МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

РАБОТА ПРОВЕРЕНА
Рецензент
____
«___»__________________ 2019 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_______________ Г.И. Радченко
«___»__________ 2019 г.

Клиентская часть автоматизированной системы управления постобработкой изображений и видеофайлов

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Руководитель работы,
к.т.н., доцент каф. ЭВМ
________________В.А. Парасич
«___»___________2019 г.

Автор работы,
студент группы КЭ-452
________________В.И. Сулимов
«___»___________2019 г.

Нормоконтролёр,
ст. преп. каф. ЭВМ
________________ С.В. Сяськов
«___»____________2019 г.

Челябинск-2019

Аннотация

В.И. Сулимов. Клиентская часть автоматизированной системы управления постобработкой изображений и видеофайлов. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШЭКН; 2019, 46 с., 6 ил., библиогр. список – 18 наим.

В рамках данной работы была спроектирована и реализована клиентская часть автоматизированной системы управления постобработкой изображений и видеофайлов. Разработанный проект является частью большого сервиса по управлению постобработкой изображений и видеофайлов. Система была протестирована и введена в эксплуатацию на предприятии, занимающемся рендерингом. Система учитывает недостатки системы-предшественника и является ее улучшенным аналогом.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 6
ЦЕЛЬ И ЗАДАЧИ РАБОТЫ 7
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 9
1.1. ОБЗОР АНАЛОГА 9
1.2. ВЫВОД 11
2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ 12
2.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 12
2.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 14
3. ПРОЕКТИРОВАНИЕ 15
3.1. ОПИСАНИЕ ИСПОЛНЯЮЩЕЙ ЧАСТИ 16
3.2. ОПИСАНИЕ УПРАВЛЯЮЩЕЙ ЧАСТИ 17
3.3. ФУНКЦИОНИРОВАНИЕ СИСТЕМЫ 18
4. РЕАЛИЗАЦИЯ 21
4.1. РЕАЛИЗАЦИЯ ИСПОЛНЯЮЩЕЙ ЧАСТИ 21
4.2. РЕАЛИЗАЦИЯ УПРАВЛЯЮЩЕЙ ЧАСТИ 23
5. ТЕСТИРОВАНИЕ 27
5.1. МЕТОДОЛОГИИ ТЕСТИРОВАНИЯ 27
5.2. ТЕСТИРОВАНИЕ ИСПОЛНЯЮЩЕЙ ЧАСТИ 27
5.3. ТЕСТИРОВАНИЕ УПРАВЛЯЮЩЕЙ ЧАСТИ 27
6. ЗАКЛЮЧЕНИЕ 32
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 33
ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ВОРКЕРА WATERMARK 35
ПРИЛОЖЕНИЕ Б ИСХОДНЫЙ КОД КЛАССА MAIN УПРАВЛЯЮЩЕЙ ЧАСТИ 41
ПРИЛОЖЕНИЕ В ИСХОДНЫЙ КОД КЛАССА LAUNCHER УПРАВЛЯЮЩЕЙ ЧАСТИ 43
ПРИЛОЖЕНИЕ Г ИСХОДНЫЙ КОД МОДУЛЬНОГО ТЕСТА ДЛЯ ВОРКЕРА WATERMARK 46

ВВЕДЕНИЕ

В настоящее время компьютерная графика является часто используемым инструментом при создании видео. Фильмы и сериалы, мультфильмы и сцены для компьютерных игр, проморолики и реклама – везде, для того чтобы встретить примеры применения компьютерной графики, долго искать не придется. И везде, где необходимо применить компьютерную графику, особенно в случаях, когда она сложная и ее много, создатели упираются в длительный процесс обработки, требующий большого количества вычислительных ресурсов – рендеринг.

Рендеринг – это термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы[1]. Модель, в свою очередь, это описание любых объектов или явлений строго на определенном языке или в виде структуры данных. Любые параметры – тип источника света, направление, свойства материала смоделированного объекта, описываются различными методами в зависимости от программы для моделирования, а после превращаются в привычные глазу изображения благодаря процессу рендеринга.

Рендеринг – очень время- и ресурсозатратная процедура[2], поэтому совершенно предсказуемо появление ниши, предоставляющей услуги рендеринга на своих вычислительных мощностях за соответствующую плату. Компании, оказывающие подобные услуги, росли, конкуренция повышалась, рынок развивался. В следствие этого на данный момент на рынке существуют компании, которые оказывают услуги по совершению полного цикла преобразования 3D-сцен в видеофайл[3].

Помимо рендеринга в список оказываемых услуг входит большое количество второстепенных задач, необходимых для превращения модели в изображение. В рамках данной работы некоторая часть таких услуг будет называться постобработкой.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Целью проекта является разработка клиентской части автоматизированной системы управления постобработкой изображений и видеофайлов, которая полностью заменит существующую на предприятии систему постобработки.

Для реализации данной цели были поставлены следующие задачи:

1) проанализировать существующую систему, выделить её недостатки;
2) выполнить анализ требований;
3) выполнить проектирование клиентской части системы;
4) реализовать клиентскую часть системы;
5) протестировать разработанную систему.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

На предприятии, для которого был написан сервис, между моментом обращения клиента к компании и получением клиентом готового продукта проходит множество этапов. Конечно, наиболее крупный и важный этап – это рендеринг, но помимо рендеринга существует множество более мелких и менее заметных задач, которые, несмотря на это, очень важны и без них использование услуг компании могло бы быть гораздо менее приятным, а в некоторых случаях невозможным.

В данной выпускной квалификационной работе речь идет о задачах постобработки файлов. Необходимо понимать, что под постобработкой не подразумевается изменение «качества» изображения, постобработка не связана напрямую с рендерингом, хоть и использует продукты рендеринга. Речь идет о технических действиях, которые можно назвать обслуживающими, например, архивирование, разархивирование, создание «жестких» ссылок и так далее.

Автоматическое, безошибочное и быстрое выполнение этих задач очень важно для функционирования систем предприятия в целом.

1.1. ОБЗОР АНАЛОГА

Первоначальное требование к запрошенной заказчиком системе заключалось лишь в автоматизации процесса постобработки. Существовавшая на тот момент система, помимо многих имевшихся проблем, не была полностью автоматизированной, и часто требовала вмешательства специалиста для корректной работы, из-за чего время специалистов, которое предназначалось для других проектов, тратилось на поддержку неидеальной системы, для чего даже не требовалась серьезная квалификация.

Вскоре после начала работы над проектом, сразу же после того, как был начат анализ существовавшей системы, было выяснено, что система далеко не всегда оптимально распределяет ресурсы между задачами. Из-за чего одна

крупная задача, требующая значительное количество времени для ее выполнения, могла со временем забрать для своего выполнения все или практически все доступные в рамках одной ЭВМ вычислительные мощности процессора, что приводило к эффекту бутылочного горлышка[4]. Другие задачи не могли корректно выполняться, или выполнялись очень долго, что могло вызвать недовольство клиентов временем выполнения простой и дешевой задачи. Появилась цель избавиться от этой недоработки и реализовать алгоритм для оптимального распределения вычислительных ресурсов.

Также во время анализа старой системы выяснилось, что даже если удавалось избежать закрепления чрезмерно большого количества вычислительных мощностей за одной задачей, многие задачи выполнялись медленнее, чем это было возможно при тех же физических условиях. Это происходило по причине не оптимально спроектированных алгоритмов выполнения задач постобработки, из-за чего можно было наблюдать серьезные потери во времени работы процессора.

Конечно, помимо исправления ошибок работы прошлой системы, необходимо было, чтобы новая система выполняла все то, что прошлой системе успешно удавалось. В случае с клиентской частью – это прием запросов на выполнение задач от серверной части, их обработка, валидация и идентификация, в зависимости от исходов валидации и идентификации отказ от выполнения с соответствующими пояснениями, преобразование запросов в подходящий для выполнения задач вид, отправка преобразованного запроса на выбранный в процессе идентификации запроса исполняющий модуль, выполнение задачи, подтверждение и сообщение серверной части статуса выполнения задачи с информацией обо всех промежуточных этапах ее выполнения начиная с принятия запроса клиентской частью. Все это должно выполняться автоматически, с минимальным вмешательством специалистов в ход работы, следовательно, с минимальным количеством ошибок на пути выполнения задач.

1.2. ВЫВОД

Главная проблема, вставшая перед началом разработки – проблема бутылочного горлышка. Нивелирование эффекта бутылочного горлышка получило первостепенную важность и стало обязательным условием для новой системы. Также особое значение имеет скорость работы системы, стабильность работы и требование минимального вмешательства специалиста в работу системы.

2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

Учитывая проблемы, представленные для решения системой выше, были составлены следующие требования для клиентской части разрабатываемой системы:

1) возможность задать количество ресурсов процессора, доступных для каждого из процессов, выполняющих задачи;

2) принятие запросов на выполнение задач от сервера в установленном формате;

3) постановка в очередь задач, для запуска процесса обработки которых не хватает заданных ресурсов, без остановки выполнения менее ресурсоемких задач;

4) логирование процесса постобработки на каждом этапе;

5) длительное время работы системы без остановок и перезагрузок (от 720 часов).

