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.

Серверная 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
_____Е.А. Тушин
«___»_____2019 г.

Нормоконтролёр,
ст. преп. каф. ЭВМ
_____ С.В. Сяськов
«___»_____2019 г.

Челябинск-2019

Аннотация

Е.А. Тушин. Серверная часть автоматизированной системы управления процессом постобработки изображений и видеофайлов. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШЭКН; 2019, 34 с., 3 ил. библиогр. список – 11 наим.

В рамках данной работы было составлено два раздела выпускной квалификационной работы по созданию серверной части автоматизированной системы управления процессом постобработки изображений и видеофайлов. Была представлена актуальность такой системы в предприятии, специализирующемся на услугах рендеринга, приведен обзор предшествующей проектируемой системы. На основе обзора и запроса заказчика были составлены функциональные требования и разработан алгоритм работы серверной части от моментов получения и сбора задач до отчета о статистике загруженности.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний день примеры использования компьютерной графики можно найти практически на каждом шагу. Огромная часть визуальной рекламы, кинематограф, архитектурное планирование, эмуляция поведения физических объектов – всё это изменилось с появлением более доступной компьютерной графики. Но масштабы росли, спецэффекты в кинематографе становились сложнее, моделирование в архитектуре вырастало до размеров микрорайонов, и в связи с этим возникла проблема – 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 крупных сцен с большим количеством сложных элементов, изображение которых должно быть высокого качества, с размещением нескольких источников света и с другими усложняющими процесс рендеринга объектами, занимает значительное количество времени даже на мощных вычислительных устройствах, а рендеринг на рабочем ноутбуке дизайнера может занимать недели непрерывной работы ЭВМ. В связи с чем появился спрос на аренду вычислительных мощностей для рендеринга сцен. И, как и в любой другой нише, компании, предоставляющие такие мощности, для повышения своей конкурентоспособности начали расширять список услуг, предоставляемых клиенту помимо самих вычислительных мощностей. Что привело к появлению организаций, предоставляющих полный цикл обработки 3D-сцен. Клиент отправляет компании 3D-сцену – получает готовые отрендеренные изображения или видеофайлы.

Помимо рендеринга в список оказываемых услуг входит большое количество второстепенных задач, необходимых для превращения модели в изображение или видеофайл.

Это могут быть как задачи, которые связаны с услугами, предоставляемыми непосредственно клиенту в рамках улучшения позиций компании среди конкурентов, так и задачи «технического» характера, потребность в выполнении которых появилась в связи с усложнением полного цикла превращения 3D-модели в изображение или видеофайл. К задачам, предоставляемым клиентам, можно отнести, например, распаковку или запаковку архивов очень больших размеров или с высокой степенью сжатия. Иногда бывает быстрее и выгоднее прибегнуть к услугам компании, которая выполнит необходимые манипуляции с архивом на своих вычислительных мощностях, чем делать это на своей машине. К техническим задачам можно отнести, например, наложение водяных знаков, которое позволяет компании демонстрировать готовый продукт заказчику без риска остаться без оплаты.

В рамках данных работы некоторая часть таких услуг будет называться постобработкой. Именно для автоматизации процессов постобработки спроектирована система, серверная часть которой представлена в данной выпускной квалификационной работе.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

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

Для реализации данной цели были поставлены следующие задачи:

- проанализировать существующую систему, выделить недостатки и сильные стороны;
- выполнить анализ требований;
- выполнить проектирование серверной части системы;
- реализовать серверную часть системы;
- протестировать разработанную систему.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. ОБЗОР АНАЛОГА

Первоначальное требование к запрошенной заказчиком системе заключалось лишь в автоматизации процесса постобработки. Существовавшая на тот момент система не была в состоянии выдерживать постоянно растущую нагрузку, ввиду чего с некоторой периодичностью возникали сбои. Сбои в свою очередь требовали вмешательства специалистов для исправления.

Вскоре после начала работы над проектом, сразу же после того, как был начат анализ существовавшей системы, было выяснено, что система не 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[2]. 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.

И 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 из-за 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 же, что прошлой системе успешно удавалось. В случае с серверной частью – это передача сформированных запросов на выполнение задач, их валидация и

идентификация, в зависимости от исходов валидации и идентификации присвоение задаче определенного статуса, преобразование запросов в подходящий для выполнения задач вид, отправка преобразованного запроса на выбранного клиента, получение подтверждения принятие задачи в обработку. Все это должно выполняться автоматически, с минимальным вмешательством специалистов в ход работы, следовательно, с минимальным количеством ошибок во время выполнения задач.

