Санкт-Петербургский государственный университет

Завадский Илья Олегович

Выпускная квалификационная работа

Реализация обработки аудиофайлов из большого массива данных

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2017 «Программная инженерия»*

Научный руководитель:
к. ф.-м. н., доц. Д. В. Луцив

Рецензент:
старший программист ООО «Центр Речевых Технологий» М. С. Тимченко

Санкт-Петербург
2021

Saint Petersburg State University

Zavadskiy Ilya

Bachelor's Thesis

Implementation of Processing Audio Files From a Large Dataset

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2017 "Software Engineering"*

Scientific supervisor:
Assoc. prof., C.Sc. Dmitry Luciv

Reviewer:
Senior Developer at "Speech Technology Center" Matvey Timchenko

Saint Petersburg
2021

Оглавление

Введение **5**

1. Постановка задачи **6**

2. Обзор **7**

2.1. Распределенная система 7

2.1.1. Микросервисная архитектура 7

2.1.2. Межпроцессное взаимодействие 9

2.2. Кэширование 11

2.2.1. Политика записи 11

2.2.2. Политика промаха записи 12

2.2.3. Согласованность кэша 13

2.3. Кластеризация 14

2.4. Высокопроизводительные вычисления 15

2.5. Используемые технологии 16

2.5.1. Java 16

2.5.2. Spring Framework 16

2.5.3. PostgreSQL 17

2.5.4. Apache Ignite 18

3. Архитектура системы **21**

3.1. Функциональные требования к системе 21

3.2. Модули системы 21

3.3. Описание API 23

3.3.1. Схема обработки задания 23

3.3.2. Схема обработки команд 24

3.3.3. Схема обработки запросов 24

4. Реализация системы **25**

4.1. Описание API заданий 25

4.1.1. Описание запроса на создание задания 25

4.1.2. Описание ответа о выполненном задании 27

4.2. Описание API команд 27
4.3. Описание API запросов 29
4.4. Модуль взаимодействия с клиентом 30
4.5. Модуль обработки аудиофайлов 30
4.5.1. Задания voice модуля 30
4.5.2. Команды voice модуля 32
4.5.3. Запросы voice модуля 33
4.5.4. Работа с кэшом 34
4.5.4.1. Горячие списки 34
4.5.4.2. База идентификации 35

5. Тестирование **36**

Заключение **38**

Список литературы **39**

Введение

Когда-то люди писали программы, которые выполнялись на одной машине и к которым также осуществлялся доступ с этой машины. В настоящее время большинство приложений являются распределенными системами, которые работают на нескольких машинах и доступны множеству пользователей со всего мира.

Распределенная система — это совокупность автономных вычислительных элементов, которая представляется своим пользователям как единая связная система. На данный момент почти каждый разработчик является разработчиком или потребителем (или и тем и другим) распределенных систем [14].

С развитием технологий количество пользователей, количество данных, а также требования по скорости работы приложений растут. Для удовлетворения данных условий система должна быть масштабируемой. Следовательно, любое приложение, которое сейчас разрабатывается, должно создаваться с расчетом на масштабирование в ответ на увеличивающийся спрос. Эти ограничения и требования означают, что почти каждое разрабатываемое приложение, будь то мобильная клиентская программа или сервис обработки платежей, должно быть распределенной системой [8].

Таким образом, актуальной является задача создания системы распределенной обработки аудиофайлов, позволяющей создавать модели и сравнивать их между собой.

1. Постановка задачи

Целью данной работы является разработка распределенной системы для обработки аудиофайлов. Для достижения поставленной цели были сформулированы следующие задачи.

- Исследовать подходы и решения по тематике работы:
  - организация микросервисного ПО.
  - кэширование.
  - высокопроизводительные вычисления.
  - популярные инструменты и библиотеки.
- Спроектировать и разработать распределенную систему, позволяющую
  - построить голосовую модель по аудиофайлу.
  - сравнивать уже построенными голосовыми моделями.
  - фильтровать результаты выполнения заданий по сравнению моделей.
- Разработать сервис взаимодействия системы с внешними клиентами, позволяющий
  - отправлять запросы на обработку голосовой модели.
  - получать информацию об уже построенных моделях.
  - получать информацию о кластере.
- Провести тестирование реализованной системы.

2. Обзор

2.1. Распределенная система

Распределенная система — это совокупность автономных вычислительных элементов, которая представляется своим пользователям как единая связная система. Тремя важными характеристиками распределенных систем являются параллелизм компонентов, отсутствие глобальных часов и независимый отказ компонентов [14].

Компьютерная программа, которая выполняется в распределенной системе, называется распределенной программой (а распределенное программирование - это процесс написания таких программ).

Распределенные вычисления также относятся к использованию распределенных систем для решения вычислительных задач. В распределенных вычислениях проблема делится на множество задач, каждая из которых решается одним или несколькими компьютерами, которые взаимодействуют друг с другом посредством передачи сообщений [2].

Важным классом распределенных систем является тот, который используется для задач высокопроизводительных вычислений (англ. high-performance computing tasks) [14].