2.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Функциональные требования определяют функциональность программного обеспечения, то есть описывают, какое поведение должна предоставлять разрабатываемая система. В ряду функциональных возможностей системы должны быть доступны:

1) жесткая линковка объектов методами «директория-директория», «файл-файл», «файл-директория»; под жесткой линковкой понимается инструмент ОС Linux, который позволяет создавать жесткую ссылку на файл, которая не перестанет функционировать после любых изменений исходного файла, в том числе его перемещения; технически можно представить это как один и тот же файл с двумя именами[5];

2) автоматическая идентификация типа архива с последующей разархивацией с заданными параметрами в целевые каталоги;

3) автоматическая архивация заданных по маске объектов в указанные каталоги с учетом заданных параметров;

4) автоматическое выполнение наложения водяного знака (watermark), выбранного клиентом, на заданный диапазон графических изображений определенных форматов;

5) создание видеофайла заданного формата из выбранного диапазона кадров путем склейки предоставленного списка изображений;

6) создание единственного видеофайла заданного формата из диапазона массива некоторого количества более мелких видеофайлов, находящихся в указанной директории, в результате работы воркера video; название «воркер» было дано модулям, которые оборачивают ПО, которое, непосредственно, выполняют какую-либо задачу; воркер занимается обслуживанием этого ПО, связывается с управляющей частью клиентской части системы, считывает все сообщения, которые выводит ПО, следит за корректностью выполнения задания;

7) «склеивание» изображения из страйпов (stripe), полученных в результате процесса обработки сцен; страйп – это фрагмент изображения, полученного в результате рендера статичной сцены, который выглядит как горизонтальная полоса; страйпы образуются в первую очередь при параллельном рендеринге одной статичной сцены несколькими процессами, каждый процесс соответственно рендерит свой страйп;

8) роутинг (routing) и форматирование принятых клиентской частью от серверной части запросов на соответствующий воркер в зависимости от типа запроса;

9) управление вычислительными ресурсами машины для нивелирования эффекта бутылочного горлышка;

10) логирование и отправка лог-сообщений на выделенный сокет сервера, содержащих в себе полные сведения и процент выполнения любого отдельно взятого задания.

2.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

В ряду нефункциональных требований были выделены следующие требования:

– система должна быть написана на языке C++;
– система должна работать на ОС Linux.

Требования были установлены заказчиком сервиса.

3. ПРОЕКТИРОВАНИЕ

На первых этапах проектирования произошло разделение клиентской части разрабатываемого сервиса на две логических части. Были выделены управляющая часть и исполняющая часть.

Исполняющая часть занимается выполнением задач постобработки. Она состоит из воркеров, каждый из которых выполняет свой тип задач. Задачи исполняющая часть получает от управляющей части, ей же она и возвращает результаты.

Управляющая часть обеспечивает стабильную работу исполняющей части и занимается «общением» с серверной частью сервиса. Управляющая часть занимается подключением к серверу, получением задач от сервера и их отправкой исполняющей части и получением результатов от исполняющей части. Логическая структура клиентской части представлена на рисунке 1.

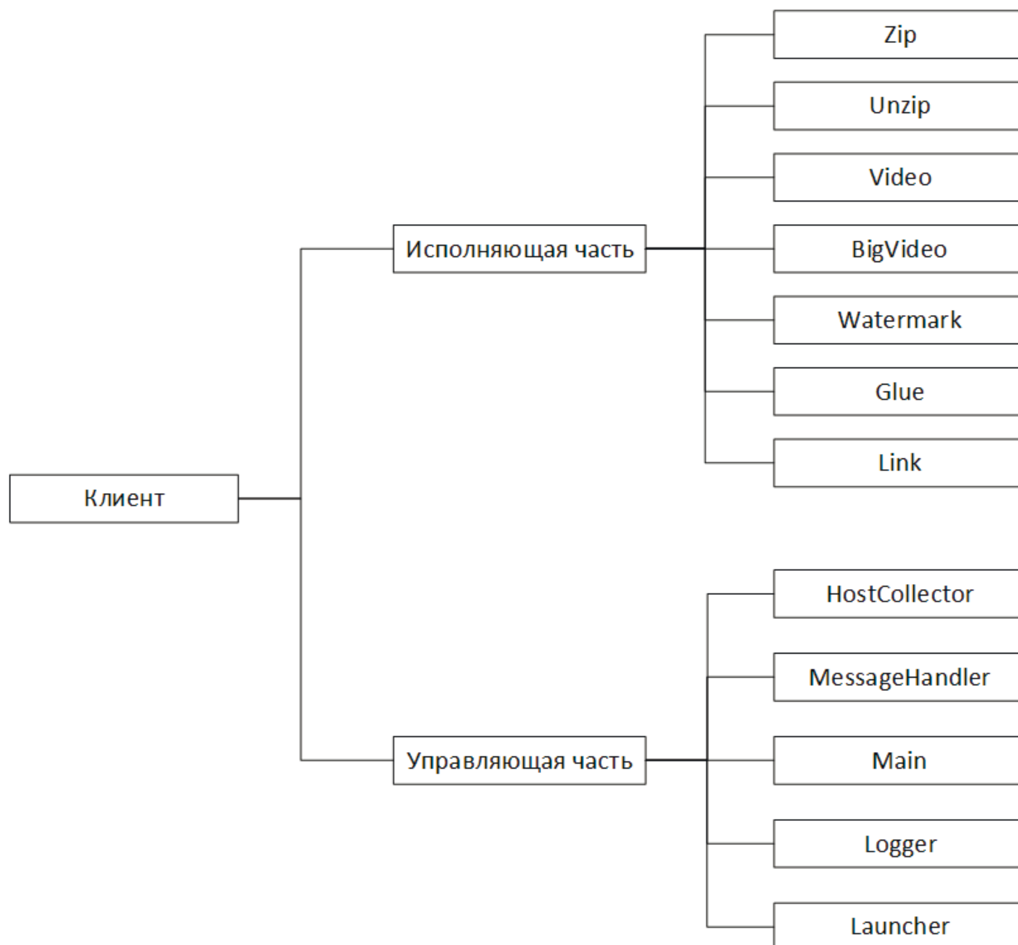

Рисунок 1 – Логическая структура исполняющей и управляющей частей

3.1. ОПИСАНИЕ ИСПОЛНЯЮЩЕЙ ЧАСТИ

Во время проектирования исполняющей части для нее были выделены семь задач постобработки, для каждой из которых был спроектирован свой воркер. В результате проектирования было создано семь воркеров:

- Zip;
- UnZip;
- Video;
- BigVideo;
- Watermark;
- Glue;
- Link.

Задача воркера Zip архивировать указанные файлы. Файлы должны быть указаны по маске с помощью регулярных выражений.

Задача воркера UnZip разархивировать заданный архив в указанную директорию.

Задача воркера Video склеить диапазон изображений в видеофайл. Для идентификации диапазона изображений воркер Video должен получать во входных данных директорию, в которой находятся изображения для склеивания, и префикс имени, по которому файлы будут искаться в указанной директории. Указание префикса имени файлов связано с правилом хранения файлов, полученных в процессе рендеринга видео, действующим в организации. Все изображения, полученные в результате рендеринга динамичной сцены, имеют имя, например, “example_0001.png”, “example_0002.png” и далее по тому же правилу. Префикс для изображений, относящихся к одной сцене, остается неизменным.

Задача воркера BigVideo склеить диапазон видеофайлов в один видеофайл. BigVideo, в свою очередь, не принимает на вход префикс для распознавания диапазона файлов для склеивания, а принимает только название директории. Это связано с тем, что BigVideo, при обычном его использовании, работает с

продуктами работы воркера Video, в связи с чем использование префикса становится излишним, так как воркер Video всегда сохраняет результат с префиксом “VideoWorker_”.

Задача воркера Watermark наложить водяной знак на указанный диапазон изображений. Также, согласно правилам, установленным в организации, иногда происходит ситуация, в которой нужно наложить два водяных знака на одно изображение с разным текстом. Воркер Watermark обязательно принимает на вход текст, который будет отображен на первом водяном знаке и опционально принимает текст для второго водяного знака.

Задача воркера Glue склеить изображение из страйпов. Страйп – это фрагмент изображения, полученный в процессе рендеринга. Выглядит страйп как горизонтальная полоса во всю ширину изображения. Фрагментация изображения на страйпы – это особенность многопоточного рендеринга, который используется для особенно крупных сцен, например, если нужно отрендерить 3D-модель архитектурного плана парка со множеством мелких деталей[6].

Задача воркера Link создать жесткую ссылку (hardlink). Для выполнения этой задачи нужно всего два параметра – исходный файл/каталог и целевой файл/каталог.

3.2. ОПИСАНИЕ УПРАВЛЯЮЩЕЙ ЧАСТИ

Управляющая часть спроектирована как посредник и «менеджер» исполняющей части. Также именно она отвечает за работу клиентской части как процесса в целом.

Первое, что происходит после запуска клиентской части – сбор информации о хосте. После запуска клиентской части HostCollector собирает такую информацию как имя хоста, количество доступных ядер и названия доступных воркеров. Вся информация сериализуется и отправляется серверу в приветственном сообщении. IP-адрес сервера и его порт задаются как аргументы при запуске клиентской части. В случае успешного подключения клиента к

серверу управляющая часть переходит в режим ожидания сообщения. Когда приходит сообщение от сервера управляющая часть его десериализует, валидирует, распознает его тип и предпринимает необходимые действия в зависимости от вида сообщения. Если сообщение пришло валидное, с одним из типов задач постобработки внутри, то такое сообщение переформируется и отправляется на соответствующий воркер. Если сообщение валидное, с типом “shutdown”, то управляющая часть ожидает пока закончат свое выполнение уже начатые задачи, если такие существуют, и завершает процесс клиента, уведомляя об этом серверную часть. Если сообщение пришло валидное, но с типом задачи, который не может выполниться на данном экземпляре клиента, то управляющая часть уведомляет об этом сервер и ожидает дальнейших сообщений. В случае получения невалидного сообщения управляющая часть также сообщает об этом серверу и продолжает ожидание сообщений.

Также именно управляющая часть выполняет свою часть решения проблемы бутылочного горлышка. При приеме сообщения MessageHandler считывает запрашиваемое количество ядер для поступившей задачи, если этот параметр присутствует, и передает его, среди всей остальной информации, модулю Launcher. Launcher в свою очередь устанавливает конкретные ядра для запускаемой задачи, таким образом ни одна задача не сможет забрать больше вычислительных ресурсов процессора чем ей было указано, что позволяет избежать эффекта бутылочного горлышка. Это часть общего решения проблемы бутылочного горлышка, также в решении проблемы принимает участие и серверная часть.