2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ

2.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

После выведения требований, исходя из анализа системы-предшественника и согласования требований с заказчиком, были выделены следующие функциональные требования:

1. Серверная часть системы управления постобработкой должна принимать сообщения о задачах постобработки от стороннего бэкенд – сервиса предприятия.
2. Система должна заниматься «умным» распределением задач между клиентами (отдать задачу наиболее свободному и мощному клиенту на данный момент), тем самым решая на своей стороне проблему бутылочного горлышка.
3. Система должна подключать новых клиентов в случае получения от них приветственных сообщений и включать его в список доступных для выполнения задач клиентов.
4. Серверная часть должна вести статистику загруженности системы в целом;
5. Необходимо логирование на сетевое хранилище данных.
6. Серверная часть системы должна вести мониторинг состояния клиентов, уведомлять об изменениях в их работе.
7. Задачей серверной части является получение от экземпляров клиентов ответов со статусами выполнения задач, и сообщений с процентом выполнения процессов для некоторых типов задач.
8. У серверной части должен быть разработан API.

2.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Требованием заказчика при выборе платформы был Linux, так как система тесно взаимодействует с различными проектными решениями предприятия, косвенно связанными с постобработкой, поэтому выбора операционной системы для написания проекта не было.

Языком для написания клиентской части системы был выбран Java с использование фреймворка Spring, так как была необходимость соответствовать общему стеку технологий заказчика.

2.3. ВЫВОД

В ходе анализа проблем, связанных с предыдущим решением предприятия, были установлены и утверждены требования для будущей системы, начато формирование технического задания и выбраны платформа и язык программирования.

3. ПРОЕКТИРОВАНИЕ СИСТЕМЫ

3.1. АРХИТЕКТУРА ПРЕДПОЛАГАЕМОГО РЕШЕНИЯ

Проект представляет собой приложение, написанное на языке Java, и содержит нижеперечисленные пакеты:

- `Client`;
- `API`;
- `Distribution`;
- `Message`.

Под пакетом подразумевается часть функционала, объединенная общей семантикой и служащая для выполнения определенной цели. Пакеты представляют собой иерархически упорядоченную структуру классов в соответствии с такими концепциями, как принципы ООП и SOLID[3].

3.2. АЛГОРИТМЫ РАБОТЫ С КЛИЕНТАМИ

Для определения и последующего проектирования алгоритмов работы с клиентами требуется первоначально определить сущность клиента и его жизненный цикл.

Под клиентом мы будем подразумевать класс-модель, включающий в себя основную информацию об удаленном клиенте и инкапсулирующий основные функциональные возможности вроде отправки задачи на удаленного клиента и последующее получение результата ее выполнения.

Жизненный цикл клиента будет состоять из четырех стадий:

- Disconnected;
- Connecting;
- Connected;
- Work-ready.

Стадия “disconnected” подразумевает, что клиент где-то существует, но серверная часть системы ничего о нем не знает, а, следовательно, и не может использовать.

На стадию “connecting” клиент попадает сразу после установления

соединения с сервером. Характерные отличия от стадии “connected” заключаются в том, что серверная часть еще не получила основной информации для работы с клиентом (IP-адрес, количество доступных ядер и т.п.).

Затем следует стадия “connected”. На этой стадии серверу уже известна вся основная информация о клиенте и теоретически клиент может полноценно обрабатывать задачи. Но для полной уверенности в работоспособности клиента была добавлена валидация в виде автоматического тестирования на заранее подготовленном списке задач.

В случае успешного прохождения автоматического тестирования клиент переходит на стадию “work-ready”. Находясь на этой стадии, клиент работает уже с реальными (не тестовыми) задачами.

Модуль “`client`” можно декомпозировать по смыслу на следующие составляющие:

- `ClientConnector`;
- `ClientStorage`;
- `ClientMessageStorage`;
- `UnitFactory`.

`ClientConnector` — модуль, отвечающий за подключение клиента. Берет на себя работу по получению информации от удаленного клиента, сборку модели клиента с помощью `UnitFactory` и занесению клиента в `ClientStorage`.

`ClientStorage` — модуль, отвечающий за хранение и упорядочивание клиентов. Реализует главный метод получения наиболее подходящего клиента для каждой уникальной задачи.

`ClientMessageStorage` — модуль, отвечающий за хранение и группировку поступающих от удаленного клиента сообщений. Можно провести прямую аллегорию с почтой.

`UnitFactory` — модуль, отвечающий за создание объектов моделей клиентов из полученной информации. Представляет собой паттерн Фабрика[4].

Порядок взаимодействия перечисленных выше модулей приведен на рисунках 1 и 2.

