных участков кода и накладных расходов на управление очередью задач и сетевое взаимодействие.

3.2.2 Эффективность использования вычислительных ресурсов определяется отношением ускорения к количеству узлов и рассчитывается по формуле 3.2. Идеальное значение – 1 (или 100%).

$$E_n = \frac{S_n}{n}, \quad (3.2)$$

- эффективность для 1-го узла: E1 = 1 (100%);
- эффективность для 2-х потоков: E6 = 1.17 / 6 = 0.2 (20%);
- эффективность для 4-х узлов: E12 = 1.19 / 12 = 0.1 (10%);

Снижение эффективности при увеличении числа потоков является естественным явлением, связанным с накладными расходами на управление задачами и передачу данных. Тем не менее, показатели свидетельствуют о высокой масштабируемости и эффективном распараллеливании вычислений. Графическое представление зависимости времени от количества потоков представлено на рисунке 3.2.

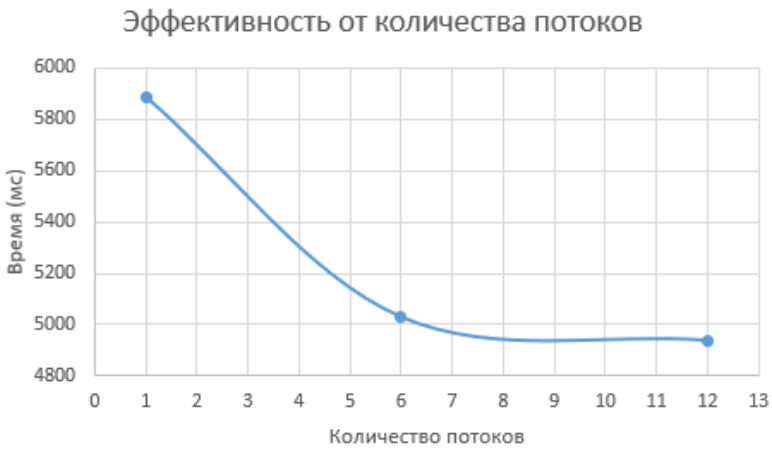

Рисунок 3.2 – Зависимость времени от количества потоков

3.2.3 Для более детального анализа производительности можно определить долю последовательной части алгоритма. В рассматриваемой системе последовательные операции включают: чтение изображений на клиентской стороне, передачу данных от клиента к *Master*-узлу и далее к *Slave*-узлам, а также финальную агрегацию результатов на *Master*-узле и отправку их обратно клиенту. Параллельная часть – это непосредственное применение оператора Робертса к изображениям на *Slave*-узлах.

На основе закона Амдала (формула 3.3) можно оценить долю последовательной работы:

$$S_n = \frac{1}{\mathrm{a} + \frac{1-\mathrm{a}}{\mathrm{n}}}, \tag{3.3}$$

где n – количество потоков, a – доля последовательной части программы.

Исходя из ускорения для шести потоков, доля последовательной части составляет:

$$1.17 = \frac{1}{\mathrm{a} + \frac{1-\mathrm{a}}{6}} \Rightarrow \mathrm{a} \approx 0.83,$$

таким образом, примерно 83% времени затрачивается на последовательные операции, а 17% – на параллельную обработку изображений.

Предельное ускорение при неограниченном количестве потоков вычисляется по формуле 3.4:

$$S_\infty = \frac{1}{\mathrm{a}} = \frac{1}{0.83} = 1.2, \tag{3.4}$$

где a – доля последовательной части программы.

Это означает, что максимальное ускорение для данной архитектуры не превысит 1.2 раз.

Прогноз по закону Густафсона-Барсиса позволяет оценить ускорение при масштабируемой задаче, когда увеличивается и размер задачи при росте числа потоков. Формула 3.5:

$$S_n'' = n + (1 - n) * a;\ S_6'' = 6 + (1 - 6) * 0.17 = 1.85\ , \qquad (3.5)$$

где n – количество потоков, a – доля последовательной части программы.

Это показывает, что система с шестью потоками способна обработать задачи большего объема почти в 1.85 раза быстрее, чем система с одним потоком, что отражает преимущества распределённой архитектуры.

Проведённое нагрузочное тестирование подтвердило высокую производительность и масштабируемость разработанной *Master-Slave* системы для применения оператора Робертса к изображениям. Результаты демонстрируют, что добавление вычислительных ресурсов сокращает время выполнения задач и эффективно распределяет нагрузку между узлами.

3.3 Пользовательский интерфейс распределённой информационной системы

Для взаимодействия пользователя с распределённой системой обработки изображений был разработан клиентский интерфейс на базе *Windows Presentation Foundation* (*WPF*). Клиентское приложение выполняет функции управляющей панели, позволяющей формировать набор изображений, инициировать их отправку на *Master*-узел, отслеживать прогресс распределённой обработки и просматривать полученные результаты.

Главное окно интерфейса содержит несколько логически разделённых зон, обеспечивающих последовательный рабочий процесс – от выбора изображений до визуального анализа результата применения оператора Робертса.

3.3.1 Верхняя часть окна содержит основные элементы управления:

– кнопку «Выбрать изображения», позволяющую добавить произвольное количество файлов через стандартный диалог выбора;

– кнопку «Отправить все», которая активируется после формирования списка изображений и инициирует их отправку на *Master*-узел;

– кнопку «Очистить», удаляющую текущий список и сбрасывающую состояние интерфейса.

Состояние кнопок динамически изменяется в зависимости от этапа обработки, что предотвращает ошибочные действия пользователя.

3.3.2 Основная часть интерфейса представляет собой вертикально прокручиваемую область, в которой каждый элемент соответствует отдельному изображению.

Каждая запись включает три логических блока:

1) исходное изображение – миниатюра выбранного пользователем файла;

2) результат обработки – визуализация изображения после применения оператора Робертса, полученная от *Slave*-узла;

3) индикатор состояния, отображающий:

– «В очереди» – изображение ждёт своей очереди;

– «Обрабатывается» – *Slave*-узел выполняет вычисления;

– «Завершено (100%)» – результат успешно получен.

Каждый элемент снабжён подписью с исходным именем файла, что упрощает идентификацию изображений в большом наборе.

Интерфейс клиентского приложения разработан с акцентом на удобство пользователя и наглядность процесса обработки изображений. Главное окно клиента, представленное на рисунке Д.1, содержит все необходимые элементы управления для работы с системой: панели для выбора изображений, отображения текущего статуса обработки и визуализации результатов. В верхней части окна располагаются кнопки для добавления и удаления изображений, запуска процесса обработки и мониторинга прогресса, а центральная область предназначена для наглядного отображения пар «исходник – результат». Такое расположение элементов позволяет пользователю сразу видеть, какие изображения были выбраны, как они обрабатываются и какие результаты получены, обеспечивая интуитивно понятное взаимодействие с программой.

Процесс работы с системой начинается с выбора изображений для обработки. Пользователь может добавить одно или несколько изображений через соответствующую кнопку в главном окне. После этого выбранные изображения отображаются в интерфейсе, как показано на рисунке Д.2 и Д.3, где каждая миниатюра сопровождается краткой информацией о файле. Такой подход позволяет убедиться, что все нужные изображения добавлены корректно и готовы к отправке на *Master*-узел.

После подтверждения выбора изображений начинается этап передачи данных на обработку. На рисунке Д.4 показан процесс отображения статуса изображений при отправке их на *Master*-узел и последующего распределения между *Slave*-узлами. Для каждой пары «исходник – результат» отображается текущий статус: в очереди, обработка или завершение. Такая визуализация позволяет пользователю отслеживать прогресс в реальном времени и понимать, какие изображения уже обработаны, а какие находятся в очереди.

По завершении обработки результаты отображаются непосредственно рядом с исходными изображениями, как показано на рисунке Д.5. Каждая пара «исходник – результат» расположена таким образом, что визуальное сравнение происходит мгновенно: пользователь может сразу оценить качество работы оператора Робертса, замечая даже незначительные отличия между оригиналом и обработанным изображением.

Для обеспечения контроля корректности выполнения и упрощения отладки в программной реализации предусмотрена система логирования как для *Master*-узла, так и для *Slave*-узлов. В логах фиксируются ключевые этапы работы

приложения: инициализация соединений, начало и завершение обработки, а также возможные ошибки и исключительные ситуации. Примеры журналов работы *Master*- и *Slave*-узлов приведены на рисунках Д.6 и Д.7 соответственно. Использование логирования позволяет наглядно отслеживать процесс распределённых вычислений и анализировать производительность и устойчивость системы.

Благодаря такой организации интерфейса взаимодействие с системой становится максимально удобным. Пользователь получает полное представление о процессе обработки изображений, может оперативно выявлять ошибки или некорректные результаты и проводить быстрый анализ полученных данных. Визуальная наглядность и последовательность отображения информации существенно повышают эффективность работы с программой и ускоряют процесс анализа изображений.

3.3.3 Для улучшения восприятия и повышения удобства работы с системой были применены дополнительные визуальные и интерфейсные улучшения:

– изображения автоматически масштабируются с сохранением пропорций, что позволяет комфортно просматривать результаты обработки независимо от их исходного разрешения;

– каждая пара оригинала и результата оформлена в виде независимого визуального блока с рамкой, что делает интерфейс более структурированным и помогает быстро сопоставлять исходные и обработанные данные;

– используется адаптивная разметка, корректно отображающая элементы при увеличении или уменьшении окна клиента, что обеспечивает одинаково удобную работу как на больших экранах, так и при более компактном отображении.

Такая организация интерфейса позволяет не только эффективно обрабатывать большие наборы изображений, но и делает работу с системой максимально удобной и наглядной. Пользователь в реальном времени видит состояние каждого узла, может контролировать прогресс выполнения распределённого алгоритма и быстро реагировать на любые изменения в процессе обработки. Благодаря этому система выглядит более профессиональной, надёжной и современной, обеспечивая комфорт при работе даже с действительно крупными объёмами данных.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была разработана распределённая система обработки изображений, построенная по архитектуре *master-slave* с использованием протокола *TCP*, что позволило эффективно применять оператор Робертса для выделения границ на визуальных данных. Система обеспечивает параллельное распределение задач между *slave*-узлами с применением алгоритма *Round Robin*, что способствовало равномерной загрузке вычислительных ресурсов и повышению производительности обработки. Клиентская часть приложения реализована с возможностью передачи набора изображений master-узлу, получения результатов обработки и мониторинга прогресса в режиме реального времени с использованием протокола *UDP*, что делает работу системы более наглядной и интерактивной.

В процессе проектирования были сформированы функциональные схемы распределённой системы, определена структура основных модулей и характер их взаимодействия. Разработанное программное решение способно корректно обрабатывать большие объёмы визуальных данных в распределённой среде, что подтверждено результатами тестирования. Проверка корректности работы алгоритма Робертса в условиях распределённой обработки продемонстрировала устойчивость системы и эффективность выбранного подхода к организации вычислительного процесса.

В результате работы достигнуты все поставленные цели, включая создание программного комплекса для параллельной обработки изображений с применением классического метода выделения границ. Разработанная система обеспечивает масштабируемость, надёжность и возможность визуального контроля процесса обработки, что делает её пригодной для практического применения в задачах компьютерного зрения и автоматизированного анализа изображений.