3.3. ФУНКЦИОНИРОВАНИЕ СИСТЕМЫ

В ходе проектирования клиентской части разрабатываемой системы был описан полный цикл работы клиентской части от момента запуска до завершения.

Запуск приложения происходит с обязательным указанием IP-адреса и порта сервера, с которым клиент будет взаимодействовать. Сразу после запуска

инициализируется система логирования, далее происходит опрос ЭВМ, на которой был запущен клиент, для выяснения объема вычислительных мощностей, предоставленных системе. Клиентская часть системы также «обследует» сама себя для установления перечня доступных ей воркеров. После выяснения всей необходимой информации, включая также версию приложения и имя хоста, происходит отправка приветственного сообщения по IP-адресу и порту, указанным во время запуска процесса. Приветственное сообщение включает в себя всю собранную ранее информацию.

В случае успешного подключения к серверу и получения ответного приветственного сообщения запускается цикл ожидания сообщений от сервера. Выход из цикла возможен только при получении от сервера сообщения с задачей “shutdown”. При получении сообщения от сервера управляющая часть клиентской части производит его валидацию, тем самым определяя соответствует ли сообщение установленному формату и способен ли этот экземпляр клиентской части его обработать. После валидации происходит идентификация типа задачи, поставленной для выполнения пришедшим сообщением. Клиентская часть может обработать только те задачи, тип которых соответствует типам доступных воркеров, их список устанавливается при запуске клиентской части и задачи с типом “shutdown”.

После идентификации типа задачи инициализируется соответствующий воркер, которому при запуске передается сообщение от серверной части, включающее в себя необходимую для работы воркера информацию, такую как исходные и целевые каталоги, id задачи, а также специфичные для каждого воркера данные. Также ограничивается количество ядер процессора, доступных инициализированному воркеру.

Воркер выполняет задачу, после чего отправляет отчет о выполнении управляющей части и завершается. Управляющая часть анализирует отчет и отправляет уведомление о завершении работы над задачей серверной части. Ожидание сообщений от сервера продолжается.

Упрощенная схема работы клиентской части приведена на рисунке 2.

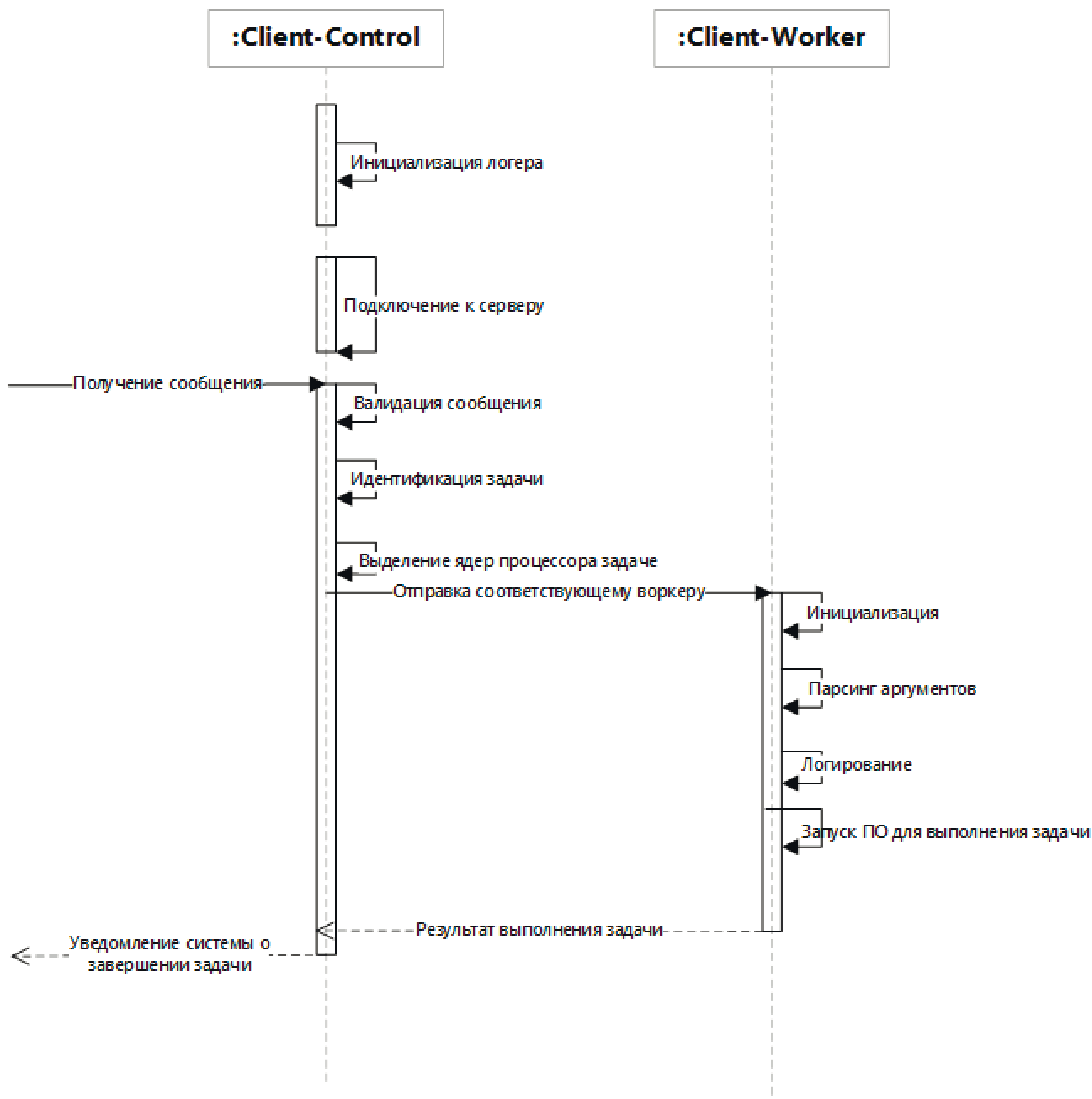

Рисунок 2 – Схема работы клиентской части

4. РЕАЛИЗАЦИЯ

4.1. РЕАЛИЗАЦИЯ ИСПОЛНЯЮЩЕЙ ЧАСТИ

Для исполняющей части было реализовано семь модулей воркеров. Общий принцип работы каждого из них в основном совпадает, но есть также и серьезные различия. Каждый из воркеров, кроме воркера Link, использует стороннее программное обеспечение для выполнения самой задачи. То есть, непосредственно сам процесс склейки кадров или наложения водяного знака происходит благодаря стороннему программному обеспечению; все, что нужно было для запуска этого стороннего ПО, для сбора информации после завершения работы, структурирования и отправки этой информации – работа самого воркера.

Воркеры Zip и UnZip используют приложение 7zip. 7zip был выбран ввиду своей распространенности, открытого исходного кода и бесплатного распространения[7]. Открытый исходный код позволяет быть уверенным в безопасности используемого программного обеспечения[8], что немаловажно при использовании ПО на предприятии, но также открытый исходных код позволяет вносить в него свои изменения, что и было использовано при написании этого проекта.

После начала использования 7zip в этом проекте и при первых тестовых заданиях возникло две проблемы: от 7zip было непросто получить индикацию хода распаковки-запаковки архива в процентах[9], а также гораздо более важная проблема, связанная с устройством сетевого хранилища данных предприятия. Для того чтобы не занимать сетевое хранилище данных огромным количеством ненужной информации, а также чтобы не приходилось удалять ее вручную, на предприятии давно работает система, которая удаляет все файлы, с даты изменения которых прошло больше двух недель. Иногда клиенты могут присылать архив на распаковку, в котором лежат файлы, не изменявшиеся гораздо больше двух недель, что приводило к тому, что файлы удалялись после их распаковки. Эта проблема, как и проблема с выводом индикации выполнения

запаковки-распаковки архива, была решена с помощью патчей для 7zip.

Для того, чтобы от 7zip всегда можно было получить индикацию выполнения в процентах была подменена проверка на терминал. Опция IsStdOutTerminal получила значение true навсегда, чтобы 7zip «думал», что все время подключен к терминалу. В таком случае 7zip отображает индикацию хода выполнения в процентах. Также при распаковке архива 7zip сразу же изменяет файлам и каталогам дату изменения на дату распаковки. Таким образом решается проблема с удалением файлов, которые были нужны для использования.

Воркеры, связанные с обработкой или генерацией видеофайлов, используют FFmpeg. FFmpeg – это набор свободных библиотек с открытым исходным кодом, которые позволяют записывать, конвертировать и передавать цифровые аудио и видеозаписи в различных форматах[10]. Выбор FFmpeg также обусловлен открытым исходным кодом и в связи с этим отсутствием платы за использование.

Воркеры, связанные с обработкой изображений, такие как Glue и Watermark, используют консольную утилиту Convert от ImageMagick. ImageMagick — набор программ (консольных утилит) для чтения и редактирования файлов множества графических форматов. Является свободным и кроссплатформенным программным обеспечением[11]. Выбор также обусловлен открытым исходным кодом и отсутствием платы за использование.

Воркер Link использует системный вызов ядра Linux link(2)[12] для создания жестких ссылок ввиду того, что создание жестких ссылок – очень частая задача постобработки, которую необходимо выполнять как можно быстрее.

Необходимо также указать, что по итогам реализации исполняющей части каждый воркер был реализован как отдельный исполняемый файл, и, при необходимости, каждый воркер может работать как отдельное приложение.

Исходный код воркера `Watermark` отображен в листинге А.3 приложения А.