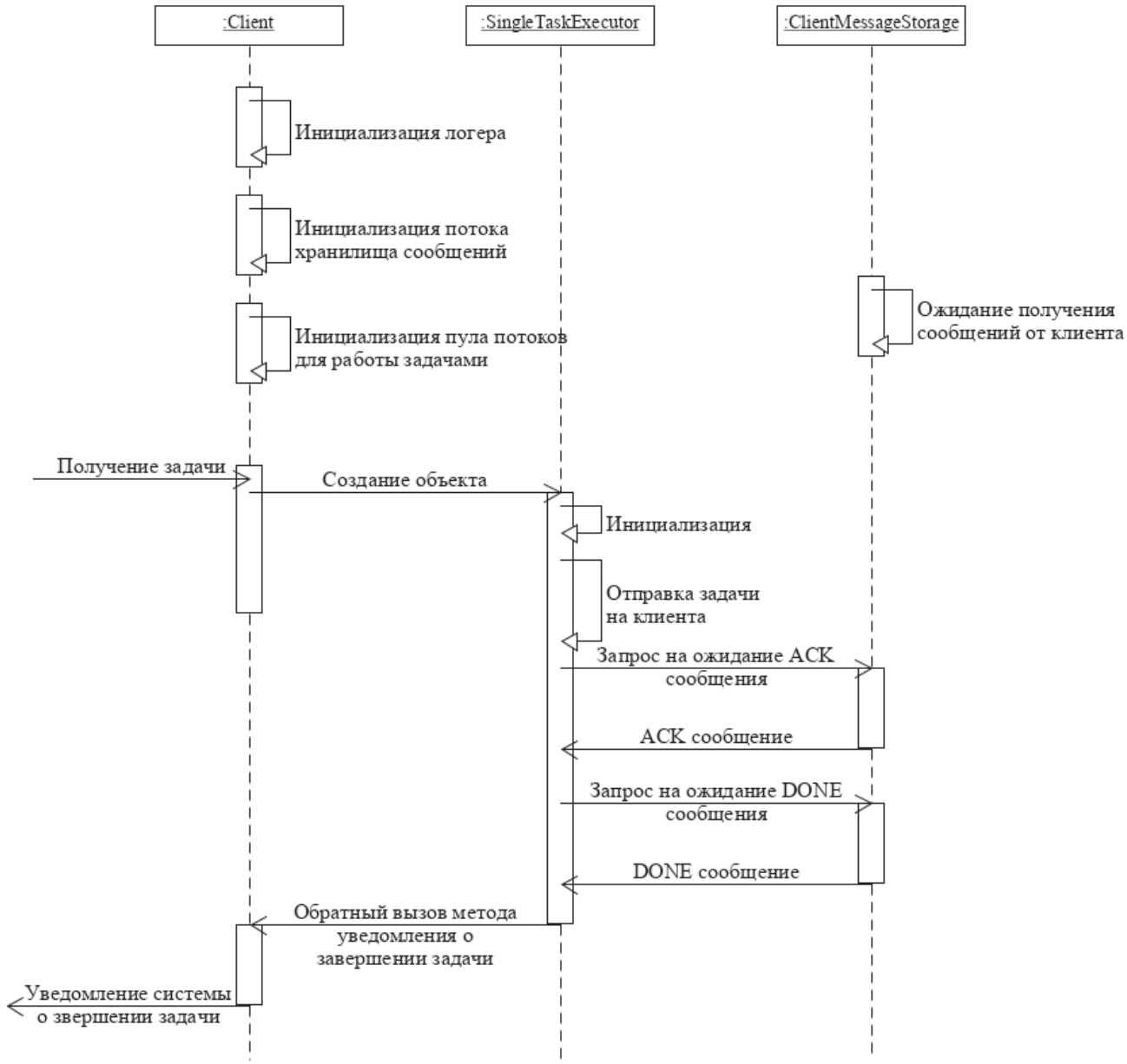

Рисунок 1 – Диаграмма взаимодействия программной сущности клиента с реальным удаленным клиентом