Реализация проекта подтверждает актуальность применения распределённых информационных технологий для обработки больших объёмов визуальных данных. Решённая задача демонстрирует, что классические алгоритмы обработки изображений, такие как оператор Робертса, могут быть эффективно интегрированы в современные распределённые вычислительные среды, обеспечивая высокую скорость и точность анализа. Полученные результаты могут служить основой для дальнейшего расширения функционала системы и её применения в практических проектах, где важна оперативная обработка изображений и надёжность вычислительного процесса.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум Э., ван Стен М. Распределённые системы. Принципы и парадигмы. – СПб.: Питер, 2018. – 512 с.
2. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2019. – 352 с.
3. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. – М.: Русская Редакция, 2017. – 896 с.
4. Куроуз Дж., Росс К. Компьютерные сети. Нисходящий подход. – М.: Эксмо, 2016. – 912 с.
5. Албахари Дж. C# 9.0. Справочник. Полное описание языка. – СПб.: Питер, 2022. – 1040 с.
6. Parallel Класс. – Электрон. данные. – Режим доступа: https://learn.microsoft.com/ru-u/dotnet/api/system.threading.tasks.parallel?view=net-9.0. – Дата доступа: 22.11.2025.
7. ThreadPool Класс. – Электрон. данные. – Режим доступа: http://learn.microsoft.com/ru-ru/dotnet/api/system.threading.threadpool?view=net-8.0. – Дата доступа: 22.11.2025.
8. Алгоритм Робертса. – Электрон. данные. – Режим доступа: https://compgraph.tpu.ru/roberts.html. – Дата доступа: 22.11.2025.
9. Master Slave Technology Explained: A Guide for Industrial Automation. – Электрон. данные. – Режим доступа: https://www.rtautomation.com/rtas-blog/master-slave-technology/?srsltid=AfmBOoq4STbbq3bF31Zk7EITpyTz8H5SZf-gq5qDMY49_FoxyjjAUZ1S. – Дата доступа: 22.11.2025.
10. Введение в параллельные вычисления. – Электрон. данные. – Режим доступа: https://hpc.hse.ru/mirror/pubs/share/966355112.pdf. – Дата доступа: 22.11.2025.

ПРИЛОЖЕНИЕ А
(обязательное)

Листинг программы

Program.cs

```
namespace SlaveNode
{
   class Program
   {
      static async Task Main(string[] args)
      {
         Console.WriteLine("====================================");
         Console.WriteLine("   SLAVE NODE - Обработчик изображений");
         Console.WriteLine("====================================\n");

         string slaveName = "Slave-1";
         string masterHost = "127.0.0.1";
         int masterPort = 5000;

         if (args.Length >= 1)
         {
            slaveName = args[0];
         }
         if (args.Length >= 2)
         {
            masterHost = args[1];
         }
         if (args.Length >= 3)
         {
            if (int.TryParse(args[2], out int port))
            {
               masterPort = port;
            }
         }

         Console.WriteLine($"Имя узла: {slaveName}");
         Console.WriteLine($"Master адрес: {masterHost}:{masterPort}");
         Console.WriteLine();

         // Создаём CancellationToken для корректной остановки
         CancellationTokenSource cts = new CancellationTokenSource();

         Console.CancelKeyPress += (sender, e) =>
         {
            e.Cancel = true;
            Console.WriteLine("\n\nПолучен сигнал остановки...");
            cts.Cancel();
         };

         SlaveWorker worker = new(slaveName, masterHost, masterPort);

         try
         {
            await worker.StartAsync(cts.Token);
         }
         catch (OperationCanceledException)
         {
            Console.WriteLine("Работа прервана пользователем.");
         }
```

```
            catch (Exception ex)
            {
                Console.WriteLine($"Критическая ошибка: {ex.Message}");
            }

            Console.WriteLine("\nНажмите любую клавишу для выхода...");
            Console.ReadKey();
        }
    }
}
```

ImageProcessor.cs

```
using System.Drawing;
using System.Drawing.Imaging;
using Common.ImageProcessing;
using Common.Messages;

namespace SlaveNode
{
    /// <summary>
    /// Обработчик изображений на Slave-узле
    /// </summary>
    public class ImageProcessor
    {
        private readonly string _slaveName;

        public ImageProcessor(string slaveName)
        {
            _slaveName = slaveName;
        }

        /// <summary>
        /// Обрабатывает изображение, применяя оператор Робертса
        /// </summary>
        public ImageMessage ProcessImage(ImageMessage inputMessage)
        {
            try
            {
                Console.WriteLine($"[{_slaveName}] Начало обработки изображения ID: {inputMessage.ImageId}, имя:
{inputMessage.FileName}");

                // Конвертируем байты в Bitmap
                Bitmap sourceImage = RobertsOperator.BytesToBitmap(inputMessage.ImageData);

                Console.WriteLine($"[{_slaveName}] Размер изображения:
{sourceImage.Width}x{sourceImage.Height}");

                Bitmap processedImage;

                processedImage = RobertsOperator.ApplyRobertsOperatorParallel(sourceImage);

                // Определяем формат для сохранения
                ImageFormat format = GetImageFormat(inputMessage.Format);

                // Конвертируем обратно в байты
                byte[] resultBytes = RobertsOperator.BitmapToBytes(processedImage, format);

                Console.WriteLine($"[{_slaveName}] Обработка завершена. Размер результата: {resultBytes.Length}
байт");

                // Создаём результирующее сообщение
                var resultMessage = new ImageMessage(
```

```
                inputMessage.ImageId,
                $"processed_{inputMessage.FileName}",
                processedImage.Width,
                processedImage.Height,
                inputMessage.Format,
                resultBytes
            );

            // Освобождаем ресурсы
            sourceImage.Dispose();
            processedImage.Dispose();

            return resultMessage;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[{_slaveName}] ОШИБКА при обработке изображения: {ex.Message}");
            throw;
        }
    }

    /// <summary>
    /// Получает ImageFormat по коду формата
    /// </summary>
    private ImageFormat GetImageFormat(int formatCode)
    {
        return formatCode switch
        {
            1 => ImageFormat.Png,
            2 => ImageFormat.Jpeg,
            3 => ImageFormat.Bmp,
            _ => ImageFormat.Png // По умолчанию PNG
        };
    }
  }
}
```

SlaveWorker.cs

```
using System.Net.Sockets;
using Common.Messages;

namespace SlaveNode
{
  /// <summary>
  /// Рабочий процесс Slave-узла
  /// </summary>
  public class SlaveWorker
  {
    private readonly string _slaveName;
    private readonly string _masterHost;
    private readonly int _masterPort;
    private readonly ImageProcessor _imageProcessor;
    private TcpClient _client;
    private NetworkStream _stream;
    private bool _isRunning;

    public SlaveWorker(string slaveName, string masterHost, int masterPort)
    {
        _slaveName = slaveName;
        _masterHost = masterHost;
        _masterPort = masterPort;
        _imageProcessor = new ImageProcessor(slaveName);
```

```
}

/// <summary>
/// Запускает Slave-узел
/// </summary>
public async Task StartAsync(CancellationToken cancellationToken)
{
    _isRunning = true;
    Console.WriteLine($"[{_slaveName}] Запуск Slave-узла...");

    while (_isRunning && !cancellationToken.IsCancellationRequested)
    {
        try
        {
            // Подключаемся к Master
            await ConnectToMasterAsync();

            Console.WriteLine($"[{_slaveName}] Успешно подключен к Master ({_masterHost}:{_masterPort})");
            Console.WriteLine($"[{_slaveName}] Ожидание задач...\n");

            // Главный цикл обработки
            await ProcessTasksAsync(cancellationToken);
        }
        catch (SocketException ex)
        {
            Console.WriteLine($"[{_slaveName}] Ошибка подключения к Master: {ex.Message}");
            Console.WriteLine($"[{_slaveName}] Повторная попытка через 5 секунд...\n");
            await Task.Delay(5000, cancellationToken);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[{_slaveName}] Неожиданная ошибка: {ex.Message}");
            await Task.Delay(2000, cancellationToken);
        }
        finally
        {
            DisconnectFromMaster();
        }
    }

    Console.WriteLine($"[{_slaveName}] Slave-узел остановлен.");
}

/// <summary>
/// Подключается к Master-узлу
/// </summary>
private async Task ConnectToMasterAsync()
{
    _client = new TcpClient();
    _client.NoDelay = true;
    await _client.ConnectAsync(_masterHost, _masterPort);
    _stream = _client.GetStream();
}

/// <summary>
/// Отключается от Master-узла
/// </summary>
private void DisconnectFromMaster()
{
    _stream?.Close();
    _client?.Close();
    Console.WriteLine($"[{_slaveName}] Отключен от Master");
```

```
    }

    private async Task ProcessTasksAsync(CancellationToken cancellationToken)
    {
        while (!cancellationToken.IsCancellationRequested)
        {
            try
            {
                byte[] header = new byte[8];
                int read = await ReadExactAsync(_stream, header, 0, 8, cancellationToken);
                if (read == 0)
                {
                    Console.WriteLine($"[{_slaveName}] Master закрыл соединение. Переподключаемся...");
                    break;
                }

                int messageType = BitConverter.ToInt32(header, 0);
                int payloadLength = BitConverter.ToInt32(header, 4);

                if (payloadLength < 0)
                    throw new Exception($"Некорректная длина payload: {payloadLength}");


                byte[] payload = new byte[payloadLength];
                int readPayload = await ReadExactAsync(_stream, payload, 0, payloadLength, cancellationToken);
                if (readPayload < payloadLength)
                {
                    Console.WriteLine($"[{_slaveName}] Payload не полный");
                    break;
                }

                ImageMessage taskMessage = MessageSerializer.DeserializeImageMessage(payload, messageType, pay-
loadLength);
                ImageMessage result = _imageProcessor.ProcessImage(taskMessage);

                byte[] resultData = MessageSerializer.SerializeImageMessage(MessageType.SlaveToMasterResult, result);
                await _stream.WriteAsync(resultData, cancellationToken);
                await _stream.FlushAsync();
            }
            catch (Exception ex)
            {
                Console.WriteLine($"[{_slaveName}] Ошибка обработки задачи: {ex.Message}");
                DisconnectFromMaster();
                break;
            }
        }
    }

    /// <summary>
    /// Читает точное количество байт из потока
    /// </summary>
    private async Task<int> ReadExactAsync(NetworkStream stream, byte[] buffer, int offset, int count, Cancella-
tionToken cancellationToken)
    {
        int totalRead = 0;

        while (totalRead < count)
        {
            int read = await stream.ReadAsync(buffer, offset + totalRead, count - totalRead, cancellationToken);

            if (read == 0)
                return totalRead;
```

```
            totalRead += read;
        }

        return totalRead;
    }
  }
}
```