4.2. РЕАЛИЗАЦИЯ УПРАВЛЯЮЩЕЙ ЧАСТИ

Так как одна из основных задач управляющей части – общение с серверной частью, необходимо было определиться с «языком» общения. «Языком» общения был выбран JSON, для более эффективной работы с JSON в C++ был использован пакет `nlohmann:json`. Это проект с открытым кодом, стабильно обновляющийся разработчиком[13].

Пример JSON-сообщения, передаваемого от серверной части системы к клиентской части системы, представлен на листинге 1.

Листинг 1 – JSON-сообщение о задаче типа Watermark

```
{
    "uid":"7270ca3a-72aa-49bd-ab18-16479e2d2420",
    "id":"12345",
    "type":"task",
    "src":"/mnt/ferma/srctest",
    "dst":"/mnt/ferma/dsttest",
    "first":"demo mode",
    "second":"test test",
    "subtype":"watermark",
}
```

Сообщения о других типах задач выглядят идентично, иногда добавляются дополнительные поля, например, для задачи типа Video есть возможность указать расширение выходного файла и частоту кадров. Но основной состав полей остается неизменным для всех задач.

Поле “`uid`” – глобальный id задачи, который используется в том числе вне системы управления постобработкой, предоставляется серверной частью.

Поле “`type`” – тип сообщения, отправленного серверной частью. Возможны типы “`task`” для задачи, “`log`” для лог-сообщений и “`shutdown`”

для сообщения о выключении клиентской части.

Поле “src” – путь к исходному каталогу, поле присутствует для сообщений об абсолютно всех типах задач. В данном случае “src” – это каталог, в котором находятся изображения, на которые нужно наложить водяной знак.

Поле “dst” – путь к целевому каталогу, также присутствует для сообщений о любых задачах. В данном случае указывает куда сохранить изображения с наложенными водяными знаками. Очень удобно в случаях, когда последнее, что нужно сделать с изображениями перед демонстрацией клиенту – это наложить водяные знаки. В таком случае изображения сразу же могут быть перемещены в каталог, доступный заказчику для последующей демонстрации.

Поля “first” и “second” – это, соответственно, текст для первого и второго водяного знака, который будет наложен на изображение. Возможность накладывать до двух изменяемых водяных знаков – требование заказчика.

Поле “subtype” – обязательное поле для всех сообщений с типом “task”, хранит в себе тип задачи, которую необходимо выполнить. Возможны варианты “video”, “bigvideo”, “zip”, “unzip”, “watermark”, “glue” и “link”.

Так как устанавливать различные стандарты передачи сообщений внутри одной системы – излишнее усложнение системы, все передаваемые сообщения, в том числе и внутри клиентской части, имеют JSON-формат. Пример лог-сообщения представлен на листинге 2.

Листинг 2 – пример лог-сообщения


```
{
    "id":12345,
    "level":"info",
    "message":"Запущен с аргументами:",
    "time":"2019-05-26.15:01:28",
    "type":"log"
}
```


Поле “id” – id задачи, необходимо для того, чтобы серверная часть могла идентифицировать лог-сообщение и определить его как лог-сообщение о

конкретной задаче.

Поле “level” – уровень логирования. Всего существует 4 уровня логирования: “info”, “warning”, “error” и “percent”. Соответственно уровень “info” – информационное сообщение, “warning” – сообщение-предупреждение, указывает на произошедшую ошибку, которая не является критической для выполнения задачи, “error” – сообщение-уведомление о том, что выполнение задачи завершилось с ошибкой или не началось, “percent” – уровень логирования для индикации хода выполнения задач. Некоторые задачи, в которых возможно посчитать прогресс выполнения, используют этот уровень логирования, отправляя только число от 0 до 100.

Поле “message” – само лог-сообщение, включает в себя только одну строку.

Поле “time” – время отправки лог-сообщения.

Поле “type” – такое же поле, как и в сообщении о задаче, означает тип сообщения.

В управляющей части клиентской части системы можно выделить 4 класса, которые используются при постоянной работе клиента. Такие классы как HostCollector, которые используются только при запуске системы, можно опустить, поскольку они не так сильно влияют на продолжительную работу клиентской части. В основные классы входят:

- “MessageHandler” – класс-обработчик сообщений, его функции вызываются, когда приходят сообщения от сервера, они, в свою очередь, решают, валидное ли пришло сообщение, идентифицируют его и выбирают, кому из воркеров его отправить, если это сообщение о задаче;
- “Launcher” – класс, который работает после класса “MessageHandler”, получив от него информацию с именем воркера для запуска; именно этот класс ограничивает количество ядер для запускаемого процесса, тем самым решая проблему бутылочного горлышка для клиентской части, после чего запускает необходимого воркера;

- “Main” – основной класс, именно в нем постоянно запущен цикл ожидания сообщений от сервера, также он отвечает за первоначальный запуск системы в целом;
- “Logger” – класс-логер, используется практически всеми модулями в проекте, объект этого класса инициализируется первым после запуска клиента как процесса, и первое, что делает клиент – формирует лог-сообщение о своем запуске.

Исходный код классов “Main” и “Launcher” представлен в листингах Б.4 и В.5 приложений Б и В соответственно.

5. ТЕСТИРОВАНИЕ

5.1. МЕТОДОЛОГИИ ТЕСТИРОВАНИЯ

В связи с тем, что каждый воркер представляет из себя отдельный исполняемый файл, для исполняющей части клиентской части системы была выбрана методология модульного тестирования. Для управляющей части была выбрана методология приемочного тестирования[14], которое проводилось интеграционной группой разработчиков предприятия.

5.2. ТЕСТИРОВАНИЕ ИСПОЛНЯЮЩЕЙ ЧАСТИ

Для каждого воркера исполняющей части клиентской части системы был написан модульный тест с использованием Google C++ Testing Framework или GTest. GTest – это библиотека для модульного тестирования на языке C++, построенная по методологии тестирования xUnit[15], то есть когда отдельные части программы проверяются отдельно друг от друга, в изоляции. Исходный код Google Test также является открытым[16], IDE CLion, использовавшаяся при написании проекта, удобно интегрируется с Google Test[17] и функционал Google Test полностью покрывает все необходимые для заказчика тесты исполняющей части[18], поэтому был выбран именно этот фреймворк.

Исходный код теста, написанного для воркера Watermark, приведен в листинге Г.6 приложения Г.

5.3. ТЕСТИРОВАНИЕ УПРАВЛЯЮЩЕЙ ЧАСТИ

Тестированием управляющей части системы занимались разработчики, которые были ответственны за внедрение системы в системную структуру предприятия. Поэтому была выбрана методология приемочного тестирования.

Для определения уровня готовности системы к эксплуатации системы был написан монитор – фронтенд-приложение с визуальным интерфейсом, отображающее некоторую информацию о системе управления постобработкой в целом. Главный экран интерфейса представлен на рисунке 3.