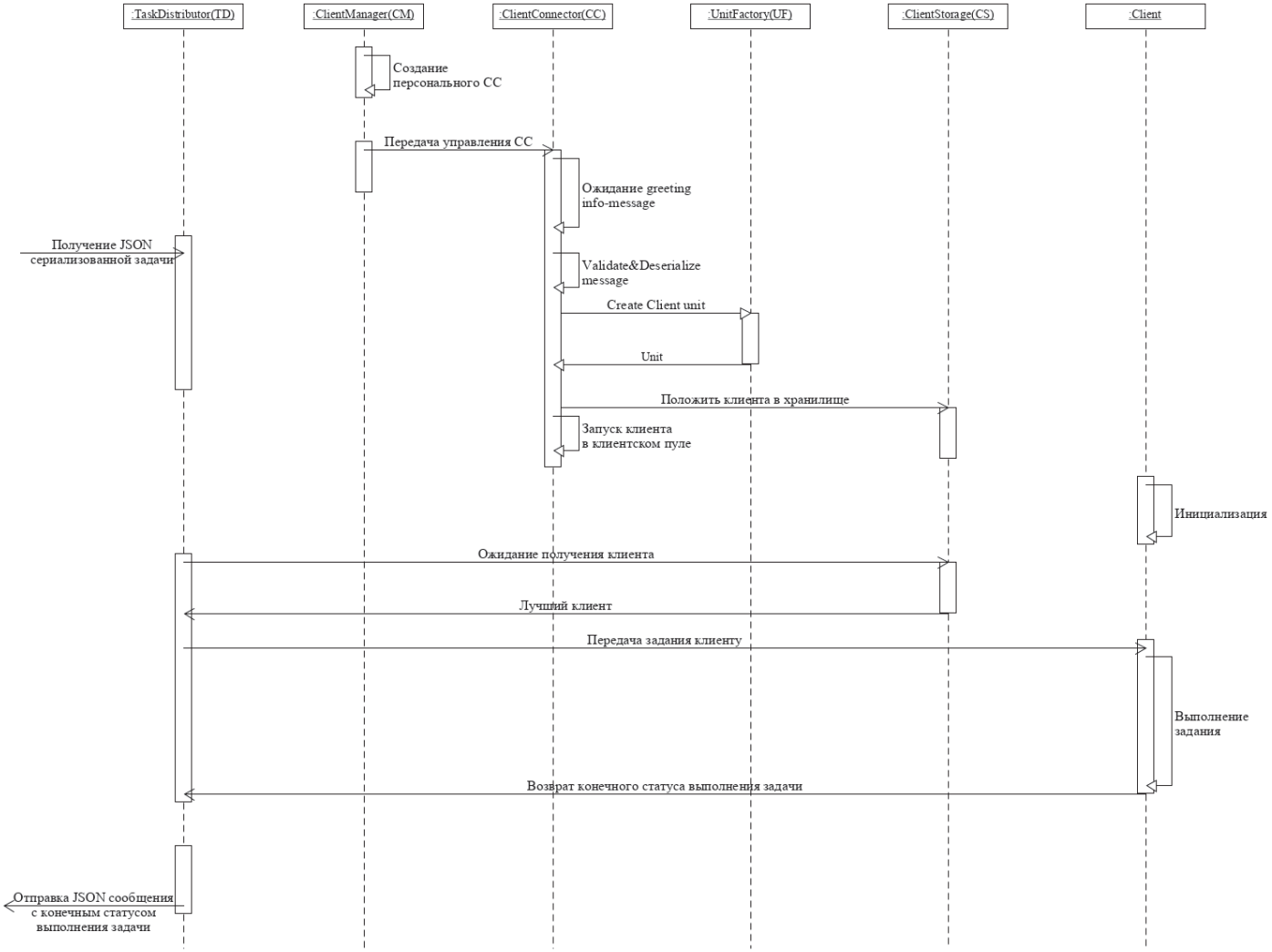

Рисунок 2 – Временная диаграмма взаимодействия модулей системы

3.3. ПРОЕКТИРОВАНИЕ REST API

Исходя из определения, API должен иметь доступ к основному функционалу сервиса и уметь корректно обрабатывать поступающие запросы[5]. Можно выделить три основные части модуля по соответствующим им основным компонентам сервиса:

- `ClientsController`;
- `LogsController`;
- `TasksController`.

Все три части дают возможность управлять компонентами сервиса в соответствии со своим названием.

`TasksController` представляет собой обычный CRUD (Create, Read, Update, Delete)-доступ к объектам задач.

`LogsController` реализует только чтение логов задач. Ввиду того, что логи априори неизменяемы извне, функционал записи логов извне не требуется — функционал создания, удаления и изменения логов не востребован.

`ClientsController` реализует чтение информации выборочно об одном клиенте либо о всех подключенных клиентах на момент выполнения запроса, а также выборочное отключение одного клиента.

Основополагающим принципом при проектировании любого REST API является идемпотентность[6]. Следование этому принципу поможет обеспечить устойчивую и предсказуемую работу модуля API.

3.4. ПРОЕКТИРОВАНИЕ МОДУЛЯ DISTRIBUTION

Самый маленький по объему модуль содержит в себе функциональность получения сообщений задач на постобработку со сторонних сервисов предприятия средствами RabbitMQ[7] с последующим распределением на подключенных клиентов. Основной задачей модуля является решение проблемы бутылочного горлышка.

3.5. ПРОЕКТИРОВАНИЕ МОДУЛЯ MESSAGE

Относительно простой с точки зрения логики модуль содержит в себе POJO модели всех типов сообщений, используемых сервисом. Сообщения можно классифицировать по области применения на две категории:

- сообщения задач;
- служебные сообщения.

Сообщения задач представляют собой семь POJO[8] моделей — по одной на каждый тип задач:

- `BigVideoTaskMessage`;
- `DirectLinkTaskMessage`;
- `GlueTaskMessage`;
- `UnzipTaskMessage`;
- `VideoTaskMessage`;
- `WatermarkTaskMessage`;
- `ZipTaskMessage`.

Каждое сообщение задачи включает в себя всю необходимую информацию для успешного выполнения поставленной задачи клиентской частью, включая заранее определенное количество ядер.

Служебные сообщения представляют из себя POJO модели сообщений, необходимые для корректного и однозначного взаимодействия серверной части сервиса с клиентской. Типы служебных сообщений:

- `ClientInfoMessageResponse`;
- `AckMessageResponse`;
- `DoneMessageResponse`;
- `TaskLogMessageResponse`.