MasterServer.cs

```
using System.Net;
using System.Net.Sockets;
using Common.Messages;

namespace MasterNode
{
  /// <summary>
  /// Главный сервер, управляющий TCP-подключениями от клиентов и Slave-узлов.
  /// </summary>
  public class MasterServer
  {
    private readonly int _slavePort;
    private readonly int _clientPort;
    private TcpListener _slaveListener;
    private TcpListener _clientListener;
    private readonly TaskScheduler _scheduler;
    private readonly ProgressSender _progressSender;
    private bool _isRunning;

    public MasterServer(int slavePort, int clientPort)
    {
      _slavePort = slavePort;
      _clientPort = clientPort;
      _progressSender = new ProgressSender();
      _scheduler = new TaskScheduler(_progressSender);
    }

    /// <summary>
    /// Запускает Master-сервер.
    /// </summary>
    public async Task StartAsync(CancellationToken cancellationToken)
    {
      _isRunning = true;

      // Запуск слушателя для Slave-узлов
      _slaveListener = new TcpListener(IPAddress.Any, _slavePort);
      _slaveListener.Start();
      Console.WriteLine($"[MasterTCP] Сервер для Slave запущен на порту {_slavePort}...");

      // Запуск слушателя для Клиентов
      _clientListener = new TcpListener(IPAddress.Any, _clientPort);
      _clientListener.Start();
      Console.WriteLine($"[MasterTCP] Сервер для Клиентов запущен на порту {_clientPort}...");

      Task slaveTask = ListenForSlavesAsync(cancellationToken);
      Task clientTask = ListenForClientsAsync(cancellationToken);

      try
      {
        await Task.WhenAny(slaveTask, clientTask);
      }
      catch (OperationCanceledException)
```

```
    {

    }
    catch (Exception ex)
    {
        Console.WriteLine($"[Master] Критическая ошибка сервера: {ex.Message}");
    }
    finally
    {
        Stop();
    }
}

/// <summary>
/// Ожидает и обрабатывает подключения от Slave-узлов.
/// </summary>
private async Task ListenForSlavesAsync(CancellationToken cancellationToken)
{
    while (_isRunning && !cancellationToken.IsCancellationRequested)
    {
        try
        {
            TcpClient client = await _slaveListener.AcceptTcpClientAsync(cancellationToken);

            // Создаем обработчик для нового Slave
            SlaveHandler slaveHandler = new SlaveHandler(client, _scheduler);
            _scheduler.AddSlave(slaveHandler);

            // Запускаем асинхронное прослушивание Slave-узла
            Task.Run(() => slaveHandler.StartListeningAsync(cancellationToken));
        }
        catch (OperationCanceledException)
        {
            break;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[MasterTCP-Slave] Ошибка при приеме Slave: {ex.Message}");
        }
    }
}

/// <summary>
/// Ожидает и обрабатывает подключения от Клиентов.
/// </summary>
private async Task ListenForClientsAsync(CancellationToken cancellationToken)
{
    while (_isRunning && !cancellationToken.IsCancellationRequested)
    {
        try
        {
            TcpClient client = await _clientListener.AcceptTcpClientAsync(cancellationToken);

            // Обрабатываем клиента в отдельном потоке
            Task.Run(() => HandleClientConnectionAsync(client, cancellationToken));
        }
        catch (OperationCanceledException)
        {
            break;
        }
        catch (Exception ex)
        {
```

```
                Console.WriteLine($"[MasterTCP-Client] Ошибка при приеме Клиента: {ex.Message}");
            }
        }
    }

    /// <summary>
    /// Обрабатывает входящее сообщение от клиента (задачу).
    /// </summary>
    private async Task HandleClientConnectionAsync(TcpClient tcpClient, CancellationToken cancellationToken)
    {
        NetworkStream stream = tcpClient.GetStream();
        var clientUdpEndpoint = new IPEndPoint(IPAddress.Loopback, 6000);

        try
        {
            byte[] header = new byte[8];
            await ReadExactAsync(stream, header, 0, 8, cancellationToken);

            int typeInt = BitConverter.ToInt32(header, 0);
            int length = BitConverter.ToInt32(header, 4);

            if ((MessageType)typeInt != MessageType.ClientToMasterBatch) return;

            byte[] payload = new byte[length];
            await ReadExactAsync(stream, payload, 0, length, cancellationToken);

            var batch = MessageSerializer.DeserializeBatchRequest(Combine(header, payload), out _);

            Console.WriteLine($"[Master] Получен батч {batch.BatchId} — {batch.Images.Count} изображений");

            foreach (var img in batch.Images)
            {
                var task = new ImageTask(img, stream, clientUdpEndpoint);
                _scheduler.EnqueueBatch(batch.BatchId, task);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ClientHandler] Клиент отвалился при чтении батча: {ex.Message}");
            tcpClient.Close();
        }
    }

    private static byte[] Combine(byte[] a1, byte[] a2)
    {
        byte[] result = new byte[a1.Length + a2.Length];
        Buffer.BlockCopy(a1, 0, result, 0, a1.Length);
        Buffer.BlockCopy(a2, 0, result, a1.Length, a2.Length);
        return result;
    }

    /// <summary>
    /// Читает точное количество байт из потока
    /// </summary>
    private async Task<int> ReadExactAsync(NetworkStream stream, byte[] buffer, int offset, int count, Cancella-
tionToken cancellationToken)
    {
        int totalRead = 0;
        while (totalRead < count)
        {
            int read = await stream.ReadAsync(buffer, offset + totalRead, count - totalRead, cancellationToken);
            if (read == 0)
```

```
                throw new EndOfStreamException("Remote closed connection while reading data.");

            totalRead += read;
        }
        return totalRead;
    }


    /// <summary>
    /// Останавливает Master-сервер.
    /// </summary>
    public void Stop()
    {
        if (_isRunning)
        {
            _isRunning = false;
            _slaveListener?.Stop();
            _clientListener?.Stop();
            _progressSender?.Close();
            Console.WriteLine("[Master] Master-сервер остановлен.");
        }
    }
  }
}
```

Program.cs

```
namespace MasterNode
{
  class Program
  {
    static async Task Main(string[] args)
    {
      Console.WriteLine("====================================");
      Console.WriteLine("       MASTER NODE - Координатор");
      Console.WriteLine("====================================\n");

      // Порты по умолчанию
      int slavePort = 5000;  // Для подключения Slave-узлов
      int clientPort = 5001; // Для подключения Клиентов (TCP)

      // Если передали аргументы командной строки
      if (args.Length >= 1 && int.TryParse(args[0], out int sPort))
      {
        slavePort = sPort;
      }
      if (args.Length >= 2 && int.TryParse(args[1], out int cPort))
      {
        clientPort = cPort;
      }

      Console.WriteLine($"Slave TCP порт: {slavePort}");
      Console.WriteLine($"Client TCP порт: {clientPort}");
      Console.WriteLine();

      // Создаём CancellationToken для корректной остановки
      CancellationTokenSource cts = new CancellationTokenSource();

      // Обработка Ctrl+C для корректного завершения
      Console.CancelKeyPress += (sender, e) =>
      {
        e.Cancel = true;
        Console.WriteLine("\n\nПолучен сигнал остановки...");
```

```
            cts.Cancel();
        };

        // Создаём и запускаем Master-сервер
        MasterServer server = new MasterServer(slavePort, clientPort);

        try
        {
            await server.StartAsync(cts.Token);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Работа Master-узла прервана пользователем.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Критическая ошибка Master-узла: {ex.Message}");
        }
        finally
        {
            server.Stop();
        }

        Console.WriteLine("\nMaster-узел завершил работу.");
        Console.WriteLine("Нажмите любую клавишу для выхода...");
        Console.ReadKey();
    }
  }
}
```

ProgressSender.cs

```
using System.Net.Sockets;
using Common.Messages;

namespace MasterNode
{
  public class ProgressSender
  {
    private readonly ReliableUdpSenderWithQueue _sender;

    public ProgressSender()
    {
      _sender = new ReliableUdpSenderWithQueue();
    }

    public async Task SendProgressAsync(ImageTask task, int totalImages, int processedImages, string info = "")
    {
      var message = new ProgressMessage(
        task.ImageId,
        totalImages,
        processedImages,
        task.Status,
        task.FileName,
        info
      );

      try
      {
        byte[] data = MessageSerializer.SerializeProgressMessage(message);

        _sender.Enqueue(data, task.ClientUdpEndpoint);
      }
```

```
            catch (Exception ex)
            {
                Console.WriteLine($"[MasterUDP] Ошибка отправки прогресса: {ex.Message}");
            }
        }

        public async Task SendProgressAsync(ImageTask task, int totalImages, string info = "")
        {
            int processed = task.Status == 2 ? 1 : 0;
            await SendProgressAsync(task, totalImages, processed, info);
        }

        public void Close()
        {
            try { _sender.Close(); } catch { }
        }
    }
}
```

ReliableUdpSenderWithQueue.cs

```
using System.Collections.Concurrent;
using System.Net;
using System.Net.Sockets;

namespace MasterNode
{
    public class ReliableUdpSenderWithQueue
    {
        private readonly UdpClient _udp;
        private readonly CancellationTokenSource _cts = new();
        private readonly BlockingCollection<(byte[] data, IPEndPoint ep)> _queue;

        private int _sequence = 0;

        public ReliableUdpSenderWithQueue()
        {
            _udp = new UdpClient();
            _udp.Client.ReceiveTimeout = 300;

            _queue = new BlockingCollection<(byte[], IPEndPoint)>(new ConcurrentQueue<(byte[], IPEndPoint)>());

            Task.Run(SendLoopAsync);
        }

        public void Enqueue(byte[] data, IPEndPoint endpoint)
        {
            _queue.Add((data, endpoint));
        }

        private async Task SendLoopAsync()
        {
            while (!_cts.IsCancellationRequested)
            {
                try
                {
                    var (data, endpoint) = _queue.Take(_cts.Token);

                    await SendReliableAsync(data, endpoint);

                    //await Task.Delay(1);
                }
                catch (OperationCanceledException)
```

```
            {
                break;
            }
        }
    }

    private async Task SendReliableAsync(byte[] data, IPEndPoint endpoint)
    {
        int seq = Interlocked.Increment(ref _sequence);

        byte[] packet = new byte[data.Length + 4];
        BitConverter.GetBytes(seq).CopyTo(packet, 0);
        Buffer.BlockCopy(data, 0, packet, 4, data.Length);

        for (int attempt = 1; attempt <= 5; attempt++)
        {
            await _udp.SendAsync(packet, packet.Length, endpoint);

            try
            {
                var resTask = _udp.ReceiveAsync();
                var res = await resTask;

                int ackSeq = BitConverter.ToInt32(res.Buffer, 0);

                if (ackSeq == seq)
                    return; // доставлено
            }
            catch
            {
                // ignore timeout, retry
            }

            //await Task.Delay(1);
        }

        Console.WriteLine($"[UDP] НЕ доставлено после 5 попыток → seq {seq}");
    }

    public void Close()
    {
        _cts.Cancel();
        _udp.Close();
    }
  }
}
```