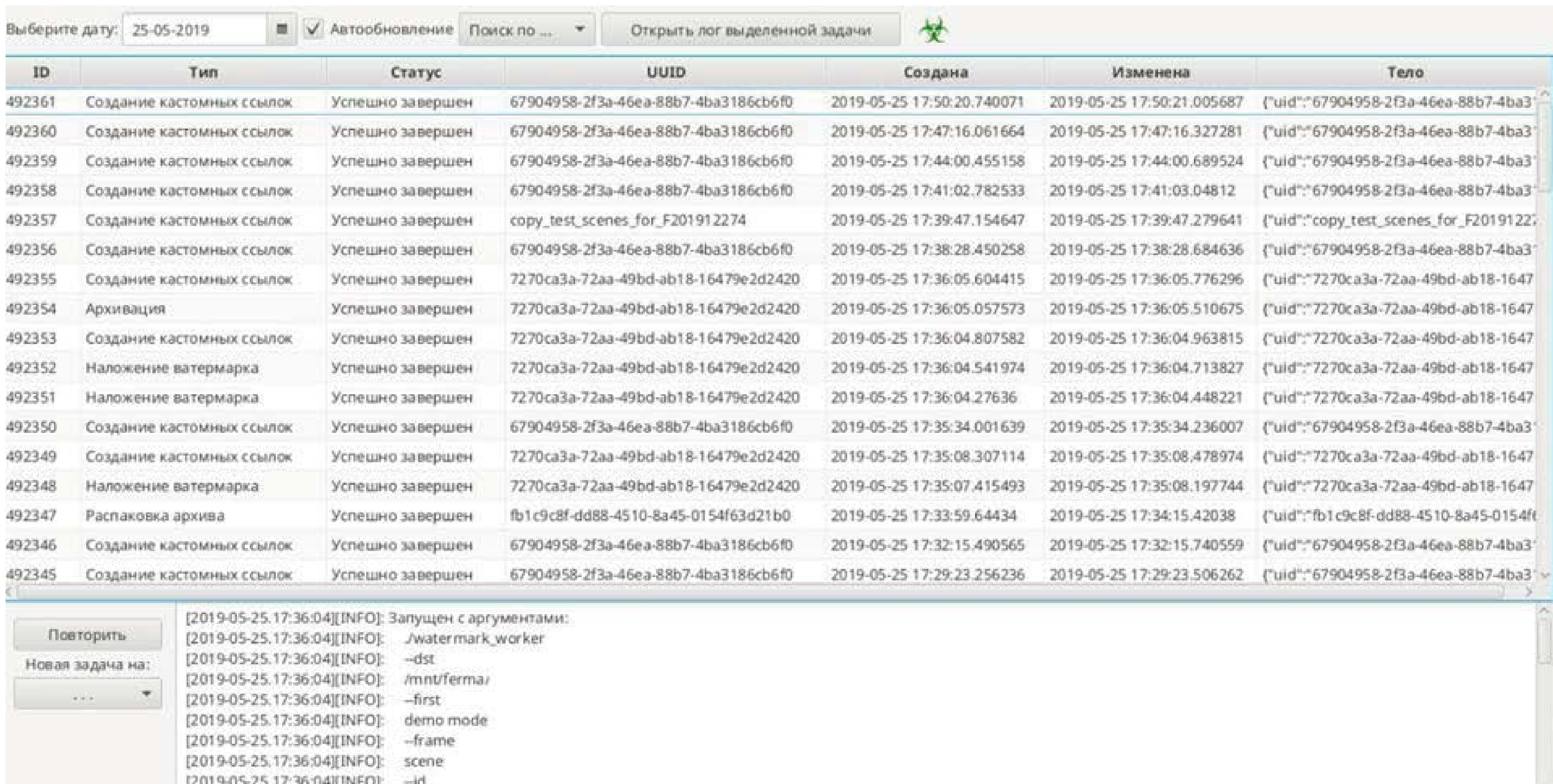

Рисунок 3 – Интерфейс монитора систем

На мониторе можно видеть краткий список последних завершенных задач, их ID, тип задачи, приведенный к человекочитаемому формату, статус завершения задачи, ее глобальный ID, дату создания сообщения о задаче, дату изменения сообщения о задаче и тело задачи, то есть JSON-сообщение, которое было отправлено серверной частью клиентской части системы. Для тестирования системы монитору была добавлена функция отправки тестовой задачи клиентской части, чтобы можно было увидеть реакцию системы на пришедшее сообщение. Окно с шаблоном задачи для отправки изображено на рисунке 4.

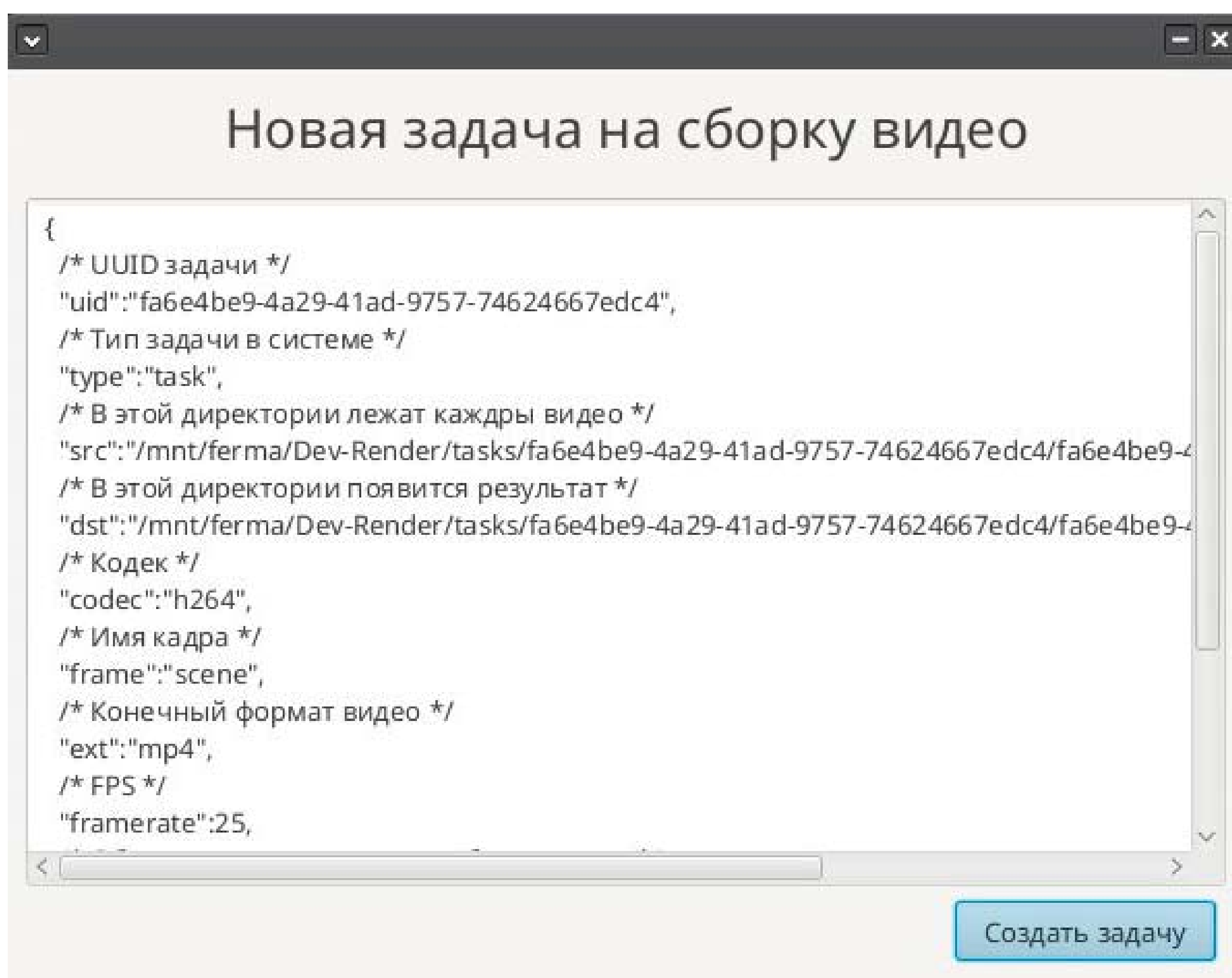

Рисунок 4 – Шаблон тестовой задачи

После нажатия на кнопку «Создать задачу» произойдет отправка JSON-сериализованного сообщения о задаче клиентской части, после выполнения которой информация о ней появится на главном экране монитора.

Монитор имеет шаблоны для каждого типа задачи, поддерживаемого исполняющей частью клиентской части системы.

Монитор отображает все функционирующие экземпляры клиента на экране подключенных клиентов. Так как одной из особенностей системы управления постобработкой в целом была масштабируемость, доступно подключение практически неограниченного количества клиентов. Предел – вычислительные мощности оборудования. На рисунке 5 можно увидеть информацию о функционировании трех экземпляров клиента, количество доступных воркеров которых отличается из-за разного количества вычислительных ресурсов, доступных каждому из экзмепляров.

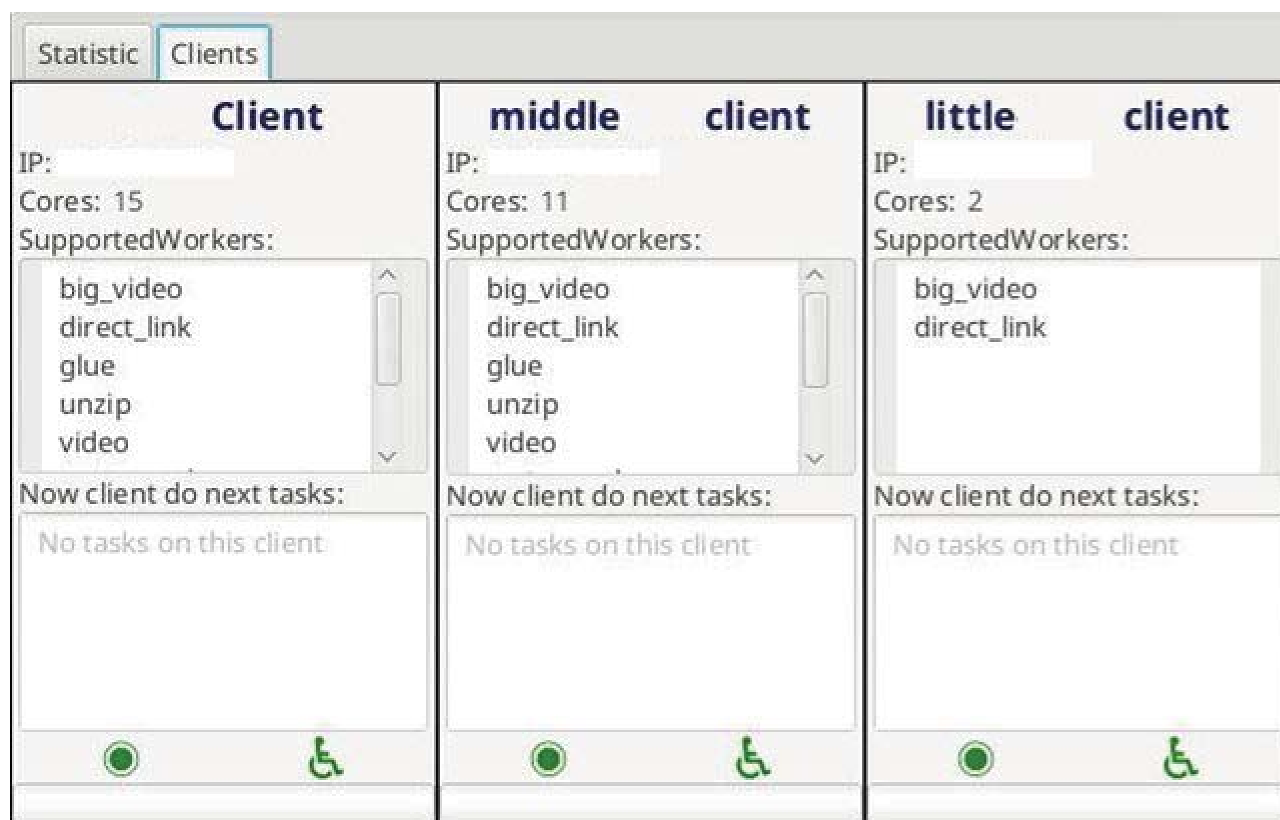

Рисунок 5 – Функционирующие экземпляры клиентов

Так как воркеры BigVideo и Link являются наименее требовательными к вычислительным ресурсам – они были запущены на экземпляре клиента с двумя доступными ядрами процессора. Этот экземпляр является приоритетным для этих задач, таким образом он разгружает другие экземпляры клиентов, которые способны выполнять крупные задачи на всех доступных ядрах.