Сообщение **`ClientInfoMessageResponse`** используется только один раз в момент идентификации клиента в момент его подключения и содержит в себе всю основную информацию о технических характеристиках клиента.

Сообщение **AckMessageResponse** принимается сервером в ответ на сообщение о постановке задачи. Содержит в себе указатель на конкретную задачу, позволяющий однозначно соотнести сообщение с предыдущим сообщением о постановке задачи, информацию о подтверждении принятия задачи в обработку и текстовое поле “reason”, которое заполняется человекочитаемым сообщением об ошибке в случае отказа принятия задачи в обработку клиентской частью.

Сообщение **DoneMessageResponse** позволяет определить, с каким статусом завершилась задача. Содержит в себе указатель на конкретную задачу и код завершения выполнения, задаваемый клиентом.

3.6. ПРОЕКТИРОВАНИЕ АЛГОРИТМА РЕШЕНИЯ ПРОБЛЕМЫ БУТЫЛОЧНОГО ГОРЛЫШКА

Краткая формулировка проблемы: при отсутствии разграничения доступных для выполнения каждой задачи мощностей возникает эффект, значительно замедляющий выполнение любых последующих задач. Эффект возникает из-за повышенного (полного) потребления ресурсов задачей, срок выполнения которой больше срока выполнения следующих за ней задач.

Предлагаемый алгоритм решения проблемы для серверной части: следует четко ограничивать доступные ресурсы для каждого определенного типа задач. Проблема частично решается балансировкой хранилища клиентов — серверная часть определяет, какой клиент наиболее подходит для решения текущей задачи по следующим критериям:

- количество свободных ядер;
- поддержка заданного типа задач.

Также должна быть предусмотрена возможность моментального подключения дополнительных клиентов.

4. РЕАЛИЗАЦИЯ

4.1. РЕАЛИЗАЦИЯ РАБОТЫ С КЛИЕНТАМИ

Обмен данными между серверной и клиентской стороной выполнен посредством BSD сокетов (сокеты Беркли)[9]. Выбор технологии обусловлен следующими факторами:

– надежность виртуальной частной сети;
– простота технологии;
– отсутствие промежуточного программного обеспечения (middleware).

Скриншот с краткой информацией о подключенных клиентах приведен на рисунке 3.

Рисунок 3 – Подключенные клиенты

Под надежностью виртуальной частной сети подразумевается надежное соединение с отсутствием потери пакетов и минимальной задержкой на их передачу в виду фактического расположения программного комплекса в рамках одного гипервизора. Также технология обеспечивает защищенность данных от внешнего прослушивания путем шифрования с использованием алгоритма RSA.

Под простотой технологии подразумевается возможность нативной работы, как со стороны клиентской части, так и со стороны серверной части.

Отсутствие же промежуточного программного обеспечения позволяет избежать временных затрат на его конфигурирование и ускорить взаимодействие сторон путем исключения лишнего звена из маршрута передачи данных и, непосредственно, времени на анализ этих данных.

4.2. РЕАЛИЗАЦИЯ REST API

Реализация REST API выполнена с использование библиотек, соответствующих спецификации JAX-RS (Java API for RESTful Web Services). Сервлет-контейнером в используемом фреймворке Spring является встроенный Tomcat компании Apache, а имплементация JAX-RS предоставляется модулем Spring MVC.

4.3. РЕАЛИЗАЦИЯ МОДУЛЯ DISTRIBUTION

Реализация `Distribution` модуля включает в себя реализацию следующего функционала:

- валидация полученных посредством RabbitMQ сообщений;
- установка обязательных параметров сообщений задач;
- присваивание внутреннего идентификатора задачи;
- отсылка задачи клиенту.

Валидация полученных сообщений проходит в два этапа: синтаксический этап и семантический этап. На синтаксическом этапе происходит проверка корректности JSON синтаксиса с использованием библиотеки `Jackson`. На семантическом этапе, после десериализации в `POJO`, происходит проверка на наличие необходимых полей, соответствующих заданному типу задачи.

После валидации происходит автоматическое задание необходимых параметров задачи (например, количество ядер для выполнения) и, при необходимости, нормализации путей.

Далее объекту присваивается уникальный номер для внутренней идентификации задачи в системе, и задача отсылается клиенту для выполнения.

Исходный код модуля `Distribution` приведен в листинге А.1 приложения А.

4.4. РЕАЛИЗАЦИЯ МОДУЛЯ MESSAGE

Реализацию модуля составляют преимущественно `POJO`. С целью уменьшения объема исходного кода и экономии времени была задействована библиотека `Lombok`, позволяющая автоматически генерировать геттеры и сеттеры.

5. ТЕСТИРОВАНИЕ

5.1. МЕТОДОЛОГИИ ТЕСТИРОВАНИЯ

Одним из главных требований к системе является бесперебойная работа на протяжении длительного промежутка времени (от одного месяца). Для обеспечения стабильной и бесперебойной работы проект необходимо подвергнуть следующим видам тестирования:

- интеграционное тестирование;
- регрессионное тестирование;
- классическое модульное тестирование.