SlaveHandler.cs

```
using Common.Messages;
using System.Net.Sockets;
using System.Numerics;
using System.Threading.Tasks;

namespace MasterNode
{
  /// <summary>
  /// Обработчик одного Slave-узла.
  /// </summary>
  public class SlaveHandler
  {
    private readonly string _slaveId;
    private readonly NetworkStream _stream;
```

```
        private readonly TcpClient _client;
        private readonly TaskScheduler _scheduler;
        private ImageTask _currentTask;
        private readonly object _lock = new object();
        private bool _isDisconnected = false;

        private bool _isAvailable = true;

        public bool IsAvailable
        {
            get
            {
                lock (_lock) { return _isAvailable; }
            }
            private set
            {
                lock (_lock) { _isAvailable = value; }
            }
        }

        public string SlaveId => _slaveId;

        public SlaveHandler(TcpClient client, TaskScheduler scheduler)
        {
            _client = client;
            _client.NoDelay = true;
            _stream = client.GetStream();
            _scheduler = scheduler;
            _slaveId = Guid.NewGuid().ToString().Substring(0, 8);

            Console.WriteLine($"[Slave-{_slaveId}] Установлено соединение от Slave: {_client.Client.Re-
moteEndPoint}");
        }

        public void SetAvailable(bool val)
        {
            lock (_lock)
            {
                IsAvailable = false;
            }
        }

        /// <summary>
        /// Слушает соединение со Slave-узлом, чтобы корректно закрыть
        /// </summary>
        public async Task StartListeningAsync(CancellationToken cancellationToken)
        {
            try
            {
                while (!cancellationToken.IsCancellationRequested && !_isDisconnected)
                {
                    await Task.Delay(1000, cancellationToken);
                }
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine($"[Slave-{_slaveId}] Прослушивание отменено.");
            }
            finally
            {
                Disconnect();
            }
```

```
    }

    /// <summary>
    /// Отправляет задачу Slave-узлу
    /// </summary>
    public async Task SendTaskAsync(ImageTask task)
    {
        lock (_lock)
        {
            if (_isDisconnected)
            {
                throw new InvalidOperationException($"Slave {_slaveId} не доступен для приема новой задачи.");
            }

            _currentTask = task;
            IsAvailable = false;
        }

        try
        {
            Console.WriteLine($"[Master] Отправка задачи ID {task.ImageId} узлу - Slave-{_slaveId}...");

            byte[] taskData = MessageSerializer.SerializeImageMessage(
                MessageType.MasterToSlaveTask,
                task.TaskMessage);

            await _stream.WriteAsync(taskData, 0, taskData.Length);

            await _stream.FlushAsync();

            Console.WriteLine($"[Master] Задача ID {task.ImageId} отправлена узлу - Slave-{_slaveId} (размер:
{taskData.Length} байт).");

            // DELAY IMITATION
            //await Task.Delay(1000);

            _scheduler.UpdateTaskStatus(task.ImageId, 1, _slaveId);

            await ReceiveResultAsync();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[Master] Ошибка при отправке задачи или получении результата, узел - Slave-
{_slaveId}: {ex.Message}");

            lock (_lock)
            {
                _currentTask = null;
                IsAvailable = true;
            }

            Disconnect();
        }
    }

    /// <summary>
    /// Получает результат от Slave
    /// </summary>
    private async Task ReceiveResultAsync()
    {
        try
        {
```

```
            byte[] header = new byte[8];
            int bytesRead = await ReadExactAsync(_stream, header, 0, 8);

            if (bytesRead == 0)
                throw new Exception("Slave закрыл соединение до отправки результата");

            if (bytesRead < 8)
                throw new Exception($"Получен неполный заголовок ({bytesRead} байт)");

            int messageType = BitConverter.ToInt32(header, 0);
            int payloadLength = BitConverter.ToInt32(header, 4);

            Console.WriteLine($"[Master] Получение результата от Slave-{_slaveId}: размер {payloadLength}
байт...");

            if ((MessageType)messageType != MessageType.SlaveToMasterResult)
                throw new Exception($"Неожиданный тип сообщения {messageType} (ожидался {(int)Mes-
sageType.SlaveToMasterResult})");

            if (payloadLength < 0)
                throw new Exception($"Недопустимый размер: {payloadLength} байт");

            byte[] payload = new byte[payloadLength];
            bytesRead = await ReadExactAsync(_stream, payload, 0, payloadLength);

            if (bytesRead < payloadLength)
                throw new Exception($"Получено {bytesRead} из {payloadLength} байт");

            ImageMessage resultMessage = MessageSerializer.DeserializeImageMessage(payload, messageType, pay-
loadLength);

            if (_currentTask == null || _currentTask.ImageId != resultMessage.ImageId)
            {
                throw new Exception($"Получен результат ID {resultMessage.ImageId}, но ожидался {_cur-
rentTask?.ImageId ?? -1}");
            }

            Console.WriteLine($"[Master] Получен результат для ID {resultMessage.ImageId} от Slave-{_slaveId}.");

            await _scheduler.HandleTaskResult(_currentTask, resultMessage);

            lock (_lock)
            {
                _currentTask = null;
                IsAvailable = true;
            }

            _scheduler.RequestTaskAssignment();

        }
        catch (Exception ex)
        {
            Console.WriteLine($"[Slave-{_slaveId}] Ошибка получения результата: {ex.Message}");
            lock (_lock)
            {
                _currentTask = null;
                IsAvailable = true;
            }
        }
    }

    /// <summary>
```

```
        /// Читает точное количество байт из потока
        /// </summary>
        private async Task<int> ReadExactAsync(NetworkStream stream, byte[] buffer, int offset, int count)
        {
            int totalRead = 0;

            while (totalRead < count)
            {
                int read = await stream.ReadAsync(buffer, offset + totalRead, count - totalRead);

                if (read == 0)
                {
                    return totalRead;
                }

                totalRead += read;
            }

            return totalRead;
        }

        /// <summary>
        /// Разрывает соединение и уведомляет планировщик
        /// </summary>
        public void Disconnect()
        {
            lock (_lock)
            {
                if (_isDisconnected)
                    return;

                _isDisconnected = true;
            }

            try
            {
                _stream?.Close();
                _client?.Close();
                Console.WriteLine($"[Slave-{_slaveId}] Соединение разорвано.");
            }
            catch { /* Игнорируем ошибки при закрытии */ }

            // Уведомляем планировщик
            _scheduler.RemoveSlave(this);
        }
    }
}
```

TaskScheduler.cs

```
using Common.Messages;
using System.Collections.Concurrent;
using System.Net.Sockets;

namespace MasterNode
{
    /// <summary>
    /// Планировщик задач. Управляет очередью, распределяет задачи по Round-Robin и обрабатывает результаты.
    /// </summary>
    public class TaskScheduler
    {
        private readonly ConcurrentQueue<ImageTask> _taskQueue = new ConcurrentQueue<ImageTask>();
```

```
    private readonly ConcurrentDictionary<int, ImageTask> _activeTasks = new ConcurrentDictionary<int, Im-
ageTask>();
    private readonly List<SlaveHandler> _slaves = new List<SlaveHandler>();
    private int _nextSlaveIndex = 0;
    private readonly ProgressSender _progressSender;
    private readonly ConcurrentDictionary<long, NetworkStream> _batchClientStreams = new();
    private readonly ConcurrentDictionary<long, int> _batchRemaining = new();

    private readonly ConcurrentDictionary<int, long> _imageToBatchId = new();
    private readonly ConcurrentDictionary<long, int> _batchTotalImages = new();
    private readonly ConcurrentDictionary<long, int> _batchProcessedImages = new();

    public TaskScheduler(ProgressSender progressSender)
    {
        _progressSender = progressSender;
    }

    /// <summary>
    /// Добавляет батч в очередь.
    /// </summary>
    public void EnqueueBatch(long batchId, ImageTask task)
    {
        int currentRemaining = _batchRemaining.AddOrUpdate(batchId, 1, (_, old) => old + 1);
        int currentTotal = _batchTotalImages.AddOrUpdate(batchId, 1, (_, old) => old + 1);

        _batchProcessedImages.TryAdd(batchId, 0);

        _batchClientStreams.TryAdd(batchId, task.ClientStream);
        _imageToBatchId[task.ImageId] = batchId;
        _taskQueue.Enqueue(task);
        _activeTasks.TryAdd(task.ImageId, task);

        UpdateTaskStatus(task.ImageId, 0);

        Console.WriteLine($"[Scheduler] Задача ID {task.ImageId} из батча {batchId} добавлена в очередь ({cur-
rentRemaining}/{currentTotal})");

        AssignNextTask();
    }

    /// <summary>
    /// Добавляет Slave-узел в список доступных.
    /// </summary>
    public void AddSlave(SlaveHandler slave)
    {
        lock (_slaves)
        {
            _slaves.Add(slave);
            Console.WriteLine($"[Scheduler] Slave {slave.SlaveId} подключен. Всего Slave-узлов: {_slaves.Count}");
        }
        AssignNextTask();
    }

    /// <summary>
    /// Удаляет Slave-узел из списка.
    /// </summary>
    public void RemoveSlave(SlaveHandler slave)
    {
        lock (_slaves)
        {
            _slaves.Remove(slave);
            Console.WriteLine($"[Scheduler] Slave {slave.SlaveId} отключен. Всего Slave-узлов: {_slaves.Count}");
```

```
        }
    }

    /// <summary>
    /// Запрашивает попытку назначения задачи. Вызывается после подключения Slave или получения результата.
    /// </summary>
    public void RequestTaskAssignment()
    {
        AssignNextTask();
    }

    /// <summary>
    /// Назначает следующую задачу по стратегии Round-Robin с учётом занятости Slave.
    /// </summary>
    private void AssignNextTask()
    {
        lock (_slaves)
        {
            if (_taskQueue.IsEmpty || _slaves.Count == 0)
                return;

            int startIndex = _nextSlaveIndex;
            bool taskAssigned = false;

            do
            {
                SlaveHandler slave = _slaves[_nextSlaveIndex];
                _nextSlaveIndex = (_nextSlaveIndex + 1) % _slaves.Count;

                Console.WriteLine($"[Scheduler] Slave-{slave.SlaveId} доступен - {slave.IsAvailable}");
                if (slave.IsAvailable && _taskQueue.TryDequeue(out ImageTask task))
                {
                    slave.SetAvailable(false);
                    task.SetProcessing(slave);

                    _ = Task.Run(async () =>
                    {
                        try
                        {
                            await slave.SendTaskAsync(task);
                        }
                        catch (Exception ex)
                        {
                            Console.WriteLine($"[Scheduler] Ошибка при назначении задачи на Slave {slave.SlaveId}: {ex}");
                            HandleTaskFailure(task, $"Задача ID {task.ImageId} не выполнена: Slave отключился или произошла ошибка.");
                        }
                    });

                    Console.WriteLine($"[Scheduler] Назначена задача ID {task.ImageId} узлу {slave.SlaveId}. Задач в очереди: {_taskQueue.Count}");

                    taskAssigned = true;
                    break;
                }

            } while (_nextSlaveIndex != startIndex);

            if (!taskAssigned)
            {
```

```
            Console.WriteLine("[Scheduler] Все Slave заняты. Задача останется в очереди.");
        }
    }
}

/// <summary>
/// Обновляет статус задачи и отправляет уведомление о прогрессе.
/// </summary>
public void UpdateTaskStatus(int imageId, int status, string info = "")
{
    if (!_activeTasks.TryGetValue(imageId, out ImageTask task)) return;
    task.Status = status;

    if (_imageToBatchId.TryGetValue(imageId, out long batchId))
    {
        int total = _batchTotalImages.GetValueOrDefault(batchId, 0);
        int processed = _batchProcessedImages.GetValueOrDefault(batchId, 0);

        _progressSender.SendProgressAsync(task, total, processed, info);
    }
    else
    {
        int activeCount = _activeTasks.Count - _taskQueue.Count;
        _progressSender.SendProgressAsync(task, activeCount, info);
    }
}

/// <summary>
/// Обрабатывает результат, полученный от Slave-узла.
/// </summary>
public async Task HandleTaskResult(ImageTask task, ImageMessage resultMessage)
{
    if (task == null) return;

    task.SetCompleted();
    if (_imageToBatchId.TryGetValue(task.ImageId, out long batchId))
        _batchProcessedImages.AddOrUpdate(batchId, 1, (k, v) => v + 1);

    Console.WriteLine($"[Scheduler] Задача ID {task.ImageId} завершена");

    // DELAY IMITATION
    //await Task.Delay(1000);

    await SendResultToClient(task, resultMessage);

    UpdateTaskStatus(task.ImageId, task.Status);
    _activeTasks.TryRemove(task.ImageId, out _);
    _imageToBatchId.TryRemove(task.ImageId, out _);
}

/// <summary>
/// Отправляет результат клиенту.
/// </summary>
private async Task SendResultToClient(ImageTask task, ImageMessage resultMessage)
{
    if (!_imageToBatchId.TryGetValue(task.ImageId, out long batchId))
        return;

    if (!_batchClientStreams.TryGetValue(batchId, out var stream) || stream == null || !stream.CanWrite)
    {
        Console.WriteLine($"[Result] Клиентский поток недоступен для батча {batchId}");
        return;
```

```
            }

            try
            {
                byte[] resultData = MessageSerializer.SerializeImageMessage(
                    MessageType.MasterToClientResult, resultMessage);

                await stream.WriteAsync(resultData, 0, resultData.Length);
                await stream.FlushAsync();

                Console.WriteLine($"[Result] Отправлен результат ID {task.ImageId} (батч {batchId})");

                int remaining = _batchRemaining.AddOrUpdate(batchId, 0, (_, old) => old - 1);

                if (remaining == 0)
                {
                    Console.WriteLine($"[Master] Батч {batchId} полностью завершён. Закрываем TCP-соединение с
клиентом.");

                    _batchClientStreams.TryRemove(batchId, out _);
                    _batchRemaining.TryRemove(batchId, out _);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"[Result] Ошибка отправки результата: {ex.Message}");
            }
        }

        /// <summary>
        /// Обрабатывает ошибку решения задачи.
        /// Отправляет клиенту сообщение об ошибке,
        /// кидает задачу обратно в очередь для повторной обработки
        /// </summary>
        private void HandleTaskFailure(ImageTask task, string errorMessage)
        {
            task.SetError();
            Console.WriteLine($"[Scheduler] Ошибка задачи ID {task.ImageId}: {errorMessage}");

            UpdateTaskStatus(task.ImageId, task.Status, "Ошибка: " + errorMessage);
            _taskQueue.Enqueue(task);
        }
    }
}
```