Также на специальном экране монитора можно увидеть суммарную статистику загруженности всех экземпляров клиентов в целом за последние сутки.

Скриншот экрана со статистикой загруженности экземпляров клиентов приведен на рисунке 6.

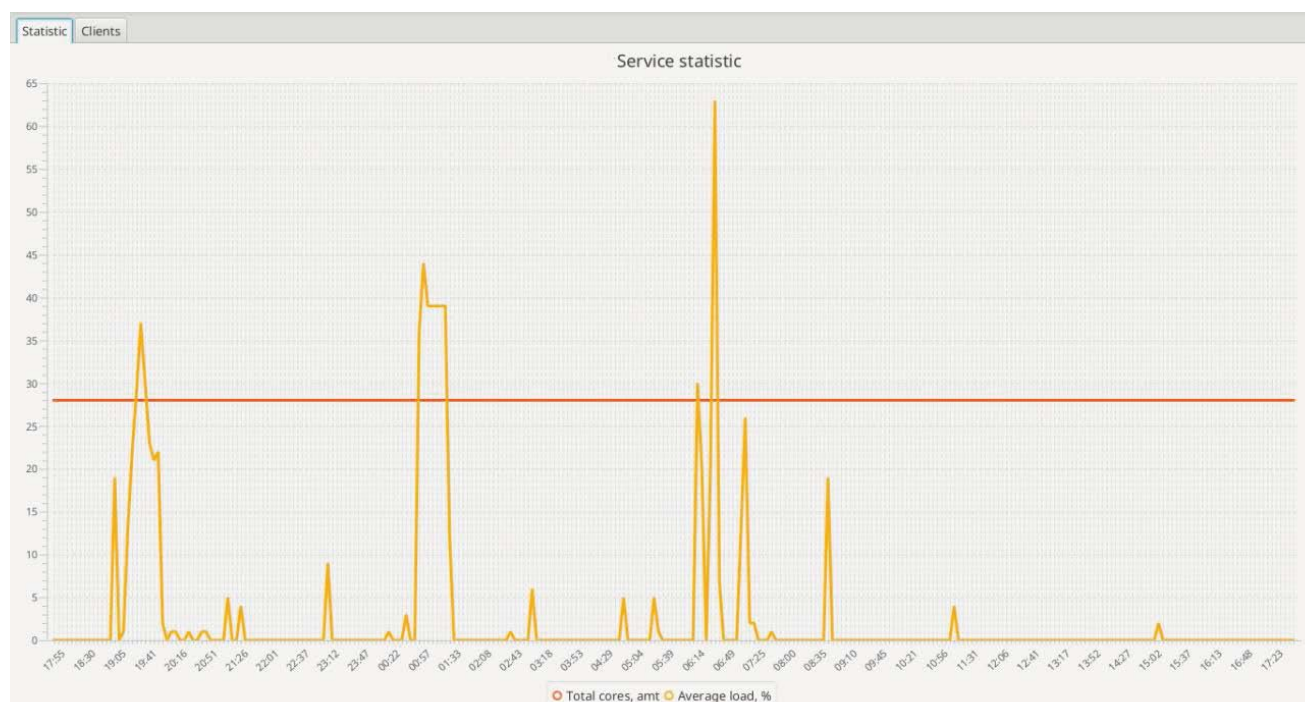

Рисунок 6 – Экран статистики загруженности экземпляров клиентов

Как видно из графика загруженности, текущая система при обычном режиме работы изредка нагружается до 60-65% от всей ее мощности, при этом такая нагрузка обычно спадает в течение 10-15 минут. Это является удовлетворительным результатом работы системы. За все время работы системы с момента ее интеграции на предприятии загруженность выше 95% ни разу не длилась больше 10-15 минут, что было отмечено заказчиком как удовлетворительный результат решения проблемы бутылочного горлышка.

6. ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы была разработана клиентская часть автоматизированной системы управления постобработкой изображений и видеофайлов. При этом были решены следующие задачи:

– проведен анализ аналога системы-предшественника, выделены основные проблемы;
– проведено формирование и согласование требований к системе;
– выполнено проектирование системы;
– выполнена реализация системы;
– проведено тестирование исполняющей части системы, подтвердившее корректность функционирования реализованного продукта;
– осуществлена консультация по интеграции системы в предприятии.

Проблема бутылочного горлышка была полностью решена согласно решению заказчика, что подтвердила демонстрация работы системы. Быстродействие системы выросло относительно системы-предшественника с тем же функционалом. Также была достигнута возможность масштабируемости системы, что не было возможным в условиях старой системы.

Перспективы развития системы:

– написание новых воркеров в случае появления новых задач постобработки;
– дальнейшая масштабируемость системы в случае увеличения объема работы предприятия.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. What is Rendering?. – https://www.techopedia.com/definition/9163/rendering. Дата обращения: 19.04.2019.
2. Компьютерная графика и анимация. – https://render.ru/xen/threads/stoimost-vremja-rendera.5679/. Дата обращения: 17.04.2019.
3. Компьютерная графика и анимация. – https://render.ru/ru/articles/post/13251. Дата обращения: 19.04.2019.
4. World of Computer Hardware. – https://pc-builds.com/everything-you-need-to-know-about-pc-bottlenecks/. Дата обращения: 26.04.2019.
5. RTFM: Linux, DevOps and system administration. – https://rtfm.co.ua/unix-chto-takoe-symlink-hardlink-i-inode/. Дата обращения: 10.05.2019.
6. Компьютерная графика и анимация. – https://render.ru/xen/threads/backburner-raznye-strajpy.136261/. Дата обращения: 17.04.2019.
7. 7-zip. – https://www.7-zip.org/. Дата обращения: 26.04.2019.
8. Открытые системы. – https://www.osp.ru/os/2005/12/380658/. Дата обращения: 10.05.2019.
9. Linux Documentation. – https://linux.die.net/man/1/7z. Дата обращения: 22.04.2019.
10. GitHub – FFmpeg. – https://github.com/FFmpeg/FFmpeg. Дата обращения: 22.04.2019.
11. GitHub – ImageMagick6. – https://github.com/ImageMagick/ImageMagick6. Дата обращения: 22.04.2019.
12. Linux Manual Page. – http://man7.org/linux/man-pages/man2/link.2.html. Дата обращения: 13.05.2019.
13. GitHub - nlohmann/json. – https://github.com/nlohmann/json . Дата обращения: 22.04.2019.
14. Аплана, тестирование ПО. – http://aplana.ru/services/testing/priemochnoe-testirovanie. Дата обращения: 17.04.2019.
15. xUnit.net. – https://xunit.net/. Дата обращения: 19.04.2019.

16. GitHub – GoogleTest. – https://github.com/google/googletest. Дата обращения: 21.04.2019.
17. Записки программиста. – https://eax.me/cpp-gtest/. Дата обращения: 03.05.2019.
18. IBM – Российская Федерация. – https://www.ibm.com/developerworks/aix/library/au-ctools1_boost/?S_TACT=105AGX99&S_CMP=CP. Дата обращения: 03.05.2019.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ВОРКЕРА WATERMARK