Интеграционное тестирование заключается, прежде всего, в тестировании межмодульного взаимодействия, так как проект состоит из двух частей, а также в тестировании взаимодействия с такими модулями внешнего окружения, как RabbitMQ и другими используемыми сторонними сервисами. Интеграционное тестирование позволяет убедиться в корректности работы системы в целом.

Регрессионное тестирование служит для своевременного обнаружения новых программных ошибок, неизбежно появляющихся в процессе разработки, которых не было изначально.

Классическое модульное тестирование, также именуемое unit-тестированием, служит для тестирования отдельных мелких программных модулей одного проекта.

5.2. РЕАЛИЗАЦИЯ ТЕСТИРОВАНИЯ

Тесты написаны с использованием фреймворка тестирования (мок-библиотека) `Mockito`[10]. Фреймворк облегчает тестирование путем автоматического создания объектов-заглушек с определенным поведением.

В качестве примера разберем создание примитивной заглушки для объекта класса `SomeObject`:

```
SomeObject someObject = Mockito.mock(SomeObject.class,
new AnswerStrategy());
```

Мы явно задали заранее подготовленную стратегию ответов для объекта `someObject`. Стоит заметить, поведение реального объекта от поведения объекта-заглушки может отличаться.

5.3. АВТОМАТИЧЕСКОЕ ТЕСТИРОВАНИЕ

Тестирование происходит автоматически на этапе развертывания проекта в рабочей среде средствами стороннего ПО TeamCity от компании JetBrains[11]. Развертка проекта на производственном окружении происходит в следующие этапы:

- получение последней версии исходного программного кода из удаленного репозитория системы управления версиями;
- сборка программного комплекса;
- автоматическое тестирование на заранее подготовленных тестах;
- загрузка собранного пакета на удаленный производственный сервер;
- перезапуск сервиса, выполненного в виде systemd службы.

6. ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы была разработана серверная 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 серверной части системы, подтвердившее корректность функционирования реализованного продукта;

– осуществлена консультация по интеграции системы в предприятии.

Проблема бутылочного горлышка была полностью решена, 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.

В настоящий момент система интегрирована на предприятие и стабильно функционирует. В отличие от системы-предшественника доступно управление через API, что было отмечено заказчиком системы, как достоинство.

Перспективы развития системы:

– написание новых контроллеров API в случае появления новых требований управления системой;

– 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. World of Computer Hardware. – https://pc-builds.com/everything-you-need-to-know-about-pc-bottlenecks/. Дата обращения: 26.04.2019.
3. Medium. – https://medium.com/webbdev/solid-4ffc018077da. Дата обращения: 19.04.2019.
4. Refactoring and Design Patterns. – https://refactoring.guru/ru/design-patterns/factory-method. Дата обращения: 26.04.2019.
5. API Словарь | MDN. – https://developer.mozilla.org/ru/docs/Словарь/API. Дата обращения: 10.05.2019.
6. REST API Tutorial. – https://restfulapi.net/idempotent-rest-apis/. Дата обращения: 17.04.2019.
7. RabbitMQ. – https://www.rabbitmq.com/. Дата обращения: 26.04.2019.
8. IBM – Российская Федерация. – https://www.ibm.com/support/knowledgecenter/ru/SSCLKU_7.5.5/org.eclipse.jst.ejb.doc.user/topics/cpojosandee5.html. Дата обращения: 10.05.2019.
9. RSDN.ru. – https://rsdn.org/article/unix/sockets.xml. Дата обращения: 22.04.2019.
10. Mockito Framework Site. – https://site.mockito.org/. Дата обращения: 22.04.2019.
11. TeamCity. – https://jetbrains.ru/products/teamcity/. Дата обращения: 22.04.2019.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД REST API