ClientService.cs

```
using Common.Messages;
using System.Diagnostics;
using System.IO;
using System.Net;
using System.Net.Sockets;

namespace ClientApp.Services
{
    public class UpdateTaskStatusData
    {
        public string FileName { get; set; }
        public string StatusText { get; set; }
    }
    /// <summary>
    /// Сервис для управления сетевым взаимодействием с MasterNode.
    /// </summary>
```

```
public class ClientService
{
    private readonly IPEndPoint _masterTcpEndpoint;
    private const int ClientUdpPort = 6000;

    public event Action<UpdateTaskStatusData> ProgressUpdated;
    public event Action<ImageMessage> TaskCompleted;
    public event Action<string> ErrorOccurred;

    public ClientService(string masterHost, int masterTcpPort)
    {
        _masterTcpEndpoint = new IPEndPoint(IPAddress.Parse(masterHost), masterTcpPort);
    }
    public async Task SendBatchAndReceiveResultAsync(BatchRequestMessage batch, CancellationToken cancella-
tionToken)
    {
        TcpClient tcpClient = new();
        NetworkStream stream = null;

        try
        {
            await tcpClient.ConnectAsync(_masterTcpEndpoint.Address, _masterTcpEndpoint.Port);
            stream = tcpClient.GetStream();

            byte[] batchData = MessageSerializer.SerializeBatchRequest(MessageType.ClientToMasterBatch, batch);
            await stream.WriteAsync(batchData, cancellationToken);
            await stream.FlushAsync(cancellationToken);

            var sw = Stopwatch.StartNew();

            _ = ListenForProgressAsync(ClientUdpPort, cancellationToken);

            int expectedResults = batch.Images.Count;

            int received = 0;
            byte[] header = new byte[8];

            while (received < expectedResults && !cancellationToken.IsCancellationRequested)
            {
                int readHeader = await ReadExactAsync(stream, header, 0, 8);
                if (readHeader == 0)
                {
                    break;
                }
                if (readHeader < 8)
                {
                    throw new EndOfStreamException("Не удалось прочитать полный заголовок.");
                }

                int messageType = BitConverter.ToInt32(header, 0);
                int payloadLength = BitConverter.ToInt32(header, 4);
                if (payloadLength < 0)
                    throw new Exception($"Некорректная длина payload: {payloadLength}");

                byte[] payload = new byte[payloadLength];
                int readPayload = await ReadExactAsync(stream, payload, 0, payloadLength);
                if (readPayload < payloadLength)
                    throw new EndOfStreamException("Не удалось прочитать полный payload.");

                var result = MessageSerializer.DeserializeImageMessage(payload, messageType, payloadLength);

                if (messageType == (int)MessageType.MasterToClientResult && result.FileName != "ERROR")
```

```
                {
                    TaskCompleted?.Invoke(result);
                    received++;
                }
            }

            sw.Stop();
            Debug.WriteLine($"Время выполнения: {sw.ElapsedMilliseconds} мс");
        }
        catch (Exception ex)
        {
            ErrorOccurred?.Invoke(ex.Message);
        }
        finally
        {
            tcpClient.Close();
        }
    }

    private async Task ListenForProgressAsync(int localUdpPort, CancellationToken cancellationToken)
    {
        UdpClient udpClient = null;
        UdpClient ackClient = null;
        try
        {
            udpClient = new UdpClient(localUdpPort);
            ackClient = new UdpClient();
            udpClient.Client.ReceiveBufferSize = 1024 * 1024;

            while (!cancellationToken.IsCancellationRequested)
            {
                UdpReceiveResult result = await udpClient.ReceiveAsync();

                int seq = BitConverter.ToInt32(result.Buffer, 0);

                byte[] msgData = new byte[result.Buffer.Length - 4];
                Buffer.BlockCopy(result.Buffer, 4, msgData, 0, msgData.Length);

                ProgressMessage progress = MessageSerializer.DeserializeProgressMessage(msgData);

                byte[] ack = BitConverter.GetBytes(seq);
                await ackClient.SendAsync(ack, ack.Length, result.RemoteEndPoint);

                string statusText = progress.Status switch
                {
                    0 => "В очереди",
                    1 => "Обрабатывается",
                    2 => "Завершено (100%)",
                    3 => "Ошибка",
                    _ => "Неизвестный статус"
                };

                ProgressUpdated?.Invoke(new UpdateTaskStatusData { FileName = progress.FileName, StatusText = sta-
tusText });
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ClientService-UDP] Ошибка прослушивания UDP: {ex.Message}");
        }
        finally
        {
```

```
            udpClient?.Close();
            ackClient?.Close();
        }
    }

    private async Task<int> ReadExactAsync(NetworkStream stream, byte[] buffer, int offset, int count)
    {
        int totalRead = 0;

        while (totalRead < count)
        {
            int read = await stream.ReadAsync(buffer, offset + totalRead, count - totalRead);

            if (read == 0)
            {
                return totalRead;
            }

            totalRead += read;
        }

        return totalRead;
    }
  }
}
```

ClientViewModel.cs

```
using ClientApp.Services;
using Common.Messages;
using Microsoft.Win32;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Runtime.CompilerServices;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media.Imaging;

namespace ClientApp.ViewModels
{
  public class ClientViewModel : INotifyPropertyChanged
  {
    private readonly ClientService _clientService;

    public ObservableCollection<ImageItem> Images { get; } = new();

    private string _progressText = "Выберите изображения и нажмите 'Отправить все'";
    public string ProgressText
    {
        get => _progressText;
        set { _progressText = value; OnPropertyChanged(); }
    }

    private bool _isProcessing;
    public bool IsProcessing
    {
        get => _isProcessing;
        set { _isProcessing = value; OnPropertyChanged(); CommandManager.InvalidateRequerySuggested(); }
    }

    public ICommand SelectImagesCommand { get; }
```

```
    public ICommand SendAllCommand { get; }
    public ICommand ClearCommand { get; }

    public ClientViewModel()
    {
        _clientService = new ClientService("127.0.0.1", 5001);
        _clientService.ProgressUpdated += OnTaskUpateStatus;
        _clientService.TaskCompleted += OnTaskCompleted;
        _clientService.ErrorOccurred += e => Application.Current.Dispatcher.Invoke(() => ProgressText =
"ОШИБКА: " + e);

        SelectImagesCommand = new RelayCommand(SelectImages);

        SendAllCommand = new RelayCommand(
            async () => await SendAll(),
            () => CanSendAll()
        );

        ClearCommand = new RelayCommand(() => Images.Clear());
    }

    private void SelectImages()
    {
        var dlg = new OpenFileDialog
        {
            Multiselect = true,
            Filter = "Изображения|*.jpg;*.jpeg;*.png;*.bmp"
        };
        if (dlg.ShowDialog() == true)
        {
            foreach (var path in dlg.FileNames)
            {
                if (Images.Any(i => i.FilePath == path)) continue;

                try
                {
                    var bitmap = new BitmapImage();
                    bitmap.BeginInit();
                    bitmap.UriSource = new Uri(path);
                    bitmap.CacheOption = BitmapCacheOption.OnLoad;
                    bitmap.EndInit();
                    bitmap.Freeze();

                    Images.Add(new ImageItem
                    {
                        FilePath = path,
                        Original = bitmap,
                        Status = "Готов к отправке"
                    });
                }
                catch { /* плохой файл — пропускаем */ }
            }
        }
    }

    private bool CanSendAll() => !IsProcessing && Images.Any(i => i.Status == "Готов к отправке" || i.Status.Con-
tains("Ошибка"));

    private async Task SendAll()
    {
        if (!Images.Any()) return;
```

```
        IsProcessing = true;
        ProgressText = $"Отправка {Images.Count} изображений...";

        var batchId = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();

        var batch = new BatchRequestMessage(
            batchId,
            Images.Select((item, index) => new ImageMessage(
                (int)(batchId + index),
                item.FileName,
                (int)item.Original.Width,
                (int)item.Original.Height,
                1,
                File.ReadAllBytes(item.FilePath)
            )).ToList()
        );

        try
        {
            await _clientService.SendBatchAndReceiveResultAsync(batch, CancellationToken.None);
        }
        catch (Exception ex)
        {
            ProgressText = $"Ошибка: {ex.Message}";
        }
        finally
        {
            IsProcessing = false;
        }
    }

    private void OnTaskUpateStatus(UpdateTaskStatusData data)
    {
        Application.Current.Dispatcher.Invoke(() =>
        {
            var item = Images.FirstOrDefault(i => i.FileName == data.FileName);

            if (item != null)
            {
                item.Status = data.StatusText;
            }

            if (Images.All(i => i.Status.ToLower().Contains("завершено") || i.Status.ToLower().Contains("ошибка")))
            {
                ProgressText = "ВСЕ ЗАДАЧИ ЗАВЕРШЕНЫ!";
                IsProcessing = false;
            }
        });
    }

    private void OnTaskCompleted(ImageMessage result)
    {
        Application.Current.Dispatcher.Invoke(() =>
        {
            var item = Images.FirstOrDefault(i => result.FileName.EndsWith(i.FileName, StringComparison.Ordi-
nalIgnoreCase));

            if (item != null)
            {
                item.Processed = BytesToBitmapSource(result.ImageData);
            }
```

```
            if (Images.All(i => i.Status.ToLower().Contains("завершено") || i.Status.ToLower().Contains("ошибка")))
            {
                ProgressText = "ВСЕ ЗАДАЧИ ЗАВЕРШЕНЫ!";
                IsProcessing = false;
            }
        });
    }

    private BitmapSource? BytesToBitmapSource(byte[] bytes)
    {
        if (bytes == null || bytes.Length == 0)
        {
            return null;
        }

        try
        {
            using var ms = new MemoryStream(bytes);
            var image = new BitmapImage();

            image.BeginInit();

            image.CacheOption = BitmapCacheOption.OnLoad;
            image.StreamSource = ms;

            image.EndInit();

            image.Freeze();

            return image;
        }
        catch (Exception ex) when (ex is EndOfStreamException || ex is IOException || ex is NotSupportedException ||
ex is ArgumentException)
        {
            Debug.WriteLine($"[BytesToBitmapSource] Ошибка при загрузке изображения: {ex.GetType().Name} -
{ex.Message}");
            return null;
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged([CallerMemberName] string name = null)
        => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));

    public class RelayCommand : ICommand
    {
        private readonly Action _execute;
        private readonly Func<bool> _canExecute;

        public RelayCommand(Action execute, Func<bool> canExecute = null)
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter) => _canExecute == null || _canExecute();

        public void Execute(object parameter) => _execute();

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
```

```
            remove { CommandManager.RequerySuggested -= value; }
        }
    }
  }
}
```