Листинг А.3 - Watermark

```
//
// Created by Владислав on 29.04.2019.
//
#include <errno.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

#include <map>
#include <regex>
#include <chrono>
#include <thread>
#include <string>
#include <iostream>
#include <algorithm>

#include "watermark.h"
#include "../Worker.h"

std::string create_watermark(std::string tmp_dir, std::string first, std::string second) {
    std::string text1 = "text 0,5 \"" + first + "\"";
    std::string text2 = "text 0,20 \"" + second + "\"";
    std::string file = tmp_dir + "/" + first + "_" + second + "_" +
std::to_string(time(NULL)) + ".miff";
    std::string out = "miff:" + file;
    const char* convert_cmd[] = {
            "/usr/bin/convert", "-size", "140x70", "xc:none", "-pointsize", "12", "-fill",
"grey",
            "-gravity", "North", "-draw", text1.c_str(),
            "-gravity", "North", "-draw", text2.c_str(),
            out.c_str(), NULL
    };
    LOG(LINFO) << convert_cmd << std::endl;
    int pid = fork();
    switch (pid) {
        case -1:
            LOG_ERRNO("Ну удалось форкнуться");
            return "";
        case 0: {
            execv(convert_cmd[0], (char *const *) convert_cmd);
            LOG_ERRNO("Не удалось exec'нуть convert")
            _exit(DONE_CRITICAL);
        }
        default: {
            int result;
            /* Сначала дожидаемся завершения convert */
            waitpid(pid, &result, 0);
            result = WEXITSTATUS(result);
            if(result)
                return "";
        }
    }
    return file;
}
```

```
/* Обрабаывает один из файлов в пуле потоков */
int watermark_file(std::string file, std::string dst_dir, std::string watermark_file) {
    /* Вычленяем имя файла */
    int pos = file.find_last_of("/");
    std::string output = dst_dir + "/" + file.substr(pos + 1);

    LOG(LINFO) << "Накладываем ватермарку " << watermark_file << " на файл " << file << ".
Результирующий файл записывается "
                  "в " << output << std::endl;

    const char* composite_cmd[] = {
        "/usr/bin/composite", "-tile", watermark_file.c_str(), file.c_str(),
output.c_str(), NULL
    };
    LOG(LINFO) << composite_cmd << std::endl;
    int pid = fork();
    switch (pid) {
        case -1:
            LOG_ERRNO("Ну удалось форкнуться");
            return DONE_CRITICAL;
        case 0:
            execv(composite_cmd[0], (char* const*)composite_cmd);
            LOG_ERRNO("Не удалось exec'нуть composite")
            _exit(DONE_CRITICAL);
        default: {
            int result;
            waitpid(pid, &result, 0);
            result = WEXITSTATUS(result);
            LOG(LINFO) << "convert завершился с кодом возврата " << result << std::endl;
            return result;
        }
    }
}

/* Распихивает файлы по пулу потоков */
int watermark_vector(std::vector<std::string> files, std::string dst_dir,
                     std::string first, std::string second) {

    /* Заранее создаём файл для заполнения */
    std::string watermark = create_watermark("/tmp", first, second);
    if(watermark == "")
        return DONE_CRITICAL;

    int error = 0;

    /* Данная магия выполняется в один поток, потому что так быстрее */
    auto it = files.begin();
    while(it != files.end()) {
        int exitcode = watermark_file(*it, dst_dir, watermark);
        it++;
        exitcode = WEXITSTATUS(exitcode);
        if(error == 0) {
            error = exitcode;
        }
    }
    if(error == 0) {
        LOG(LINFO) << "Так как ошибок нет, удаляем промежуточный файл с ватермаркой." <<
std::endl;
        unlink(watermark.c_str());
    }
    return error;
}
```

```
/* Проверяем, что расширение верное по регулярке*/
std::regex regex_file_to_watermark("^.*\\.(jpeg|jpg|gif|tif|tiff|exr|png|jpe|bmp|tga)$",
std::regex::icase);
/* Функция рекурсивно (ну или нет) обыскивает каталог на предмет разрешенных файлов */
void find_images(std::string dir, std::vector<std::string> & list, bool recursive = true) {
    /* Открываем каталог */
    DIR* d = opendir(dir.c_str());
    if(d == NULL) {
        LOG_ERRNO(dir);
        _exit(DONE_CRITICAL);
    }
    errno = 0;
    struct dirent* entry;
    while((entry = readdir(d)) != NULL) {
        std::string entry_name = (entry->d_name);
        /* Заведомо пропускаем текущий каталог и родительский */
        if (entry_name == "." || entry_name == "..") {
            continue;
        }
        /* А вдруг это каталог? */
        if(entry->d_type & DT_DIR) {
            /* Если нужно, запускаем рекурсию! */
            if(recursive)
                find_images(dir + "/" + entry_name, list);
            continue;
        }
        /* Нет, не каталог. Проверим, что матчится с регуляркой */
        if(std::regex_match(entry_name, regex_file_to_watermark)) {
            list.push_back(dir + "/" + entry_name);
        }
    }
    closedir(d);
    if(entry == NULL && errno != 0) {
        LOG_ERRNO("Проищошла ошибка поиска файлов");
        return _exit(DONE_CRITICAL);
    }
}

int watermark(std::string id,
              std::string uid,
              std::string src_dir,
              std::string dst_dir,
              std::string first,
              std::string second) {
    /* Выводим пришедшую информацию */
    LOG_TASK("Задача на наложение ватермарок", id, uid);

    LOG(LINFO) << "Запускаем задачу..." << std::endl;

    /* Создаём каталог для хранения результатов */
    auto start = std::chrono::steady_clock::now();

    check_dst(dst_dir);

    LOG(LINFO) << "Ищем файлы..." << std::endl;

    /* Находим исходники */
    std::vector<std::string> src_list;
    find_images(src_dir, src_list);

    LOG(LINFO) << "В исходной директории найдено " << src_list.size() << " файлов " <<
std::endl;
```

```
    /* Находим файлы в целевом каталоге */
    std::vector<std::string> dst_list;
    find_images(dst_dir, dst_list, false);

    LOG(LINFO) << "В целевом каталоге найдено " << dst_list.size() << " файлов " <<
std::endl;

    /* Удаляем из src_list то, что есть в dst_list, то есть уже с ватермарками */
    for(size_t i = 0; i < dst_list.size(); i++) {
        std::remove(src_list.begin(), src_list.end(), dst_list[i]);
    }

    LOG(LINFO) << "На обработку пойдёт " << src_list.size() << " файлов " << std::endl;

    int result = watermark_vector(src_list, dst_dir, first, second);

    auto end = std::chrono::steady_clock::now();
    LOG(LINFO) << "Задача завершена " <<  (result == 0 ? "успешно" : "с ошибкой") << " за "
<< std::chrono::duration <double, std::milli> (end-start).count() / 1000 << "с" << std::endl;
    return result;
}

//
// Created by Владислав on 30.04.2019.
//
#include <getopt.h>
#include <unistd.h>

#include <string>
#include <iostream>

#include "watermark.h"
#include "../Worker.h"


/* Назначаем значения по умолчанию */
struct args_t {
    /* Режимы */
    bool work = true;    /* Режим работы */
    bool test = false;  /* Режим проверки окружения */
    /* Обязательные параметры */
    std::string id;
    std::string uid;
    std::string src;
    std::string dst;
    std::string first;
    std::string second;
} args;

/**
 * Эта функция проверяет, что данный воркер спсобен работать в окружении, в котором запущен
 * @return код ошибки из массива test_errors
 */
std::string test_errors[] = {
        "Окружение подходит",
        "shell недоступен",
        "convert не найден\n",
        "composite не найден\n",
};
int test_enviroment() {
    if(system(NULL) == 0) {
        return 1;
```

```
    }
    if(system("type convert >/dev/null") != 0) {
        return 2;
    }
    if(system("type composite >/dev/null") != 0) {
        return 2;
    }
    return 0;
}

/* Список ключей */
static const char* optString = "i:u:s:d:F:S:ThR";
/* Список длинный ключей */
static const struct option longOpts[] = {
        { "id", required_argument, NULL, 'i' },       /* id задачи в базе данных*/
        { "uid", required_argument, NULL, 'u' },      /* UID как на сайте */
        { "src", required_argument, NULL, 's' },      /* Исходный каталог */
        { "dst", required_argument, NULL, 'd' },      /* Целевой каталог */
        { "first", required_argument, NULL, 'F' },    /* Первый ватермарк */
        { "second", required_argument, NULL, 'S' },   /* Второй ватермарк */
        { "test", no_argument, NULL, 'T' },           /* Протестировать окружение */
        { "help", no_argument, NULL, 'h' },           /* Вызывать справку */
        { "reg", no_argument, NULL, 'R' },            /* Для клиента */
        { NULL, no_argument, NULL, 0 }
};
/* Проверка, что все обязательные ключи прописаны */
bool opt_check() {
    /* В режиме тестирования нам безразличны остальыне ключи */
    if(args.test) {
        return true;
    }
    OPT_CHECK(args.id.empty(), "Пропущен ключ id!")
    OPT_CHECK(args.uid.empty(), "Пропущен ключ uid!")
    OPT_CHECK(args.src.empty(), "Пропущен ключ src!")
    OPT_CHECK(args.dst.empty(), "Пропущен ключ dst!")
    OPT_CHECK(args.first.empty(), "Пропущен ключ first!")
    OPT_CHECK(args.second.empty(), "Пропущен ключ second!")
    return true;
}
/* Вывод usage */
void usage(char* cmd) {
    std::cerr << NAME << " " << VERSION << std::endl;
    std::cerr << "Использование: " << cmd << " options... " << std::endl;
    std::cerr << "Обязательные ключи:" << std::endl;
    std::cerr << "  -i <id>  --id=<id>       Идентификатор задачи в бд" << std::endl;
    std::cerr << "  -u <uid> --uid=<uid>     Идентификатор задачи на сайте" << std::endl;
    std::cerr << "  -s <src> --src=<src>     Путь до каталога со страйпами" << std::endl;
    std::cerr << "  -d <dst> --dst=<dst>     Путь до целевого каталога" << std::endl;
    std::cerr << "  -F <txt> --first=<txt>   Текст первой ватермарки" << std::endl;
    std::cerr << "  -S <txt> --second=<txt>  Текст второй ватермарки" << std::endl;
    std::cerr << "Опциональные ключи:  " << std::endl;
    std::cerr << "  -T       --test          Протестировать окружение" << std::endl;
    std::cerr << "  -h       --help          Вывести эту справку" << std::endl;
}

/* Аргументы для препроцессора */
void parse_args(int argc, char* argv[]) {
    PARSE_OPT_START
        PARSE_OPT_ASSIGN_STR('i', args.id)
        PARSE_OPT_ASSIGN_STR('u', args.uid)
        PARSE_OPT_ASSIGN_STR('s', args.src)
        PARSE_OPT_ASSIGN_STR('d', args.dst)
```

```
PARSE_OPT_ACTION('T', {
            args.work = false;
            args.test = true;
        })
        PARSE_OPT_ASSIGN_STR('F', args.first)
        PARSE_OPT_ASSIGN_STR('S', args.second)
        PARSE_OPT_UNKNOWN
        PARSE_OPT_MISSING_ARG
        PARSE_OPT_USAGE
        PARSE_REG
    PARSE_OPT_END
}


int main(int argc, char* argv[]) {
    parse_args(argc, argv);
    if(args.test) {
        int result = test_enviroment();
        LOG(LINFO) << test_errors[result] << std::endl;
        return result;
    }
    return watermark(args.id, args.uid, args.src, args.dst, args.first, args.second);
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД КЛАССА MAIN УПРАВЛЯЮЩЕЙ ЧАСТИ

Листинг Б.4 - Main

```
#include "main.h"
#include "socket.h"
#include "launcher.h"
#include "host_collector.h"
#include "message_handler.h"
#include "../workers/Logger.h"
#include "../third-party/json.hpp"

#include <signal.h>

static void handler_chld(int signum) {
    send_done();
}

static void handler_term(int signum) {
    LOG(LINFO) << "Shutdown with signal " << signum << " requested!" << std::endl;
    global_shutdown = true;
}



/* ip port id*/
int main(int argc, char* argv[]) {

    parse_args(argc, argv);

    /* Инициализируем логгер */
    Log::Instance().print(LINFO, "0") << "Запускаем ivan_client" << std::endl;