Листинг А.1 - TaskController

```
import com.fasterxml.jackson.databind.ObjectMapper;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.amqp.core.AmqpTemplate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.MediaType;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;


@RequestMapping("/api/task_controller")

@RestController

public class TaskController {

    private AmqpTemplate template;

    private MessageTypeConverter messageTypeConverter;

    private TaskDistributor taskDistributor;

    private RabbitDatabaseTaskTrackingService rabbitDatabaseTaskTrackingService;


    private ObjectMapper objectMapper;

    private Logger log;


    @Autowired

    public TaskController(AmqpTemplate template,

                          MessageTypeConverter messageTypeConverter,

                          TaskDistributor taskDistributor,

                          RabbitDatabaseTaskTrackingService
rabbitDatabaseTaskTrackingService) {

        this.template = template;
```

```
this.messageTypeConverter = messageTypeConverter;
        this.taskDistributor = taskDistributor;
        this.rabbitDatabaseTaskTrackingService = rabbitDatabaseTaskTrackingService;


        this.objectMapper = new ObjectMapper();
        this.log = LoggerFactory.getLogger(TaskController.class.getName());
    }


    @RequestMapping(value = "/repeat", params = {"id"})
    public ResponseEntity<String> repeat(@RequestParam(required = true, name = "id") Integer
id) {
        try {
            final String taskBody = rabbitDatabaseTaskTrackingService.getBody(id);
            final TaskType type = messageTypeConverter.taskTypeOf(taskBody);
            if (type == TaskType.NONE) {
                log.error("Incorrect task message. Can't determine type of this message: {}",
taskBody);
                throw new IllegalStateException("Incorrect task message. Can't determine type
of this message.");
            }


            TaskMessage taskMessage = objectMapper.readValue(taskBody, type.getTaskClass());
            taskMessage.setNotifyRam(false);


            taskDistributor.receivePreformedTask(taskMessage);


            return ResponseEntity.ok(null);
        } catch (Exception e) {
            log.error("Task create failed: {}", e.getMessage());
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());
        }
    }
```

```
    @PostMapping(value = "/create", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<String> create(@RequestBody String jsonTask) {
        try {
            final TaskType type = messageTypeConverter.taskTypeOf(jsonTask);
            if (type == TaskType.NONE) {
                log.error("Incorrect task message. Can't determine type of this message: {}",
jsonTask);
                throw new IllegalStateException("Incorrect task message. Can't determine type
of this message.");
            }

            TaskMessage taskMessage = objectMapper.readValue(jsonTask, type.getTaskClass());
            taskMessage.setNotifyRam(false);

            taskDistributor.receivePreformedTask(taskMessage);

            return ResponseEntity.ok(null);
        } catch (Exception e) {
            log.error("Task create failed: {}", e.getMessage());
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());
        }
    }

    @Deprecated
    @RequestMapping(value = "/create_rmq", params = {"json_task"})
    public ResponseEntity<String> createThroughRabbit(
            @RequestParam(required = true, name = "json_task") String jsonTask) {
        try {
            final TaskType type = messageTypeConverter.taskTypeOf(jsonTask);
            if (type == TaskType.NONE) {
                throw new IllegalStateException("Not found \"subtype\" field!");
```

```
}


            template.convertAndSend("RamToIvan", jsonTask);

            return ResponseEntity.ok(null);

        } catch (Exception e) {

            log.error("Task send failed: {}", e.getMessage());

            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());

        }

    }

}
```

Листинг А.2 – `LogsController`

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 * API контроллер, отвечающий за работу с логами.
 */
@RequestMapping("/api/logs_controller")
@RestController
public class LogsController {
    private LogFileReader logFileReader;

    @Autowired
    public LogsController(LogFileReader logFileReader) {
        this.logFileReader = logFileReader;
    }