RobertsOperator.cs

```
using System.Drawing;
using System.Drawing.Imaging;

namespace Common.ImageProcessing
{
  /// <summary>
  /// Реализация оператора Робертса для обнаружения границ
  /// </summary>
  public static class RobertsOperator
  {
    /// <summary>
    /// Применяет оператор Робертса к изображению
    /// </summary>
    /// <param name="sourceImage">Исходное изображение</param>
    /// <returns>Изображение с выделенными границами</returns>
    public static Bitmap ApplyRobertsOperatorParallel(Bitmap sourceImage)
    {
      if (sourceImage == null)
        throw new ArgumentNullException(nameof(sourceImage));

      int width = sourceImage.Width;
      int height = sourceImage.Height;

      Bitmap resultImage = new Bitmap(width, height);

      BitmapData srcData = sourceImage.LockBits(
        new Rectangle(0, 0, width, height),
        ImageLockMode.ReadOnly,
        PixelFormat.Format24bppRgb);

      BitmapData dstData = resultImage.LockBits(
        new Rectangle(0, 0, width, height),
        ImageLockMode.WriteOnly,
        PixelFormat.Format24bppRgb);

      int strideSrc = srcData.Stride;
      int strideDst = dstData.Stride;

      unsafe
      {
        byte* srcPtr = (byte*)srcData.Scan0;
        byte* dstPtr = (byte*)dstData.Scan0;

        Parallel.For(0, height - 1, y =>
        {
          byte* rowSrc = srcPtr + y * strideSrc;
          byte* nextRowSrc = srcPtr + (y + 1) * strideSrc;
          byte* rowDst = dstPtr + y * strideDst;

          for (int x = 0; x < width - 1; x++)
          {
            // Получаем указатели на пиксели
            byte* p00 = rowSrc + x * 3;
            byte* p01 = rowSrc + (x + 1) * 3;
            byte* p10 = nextRowSrc + x * 3;
```

```
                byte* p11 = nextRowSrc + (x + 1) * 3;

                int gray00 = (int)(p00[2] * 0.299 + p00[1] * 0.587 + p00[0] * 0.114);
                int gray01 = (int)(p01[2] * 0.299 + p01[1] * 0.587 + p01[0] * 0.114);
                int gray10 = (int)(p10[2] * 0.299 + p10[1] * 0.587 + p10[0] * 0.114);
                int gray11 = (int)(p11[2] * 0.299 + p11[1] * 0.587 + p11[0] * 0.114);

                int gx = gray00 - gray11;
                int gy = gray01 - gray10;

                int gradient = (int)Math.Sqrt(gx * gx + gy * gy);
                if (gradient > 255) gradient = 255;
                if (gradient < 0) gradient = 0;

                // Запись в результирующее изображение
                byte* pRes = rowDst + x * 3;
                pRes[0] = pRes[1] = pRes[2] = (byte)gradient;
            }

            byte* pBlack = rowDst + (width - 1) * 3;
            pBlack[0] = pBlack[1] = pBlack[2] = 0;
        });

        byte* lastRow = dstPtr + (height - 1) * strideDst;
        for (int x = 0; x < width; x++)
        {
            byte* p = lastRow + x * 3;
            p[0] = p[1] = p[2] = 0;
        }
    }

    sourceImage.UnlockBits(srcData);
    resultImage.UnlockBits(dstData);

    return resultImage;
}

/// <summary>
/// Конвертирует байты в Bitmap
/// </summary>
public static Bitmap BytesToBitmap(byte[] imageBytes)
{
    using (var ms = new MemoryStream(imageBytes))
    {
        return new Bitmap(ms);
    }
}

/// <summary>
/// Конвертирует Bitmap в байты
/// </summary>
public static byte[] BitmapToBytes(Bitmap image, ImageFormat format)
{
    using (var ms = new MemoryStream())
    {
        image.Save(ms, format);
        return ms.ToArray();
    }
}
  }
}
```

ПРИЛОЖЕНИЕ Б
(обязательное)

Руководство программиста

Разработанное программное обеспечение представляет собой распределённую систему обработки изображений с использованием архитектуры *Master-Slave*, реализованную на языке *C#* с применением *WPF* для клиентского интерфейса и двух консольных приложений для *Master*- и *Slave*-узлов. Перед началом работы пользователю необходимо убедиться, что все файлы проекта находятся в доступной директории, а порты для *TCP* и *UDP* соединений не заняты другими приложениями. Без выполнения этих условий корректное взаимодействие *Master*- и *Slave*-узлов, передача изображений, применение оператора Робертса и отображение прогресса обработки будут невозможны.

Для корректной работы программного средства необходимо соблюдение следующих условий:

– поддерживаемая операционная система *Windows* 7 и выше;

– установленный *.NET Framework*;

– наличие стандартной клавиатуры и мыши для управления интерфейсом WPF, а также устройства вывода: монитор с поддержкой разрешения экрана, достаточного для отображения пользовательского интерфейса.

Перед запуском системы необходимо выполнить следующие шаги: открыть терминал или командную строку и перейти в директорию с файлами проекта; запустить *Master*-узел командой *MasterNode.exe* 5000 5001, где первый параметр – порт для подключения *Slave*-узлов, а второй – порт для подключения клиентских приложений. При необходимости порты могут быть изменены через аргументы командной строки. После запуска *Master*-узла следует запустить один или несколько *Slave*-узлов командой *SlaveNode.exe Slave*-1 127.0.0.1 5000, где первый параметр – уникальное имя *Slave*-узла, второй – адрес хоста *Master*-узла, а третий – порт для подключения. Аргументы командной строки позволяют гибко настраивать подключение узлов в локальной сети или на одном компьютере.

Запуск клиентского приложения осуществляется через *WPF*-интерфейс, который обеспечивает выбор изображений для обработки и отображение прогресса выполнения задач в реальном времени с использованием протокола *UDP*. После отправки изображений *Master*-узел распределяет их между Slave-узлами с использованием алгоритма *Round Robin, Slave*-узлы применяют оператор Робертса к изображениям и возвращают результаты обратно.

ПРИЛОЖЕНИЕ В
(обязательное)

Руководство системного программиста

Разработанное программное обеспечение представляет собой распределённую систему обработки изображений с использованием архитектуры *Master-Slave*, реализованную на языке *C#* с применением *WPF* для клиентского интерфейса и двух консольных приложений для *Master*- и *Slave*-узлов. Перед началом работы пользователю необходимо убедиться, что все файлы проекта находятся в доступной директории, а порты для *TCP* и *UDP* соединений не заняты другими приложениями. Без выполнения этих условий корректное взаимодействие *Master*- и *Slave*-узлов, передача изображений, применение оператора Робертса и отображение прогресса обработки будут невозможны.

Для корректной работы программного средства необходимо соблюдение следующих условий:

– поддерживаемая операционная система *Windows* 7 и выше;

– установленный *.NET Framework*;

– наличие стандартной клавиатуры и мыши для управления интерфейсом WPF, а также устройства вывода: монитор с поддержкой разрешения экрана, достаточного для отображения пользовательского интерфейса.

Перед запуском системы необходимо выполнить следующие шаги: открыть терминал или командную строку и перейти в директорию с файлами проекта; запустить *Master*-узел командой *MasterNode.exe* 5000 5001, где первый параметр – порт для подключения *Slave*-узлов, а второй – порт для подключения клиентских приложений. При необходимости порты могут быть изменены через аргументы командной строки. После запуска *Master*-узла следует запустить один или несколько *Slave*-узлов командой *SlaveNode.exe Slave*-1 127.0.0.1 5000, где первый параметр – уникальное имя *Slave*-узла, второй – адрес хоста *Master*-узла, а третий – порт для подключения.

ПРИЛОЖЕНИЕ Г
(обязательное)

Руководство пользователя

Разработанное программное обеспечение представляет собой распределённую систему обработки изображений с архитектурой *Master-Slave*, реализованную на языке *C#* с использованием *WPF* для клиентского интерфейса и двух консольных приложений для *Master*- и *Slave*-узлов. Система позволяет пользователю отправлять изображения на обработку, получать результаты с применением оператора Робертса и отслеживать прогресс выполнения задач в реальном времени. Перед началом работы необходимо убедиться, что все файлы проекта находятся в одной директории, а указанные порты *TCP* и *UDP* свободны для подключения *Master*-, *Slave*-узлов и клиентского приложения. Без соблюдения этих условий корректная работа системы будет невозможна.

Для корректной работы программного средства необходимо соблюдение следующих условий:

– поддерживаемая операционная система *Windows* 7 и выше;

– установленный *.NET Framework*;

– наличие стандартной клавиатуры и мыши для управления интерфейсом WPF, а также устройства вывода: монитор с поддержкой разрешения экрана, достаточного для отображения пользовательского интерфейса.

Перед запуском системы необходимо выполнить следующие шаги: открыть терминал или командную строку и перейти в директорию с файлами проекта; запустить *Master*-узел командой *MasterNode.exe* 5000 5001, где первый параметр – порт для подключения *Slave*-узлов, а второй – порт для подключения клиентских приложений. При необходимости порты могут быть изменены через аргументы командной строки. После запуска *Master*-узла следует запустить один или несколько *Slave*-узлов командой *SlaveNode.exe Slave*-1 127.0.0.1 5000, где первый параметр – уникальное имя *Slave*-узла, второй – адрес хоста *Master*-узла, а третий – порт для подключения.