    /* Ставим обработчик SIGTERM для локального выключения */
    struct sigaction sig = { 0 };
    sig.sa_handler = handler_term;
    sigaction(SIGTERM, &sig, NULL);
    sigaction(SIGINT, &sig, NULL);

    /* Ставим обработчик SIGCHLD для обработки отработавших воркеров */
    sig.sa_handler = handler_chld;
    sigaction(SIGCHLD, &sig, NULL);

    /* Переводим данную программу на 0 ядро */
    int cores[] = {0};
    set_core(cores, 1);

    /* Преобразуем */
    global_addr = args.addr;
    global_port = args.port;
    std::string id = args.id;

    Log::Instance().print(LINFO, id) << "ivan_client с сервером на " << global_addr << ":" <<
global_port << " запущен!" << std::endl;

    /* Собираем информацию об окружении */
    nlohmann::json j = collect();
    j["mnt"] = args.mnt;

    /* Отмечаем, что нулевое ядро занято */
```

```
struct core_info info = {
            0,          /* num  */
            0,          /* id   */
            getpid(),   /* pid  */
            0           /* exit */
    };
    global_cores_info.push_back(info);
    /* Отмечаем все ядра (кроме нулевого) как свободные */
    for(int i = 1; i <= global_cores; i++) {
        info.num = i;
        info.pid = 0;
        global_cores_info.push_back(info);
    }

    /* Запускаем сокет */
    socket_setup(global_addr, global_port);
    /* Отправляем приветственное сбщ */
    socket_send(j);

    /* Это глобальная переменная и она спрятана в обработчике сообщений */
    global_shutdown = false;
    int message;

    /* Работаем до тех пор, пока нет флага на выключение */
    /* И пока есть хоть одно выполняющееся задание */
    while(global_shutdown == false || global_pending_tasks > 0)
    {
        nlohmann::json ans;
        /* Если произошла ошибка или нам сказали завершаться, мы больше не принимаем никаких
сбщ */
        if(message != -2 && global_shutdown == false)
            message = socket_receive(ans);
        switch (message) {
            case 0:
                /* Сообщение здорового человека */
                handle_message(ans);
            case -1:
                /* Сообщение ещё не дошло, можем отвлечься */
                break;
            case -2:
                /* Внимание, произошло что-то плохое в связи с сервером! Пора закрываться */
                global_shutdown = true;
                break;
        }
    }
    /* Вне зависимости от состояния сокета пытаемся отправить сообщение о завершении */
    nlohmann::json shutdown;
    shutdown["type"] = "shutdown";
    socket_send(shutdown);

    LOG(LINFO) << "Все задания завершены, завершаюсь..." << std::endl;
    return 0;
}
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД КЛАССА LAUNCHER УПРАВЛЯЮЩЕЙ ЧАСТИ

Листинг В.5 - Launcher

```
//
// Created by Владислав on 25.04.2019.
//
#include "socket.h"
#include "launcher.h"
#include "../workers/Logger.h"

std::vector<struct core_info> global_cores_info;
int global_pending_tasks = 0;

void set_core(std::vector<int> cores) {
    cpu_set_t my_set;
    CPU_ZERO(&my_set);
    for(size_t i = 0; i < cores.size(); i++) {
        CPU_SET(cores[i], &my_set);
    }
    if(sched_setaffinity(0, sizeof(cpu_set_t), &my_set) == -1) {
        LOG_ERRNO("sched_setaffinity");
    }
}

void set_core(int* cores, int size) {
    cpu_set_t my_set;
    CPU_ZERO(&my_set);
    for(int i = 0; i < size; i++) {
        CPU_SET(cores[i], &my_set);
    }
    if(sched_setaffinity(0, sizeof(cpu_set_t), &my_set) == -1) {
        LOG_ERRNO("sched_setaffinity");
    }
}

void send_done() {
    for(int i = 1; i < global_cores_info.size(); i++) {
        /* Пропускаем, если pid равен нулю, так как никто не запущен */
        if(global_cores_info[i].pid == 0) {
            continue;
        }
        /* Спрашиваем, как там ребёнок */
        int result = waitpid(global_cores_info[i].pid, &global_cores_info[i].exit_code,
WNOHANG);
        /* Ну тут или ошибка, или ребёнок не готов */
        if(result == -1) {
            /* Процесс уже остановлен, значит, затираем инфу */
            global_cores_info[i].pid = 0;
        }else if(result == 0) {
            /* Ребёнок не готов, скипаем */
            continue;
        } else {
            /* Ребёнок готов, формируем сбщ об этом */
            if(result != global_cores_info[i].pid) {
                LOG(LWARN) << "Ждали pid = " << global_cores_info[i].pid << ", а дождались
почему-то " << result << std::endl;
            }
```

```
 nlohmann::json response;
            response["type"] = "done";
            response["exit"] = WEXITSTATUS(global_cores_info[i].exit_code);
            response["id"] = global_cores_info[i].id;
            socket_send(response);
            global_cores_info[i].pid = 0;
            global_pending_tasks--;
        }
    }
}


int start(int id, std::vector<std::string> cmd, int cores) {
    if(cores == 0) {
        return 1;
    }
    std::vector<int> cpu;
    /* Проверяем, найдётся ли у нас столько ядер для запуска */
    for(size_t i = 1; i < global_cores_info.size(); i++) {
        if(global_cores_info[i].pid == 0) {
            cpu.push_back(i);
        }
        if(cores == cpu.size()) {
            break;
        }
    }
    if(cores != cpu.size()) {
        return 2;
    }
    LOG(LINFO) << "Запускаем на ядрах: ";
    for(size_t i = 0; i < cpu.size(); i++) {
        LOG(LINFO) << std::to_string(cpu[i]);
    }
    LOG(LINFO) << std::endl;

    /* Погнали */
    int pid = fork();
    switch (pid) {
        case -1:
            LOG_ERRNO("fork");
            return 3;
        case 0: {
            /* Ребёнок */
            /* Ставимся на определённые ядра */
            set_core(cpu);

            /* Конвертируем плюсовый вектор в сишный */
            const char* args[cmd.size() + 1];
            for(size_t i = 0; i < cmd.size(); i++) {
                args[i] = cmd[i].c_str();
            }
            args[cmd.size()] = NULL;
            /* Создаём логгирующий сокет на том же адресе с портом+1 */
            int socket = socket_create(global_addr, global_port + 1);
            dup2(socket, STDOUT_FILENO);

            /* Исполняемся */
            execv(args[0], (char*const*)args);
            LOG_ERRNO("exec" + cmd[0]);
            _exit(1);
        }
    }
    for(size_t i = 0; i < cpu.size(); i++) {
```

```
global_cores_info[cpu[i]].pid = pid;
        global_cores_info[cpu[i]].id = id;
    }
    return 0;
}


std::string start_get_error(int code) {
    switch (code) {
        case 0:
            return "Всё ок";
        case 1:
            return "Нельзя запускать на нуле ядер";
        case 2:
            return "Недостаточно ядер";
        case 3:
            return "Не удалось сделать fork()";
        default:
            return "Неизвестная ошибка: " + std::to_string(code);
    }
}
```

ПРИЛОЖЕНИЕ Г

ИСХОДНЫЙ КОД МОДУЛЬНОГО ТЕСТА ДЛЯ ВОРКЕРА WATERMARK

Листинг Г.6 - WatermarkTest

```
//
// Created by Владислав on 03.05.2019.
//
#include <gtest/gtest.h>
#include "../workers/watermark/watermark.cpp"
TEST(WatermarkWorkerTest, find_images__empty_folder) {
    std::vector<std::string> list;
    find_images("./watermark_empty", list);
    ASSERT_TRUE(list.empty());
}
TEST(WatermarkWorkerTest, find_images__no_images) {
    std::vector<std::string> list;
    find_images("./watermark_no_images", list);
    ASSERT_TRUE(list.empty());
}
TEST(WatermarkWorkerTest, find_images__single_image) {
    std::vector<std::string> list;
    find_images("./watermark_single_image", list);
    ASSERT_EQ(1, list.size());
    ASSERT_STREQ("./watermark_single_image/test_VRayCam0010000.JPG", list[0].c_str());
}
TEST(WatermarkWorkerTest, find_images__single_image_mix_case) {
    std::vector<std::string> list;
    find_images("./watermark_mix_case", list);
    ASSERT_EQ(1, list.size());
    ASSERT_STREQ("./watermark_mix_case/test_VRayCam0010000.JpG", list[0].c_str());
}
TEST(WatermarkWorkerTest, find_images__subfolder_disabled_recursion) {
    std::vector<std::string> list;
    find_images("./watermark_subfolder", list, false);
    ASSERT_EQ(1, list.size());
    EXPECT_STREQ("./watermark_subfolder/test_VRayCam0010000.JPG", list[0].c_str());
}
TEST(WatermarkWorkerTest, find_images__subfolder) {
    std::vector<std::string> list;
    find_images("./watermark_subfolder", list);
    ASSERT_EQ(3, list.size());
    EXPECT_STREQ("./watermark_subfolder/1/2/test_VRayCam0010000.JPG", list[0].c_str());
    EXPECT_STREQ("./watermark_subfolder/1/test_VRayCam0010000.JPG", list[1].c_str());
    EXPECT_STREQ("./watermark_subfolder/test_VRayCam0010000.JPG", list[2].c_str());
}
TEST(WatermarkWorkerTest, watermark_file) {
    std::string watermark = create_watermark("/tmp", "ТЕСТИРОВАНИЕ", "МРЯМР");
    ASSERT_FALSE(watermark.empty());
    int exitcode = watermark_file("./watermark_single_image/test_VRayCam0010000.JPG",
"./test_tmp", watermark);
    ASSERT_EQ(DONE_SUCCESS, exitcode);


}
```