    /**
     * Возвращает ответ в виде лога по задаче с указанным ID.
     *
     * @param id ID задачи
     * @return plaintext log или сообщение об ошибке
     */
    @RequestMapping(value = "/get_log", params = {"id"}, produces = {"text/plain;charset=UTF-
8"})
    public String getTaskLog(@RequestParam String id) {
        return logFileReader.getLog(id);
    }
}
```

Листинг А.3 – `ClientsController`

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.module.SimpleModule;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 * Отвечает за работу с клиентами
 */
@RequestMapping("/api/clients_controller")
@RestController
public class ClientsController {
    private ClientStorage clientStorage;

    private ObjectMapper objectMapper;

    @Autowired
    public ClientsController(ClientStorage clientStorage) {
        this.clientStorage = clientStorage;

        this.objectMapper = new ObjectMapper();
        SimpleModule simpleModule = new SimpleModule();
        simpleModule.addSerializer(Client.class, new ClientSerializer());
        simpleModule.addSerializer(ClientStorage.class, new ClientStorageSerializer());
        this.objectMapper.registerModule(simpleModule);
    }

    /**
     * Сериализованная в json основная информация о всех клиентах, находящихся в хранилище.
     *
     * @return json-строка
     * @see ClientStorageSerializer
     */
    @RequestMapping(value = "/clients_info")
    public String getClientsInfo() throws Exception {
        return objectMapper.writeValueAsString(clientStorage);
    }

    /**
     * Информация о клиенте по его номеру в хранилище.
     *
     * @param number номер в хранилище
     * @return Сериализованная в json основная информация
     * @see ClientSerializer
     */
    @RequestMapping(value = "/client", params = {"number"})
    public String getClientByNumber(@RequestParam Integer number) throws Exception {
        return objectMapper.writeValueAsString(clientStorage.getClient(number));
    }

    /**
     * Количество клиентов в хранилище.
     *
     * @return Количество
     * @see ClientStorage#howManyClientsInStorage()
     */
    @RequestMapping(value = "/how_many_clients")
    public Integer howManyClients() throws Exception {
        return clientStorage.howManyClientsInStorage();
    }
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД МОДУЛЯ DISTRIBUTION

Листинг Б.4 - Distribution

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.module.SimpleModule;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.EnableRabbit;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.nio.charset.StandardCharsets;

/**
 * Распределитель задач, полученных из очереди RabbitMQ.
 */
@Service
@EnableRabbit
public class TaskDistributor {
    private TasksStorage tasksStorage;
    private MessageTypeConverter messageTypeConverter;
    private TaskMessageRecovery taskMessageRecovery;
    private ClientStorage clientStorage;
    private RabbitTaskNotifier rabbitTaskNotifier;
    private RabbitDatabaseTaskTrackingService rabbitDatabaseTaskTracker;

    private ObjectMapper objectMapper;
    private Logger log;

    @Autowired
    public TaskDistributor(TasksStorage tasksStorage,
                           MessageTypeConverter messageTypeConverter,
                           TaskMessageRecovery taskMessageRecovery,
                           ClientStorage clientStorage,
                           RabbitTaskNotifier rabbitTaskNotifier,
                           RabbitDatabaseTaskTrackingService rabbitDatabaseTaskTracker) {
        this.tasksStorage = tasksStorage;
        this.messageTypeConverter = messageTypeConverter;
        this.taskMessageRecovery = taskMessageRecovery;
        this.clientStorage = clientStorage;
        this.rabbitTaskNotifier = rabbitTaskNotifier;
        this.rabbitDatabaseTaskTracker = rabbitDatabaseTaskTracker;

        SimpleModule module = new SimpleModule();
        module.addDeserializer(RabbitTaskCreateMessage.class, new
RabbitTaskCreateMessageDeserializer());

        this.objectMapper = new ObjectMapper();
        this.objectMapper.registerModule(module);

        this.log = LoggerFactory.getLogger(TaskDistributor.class.getName());
    }

    @RabbitListener(queues = "RamToIvan")
    public void receiveBytes(byte[] bytes) {
        receiveTask(new String(bytes, StandardCharsets.UTF_8));
    }
```

```
/**
     * Метод, вызываемый при получении задачи из очереди
     *
     * @param rawJsonTask задача в json виде
     */
    public void receiveTask(String rawJsonTask) {
        TaskMessage recoveredTaskMessage = new NoneTaskMessage();

        try {
            log.info("Received task: {}", rawJsonTask);

            final MessageType messageType = messageTypeConverter.typeOf(rawJsonTask);

            if (messageType != MessageType.RMQ_TASK_CREATE) {
                log.error("Incorrect rmq_task_create message. Can't determine type of this
message: {}", rawJsonTask);
                //TODO: notify RAM
                return;
            }

            RabbitTaskCreateMessage rabbitMessage = objectMapper.readValue(rawJsonTask,
RabbitTaskCreateMessage.class);
            final Integer messageId = rabbitMessage.getMessageId();

            TaskMessage taskMessage = rabbitMessage.getTaskMessage();

            recoveredTaskMessage = taskMessageRecovery.recovery(taskMessage);

            //Создаем задачу в базе данных для истории, получаем ID
            final Integer id = rabbitDatabaseTaskTracker.create(taskMessage,
TaskStatus.WAITING);

            recoveredTaskMessage.setId(id);
            recoveredTaskMessage.setNotifyRam(true);

            final String recoveredJsonSerializedTask = taskMessage.toJsonString();
            log.debug("Recovered message: {}", recoveredJsonSerializedTask);

            if (tasksStorage.hasEqual(recoveredTaskMessage)) {
                log.info("Now executing equal task. This task #{} will assigned its status",
id);
                tasksStorage.relateToEqual(messageId, recoveredTaskMessage);
                return;
            }

            tasksStorage.init(messageId, taskMessage);

            sendToClient(recoveredTaskMessage);

        } catch (IllegalStateException e) {
            log.error("ERROR: can not receive task: {}", e.getMessage());
        } catch (Exception e) {
            log.error("CRITICAL: can not receive task: {}", e.getMessage());
            e.printStackTrace();
        }
    }

    public void receivePreformedTask(TaskMessage taskMessage) throws IllegalAccessException,
InterruptedException {
        TaskMessage recoveredTaskMessage = taskMessageRecovery.recovery(taskMessage);
        log.debug("Recovered task message: {}", recoveredTaskMessage.toJsonString());
```

```
        final Integer id = rabbitDatabaseTaskTracker.create(taskMessage, TaskStatus.WAITING);
        recoveredTaskMessage.setId(id);

        sendToClient(recoveredTaskMessage);
    }

    private void sendToClient(TaskMessage taskMessage) throws InterruptedException {
        final Integer requiredCores = taskMessage.getCores();
        final TaskType type = taskMessage.getTaskType();

        log.debug("Waiting available client...");
        Client bestClient = clientStorage.waitAvailableClient(requiredCores, type);
        bestClient.receive(taskMessage);
    }

    @Deprecated
    private void notifyTaskCreateFailed(TaskMessage taskMessage) {
        try {
            log.error("Task create failed!");
            rabbitTaskNotifier.notify(taskMessage, TaskStatus.REJECTED);
        } catch (Exception e) {
            log.error(e.getMessage());
            e.printStackTrace();
        }
    }
}
```