Запуск клиентского приложения осуществляется через *WPF*-интерфейс, который обеспечивает выбор изображений для обработки, отправку их на *Master*-узел и отображение прогресса выполнения задач в реальном времени. Master-узел распределяет изображения между *Slave*-узлами с использованием алгоритма *Round Robin, Slave*-узлы применяют оператор Робертса и возвращают результаты обратно.

Если возникают проблемы с подключением узлов или отображением прогресса, рекомендуется убедиться, что порты *TCP* и *UDP* свободны, *Master*-узел запущен корректно и все *Slave*-узлы подключены к указанному порту. В случае необходимости следует перезапустить *Master*- и *Slave*-узлы, а затем повторно запустить *WPF*-клиент.

ПРИЛОЖЕНИЕ Д
(обязательное)

Графические изображения приложения

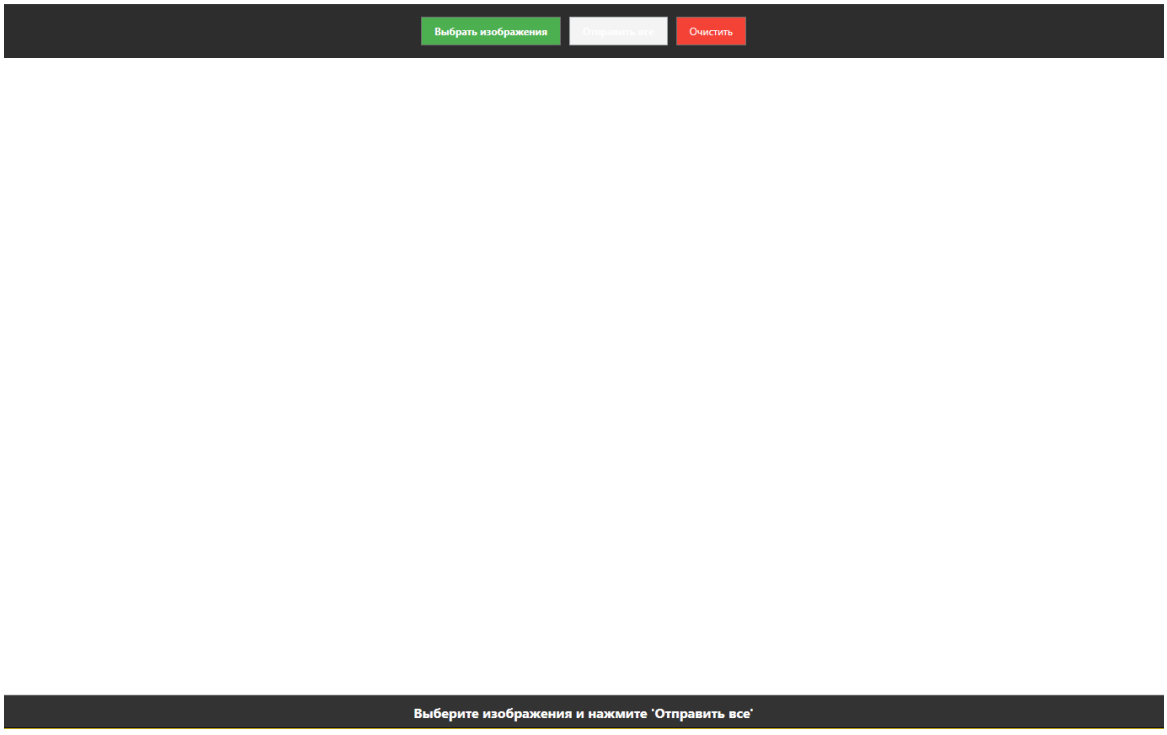

Рисунок Д.1 – Главное окно клиента

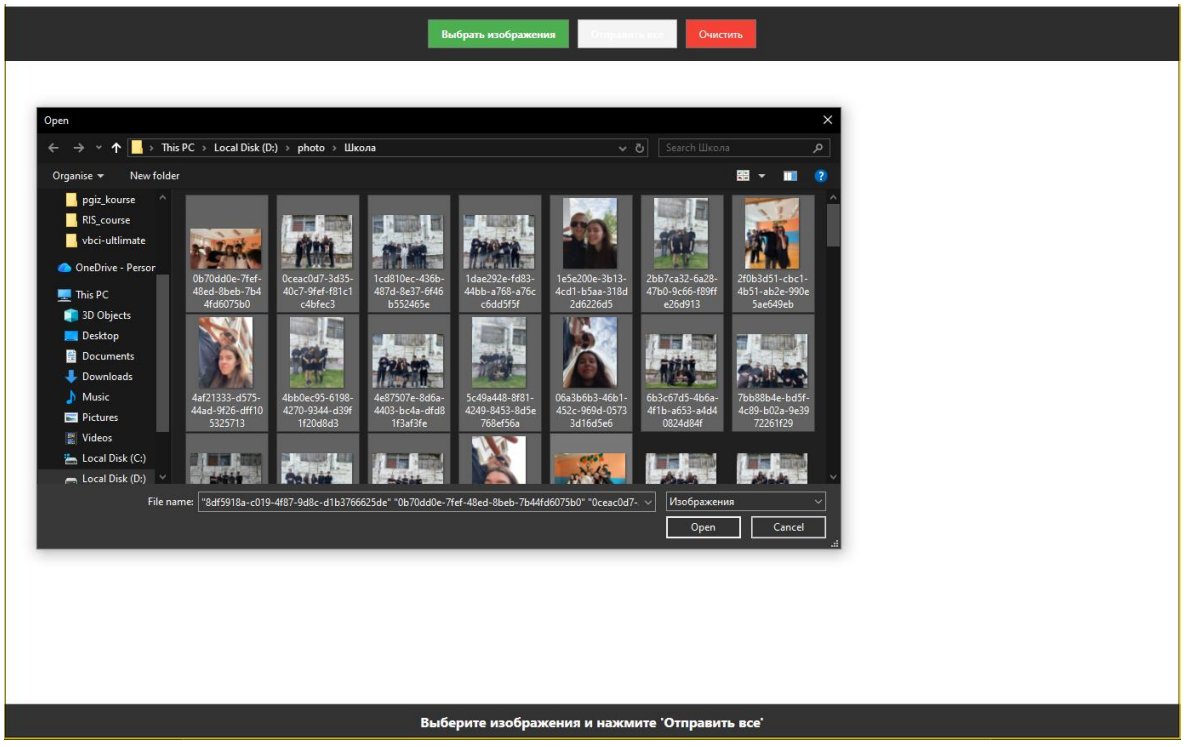

Рисунок Д.2 – Выбор изображений

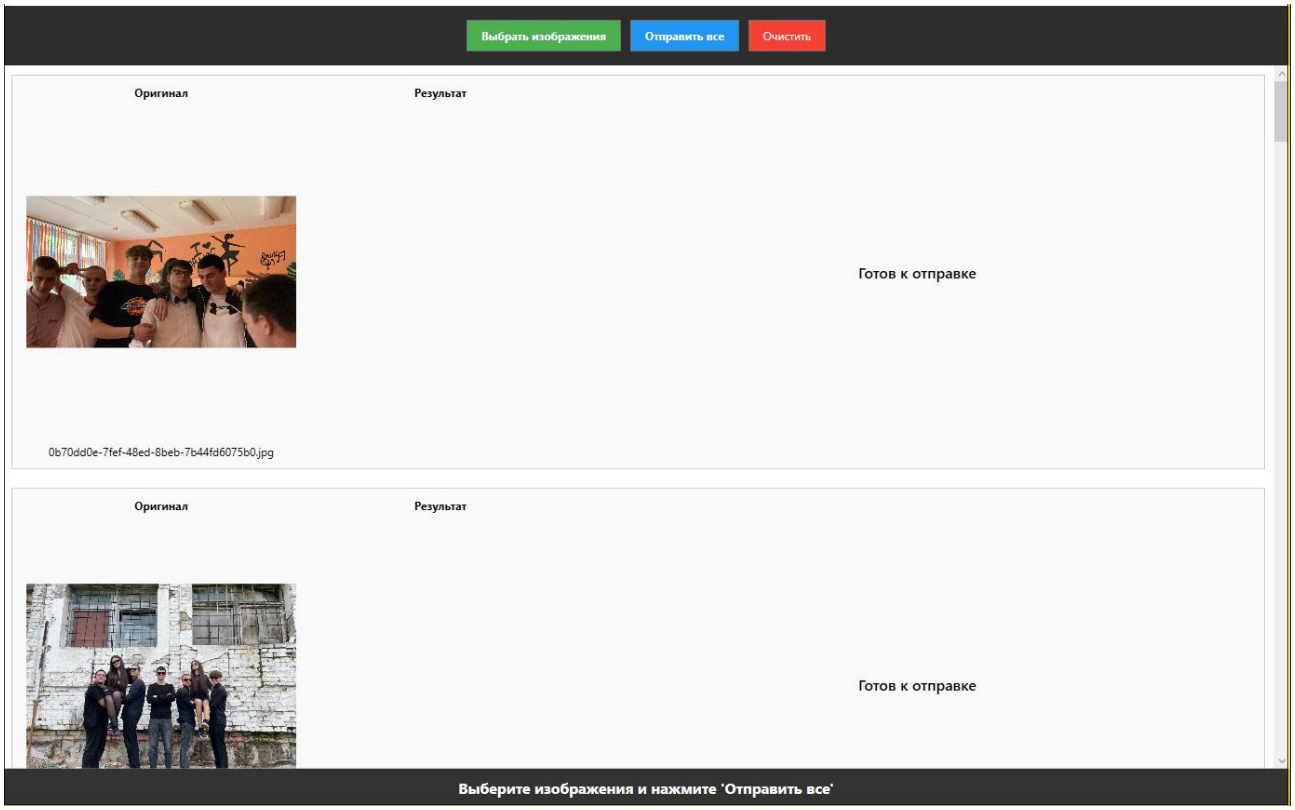

Рисунок Д.3 – Отображение выбранных изоюражений

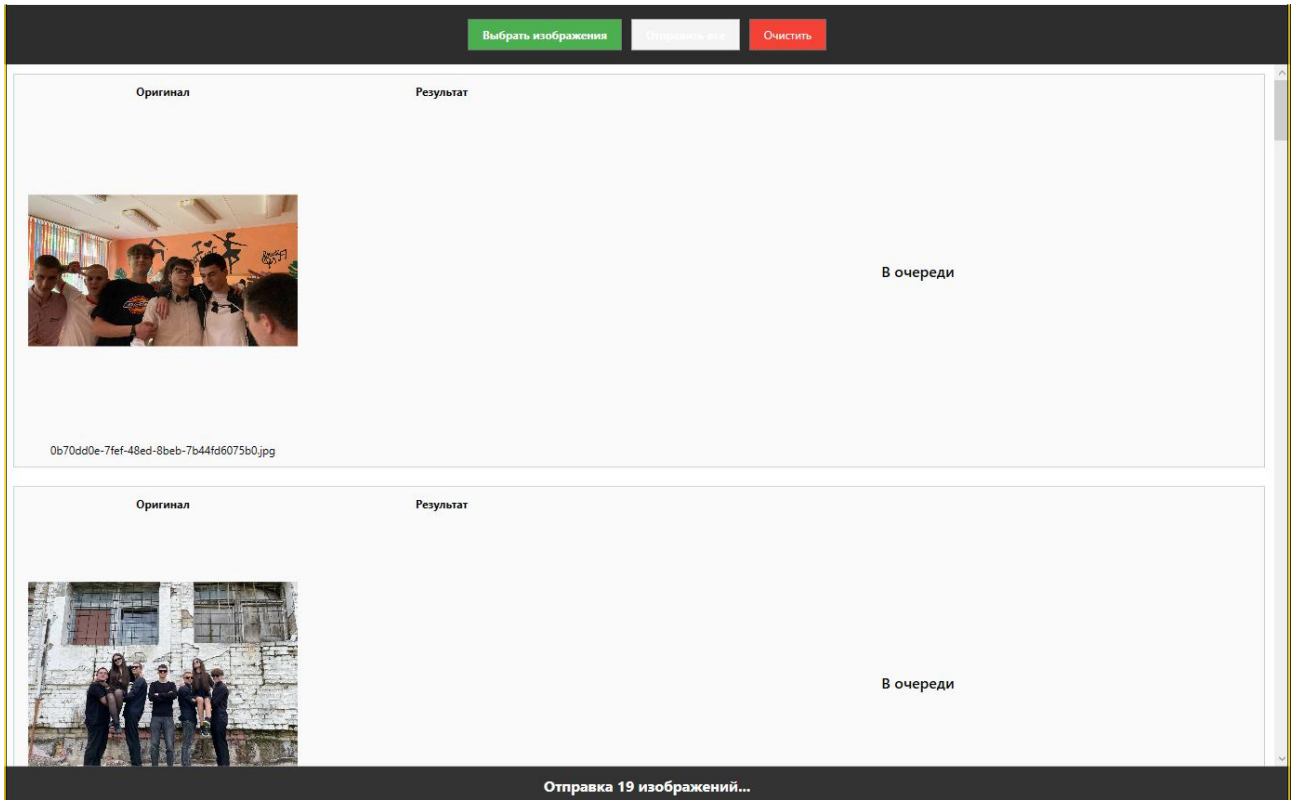

Рисунок Д.4 – Отображение статуса изображений при отправке на обработку

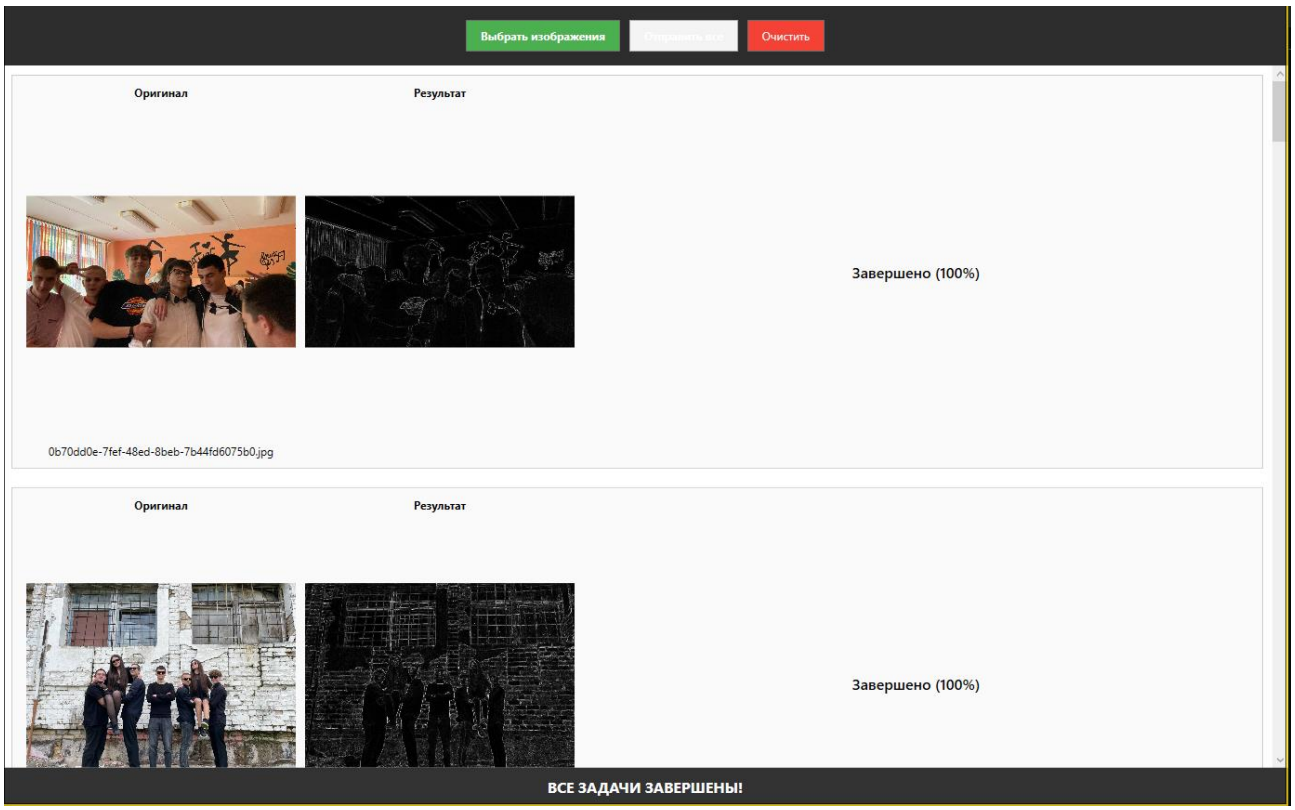

Рисунок Д.5 – Отображение результатов работы приложения

```
==================================
       MASTER NODE - Координатор
==================================

Slave TCP порт: 5000
Client TCP порт: 5001

[MasterTCP] Сервер для Slave запущен на порту 5000...
[MasterTCP] Сервер для Клиентов запущен на порту 5001...
[Slave-e6947ca6] Установлено соединение от Slave: 127.0.0.1:51930
[Scheduler] Slave e6947ca6 подключен. Всего Slave-узлов: 1
[Master] Получен батч 1765348984502 - 4 изображений
[Scheduler] Задача ID 117425846 из батча 1765348984502 добавлена в очередь (1/1)
[Scheduler] Slave-e6947ca6 доступен - True
[Scheduler] Назначена задача ID 117425846 узлу e6947ca6. Задач в очереди: 0
[Scheduler] Задача ID 117425847 из батча 1765348984502 добавлена в очередь (2/2)
[Scheduler] Slave-e6947ca6 доступен - False
[Scheduler] Все Slave заняты. Задача останется в очереди.
[Scheduler] Задача ID 117425848 из батча 1765348984502 добавлена в очередь (3/3)
[Master] Отправка задачи ID 117425846 узлу - Slave-e6947ca6...
[Scheduler] Slave-e6947ca6 доступен - False
[Scheduler] Все Slave заняты. Задача останется в очереди.
[Master] Задача ID 117425846 отправлена узлу - Slave-e6947ca6 (размер: 115303 байт).
[Scheduler] Задача ID 117425849 из батча 1765348984502 добавлена в очередь (4/4)
[Scheduler] Slave-e6947ca6 доступен - False
[Scheduler] Все Slave заняты. Задача останется в очереди.
[Master] Получение результата от Slave-e6947ca6: размер 838618 байт...
[Master] Получен результат для ID 117425846 от Slave-e6947ca6.
[Scheduler] Задача ID 117425846 завершена
[Result] Отправлен результат ID 117425846 (батч 1765348984502)
[Scheduler] Slave-e6947ca6 доступен - True
[Scheduler] Назначена задача ID 117425847 узлу e6947ca6. Задач в очереди: 2
[Master] Отправка задачи ID 117425847 узлу - Slave-e6947ca6...
[Master] Задача ID 117425847 отправлена узлу - Slave-e6947ca6 (размер: 259983 байт).
[Master] Получение результата от Slave-e6947ca6: размер 1492881 байт...
[Master] Получен результат для ID 117425847 от Slave-e6947ca6.
[Scheduler] Задача ID 117425847 завершена
[Result] Отправлен результат ID 117425847 (батч 1765348984502)
[Scheduler] Slave-e6947ca6 доступен - True
```

Рисунок Д.6 – логирование на *Master*-узле

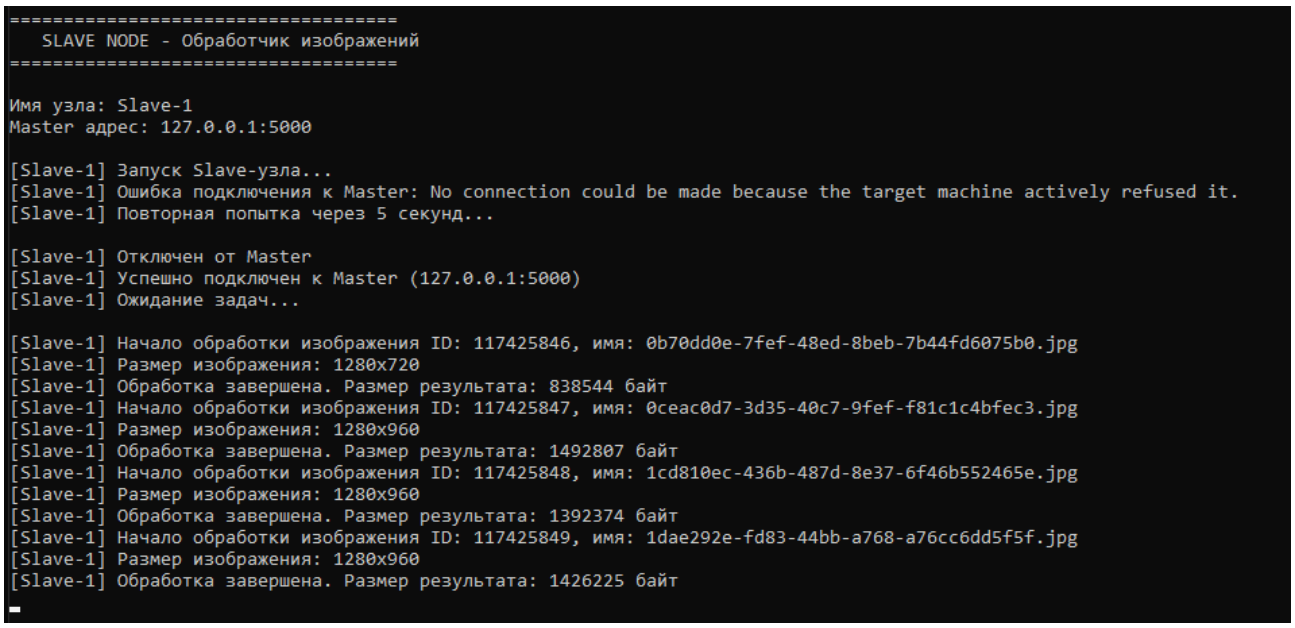

Рисунок Д.7 – логирование на *Slave*-узле