2.1.1. Микросервисная архитектура

Микросервисная архитектура — это стиль проектирования, который разбивает приложение на отдельные сервисы с разными функциями. В свою очередь сервис — это мини-приложение, реализующее узкоспециализированные функции. Заметим, что, несмотря на наличие приставки "микро", в определении микросервисной архитектуры размер не упоминается. Главное, чтобы каждый сервис имел четкий перечень связанных между собой обязанностей. Позже мы поговорим о том, что это означает [8].

Некоторые критики микросервисов утверждают, что в этом подходе нет ничего нового и что это является разновидностью сервис-ориентированной архитектуры (англ. Service-Oriented Architecture, SOA).

И сервис-ориентированная архитектура, и микросервисная архитектура — это стили проектирования, которые структурируют систему как набор сервисов. Но при более подробном рассмотрении можно обнаружить различия в межсервисном взаимодействии, в обращении с данными, а также в размере сервисов.

Как и любой стиль проектирования, микросервисная архитектура имеет преимущества и недостатки.

Микросервисная архитектура имеет следующие преимущества:

- Непрерывная доставка и развертывание крупных, сложных приложений;
- Небольшие сервисы, которые просты в обслуживании;
- Независимое друг от друга развертывание сервисов;
- Независимое друг от друга масштабирование сервисов;
- Автономность команд разработчиков;
- Возможность экспериментировать и внедрять новые технологии;
- Изолированность неполадок.

В свою очередь микросервисная архитектура обладает следующими недостатками и проблемами:

- Сложность подбора подходящего набор сервисов;
- Затруднение разработки, тестирования и развертывания из-за сложности распределенных систем;
- Тщательная координации при развертывании функций, охватывающих несколько сервисов;
- Решение о том, когда следует переходить на микросервисную архитектуру, является нетривиальным.

2.1.2. Межпроцессное взаимодействие

Существует множество разных механизмов межпроцессного взаимодействия (англ. Inter-Process communication, IPC). Сервисы могут использовать коммуникационные механизмы на основе запросов/ответов, такие как REST[1] или gRPC[2], поверх HTTP[3]. Альтернативным вариантом являются асинхронные механизмы коммуникации на основе сообщений, такие как AMQP[4] или STOMP[5]. Существует также множество других форматов сообщений — это могут быть как текстовые форматы, понятные человеку (JSON или XML), так и более эффективные двоичные, такие как Avro или Protocol Buffers.

В настоящее время наиболее популярным является разработка API в стиле RESTful. REST (Representational State Transfer — «передача состояния представления») — это механизм механизм межпроцессного взаимодействия, который зачастую задействует HTTP. Термин «REST» был введён Роем Филдингом (англ. Roy Fielding), одним из создателей протокола «HTTP», лишь в 2000 году в своей диссертации «Архитектурные стили и дизайн сетевых программных архитектур» (англ. «Architectural Styles and the Design of Network-based Software Architectures») в Калифорнийском университете. Рой Филдинг дает следующее определение этой технологии: REST предоставляет набор архитектурных ограничений, которые, если их применять как единое целое, делают акцент на масштабируемости взаимодействия между компонентами, обобщенности интерфейсов, независимом развертывании компонентов и промежуточных компонентах, чтобы снизить латентность взаимодействия, обеспечить безопасность и инкапсулировать устаревшие системы [8].

Существует шесть обязательных ограничений для построения распределенных REST-приложений. Выполнение этих ограничительных требований обязательно для REST-систем. Накладываемые ограничения определяют работу сервера в том, как он может обрабатывать и

---

[1] Representational State Transfer

[2] Remote Procedure Calls

[3] Hypertext Transfer Protocol

[4] Advanced Message Queuing Protocol

[5] Streaming Text Oriented Messaging Protocol

отвечать на запросы клиентов. Действуя в рамках этих ограничений, система приобретает такие желательные свойства как производительность, масштабируемость, простота, способность к изменениям, переносимость, отслеживаемость и надёжность [9].

Стандарт REST обладает множеством положительных качеств:

- Он простой и привычный;
- Простота тестирования;
- Встроенная поддержку стиля взаимодействия вида «запрос/ответ»;
- Протокол HTTP;
- Он не нуждается в промежуточном брокере, что упрощает архитектуру системы.

Однако использование REST имеет и недостатки.

- Он поддерживает лишь стиль взаимодействия вида «запрос/ответ»;
- Степень доступности снижена;
- Клиенты должны знать местонахождение (URL) экземпляра (-ов) сервиса;
- Извлечение нескольких ресурсов за один запрос связано с определенными трудностями;
- Иногда непросто привязать к HTTP-командам несколько операций обновления.

Несмотря на эти недостатки, REST считается стандартом де-факто для построения API.

2.2. Кэширование

Кэш в сфере вычислительной обработки данных — это высокоскоростной уровень хранения, на котором требуемый набор данных, как правило, временного характера. Доступ к данным на этом уровне осуществляется значительно быстрее, чем к основному месту их хранения. С помощью кэширования становится возможным эффективное повторное использование ранее полученных или вычисленных данных [1].

Данные в кэше обычно хранятся на устройстве с быстрым доступом, таком как ОЗУ (оперативное запоминающее устройство), и могут использоваться совместно с программными компонентами. Основная функция кэша – ускорение процесса извлечения данных. Он избавляет от необходимости обращаться к менее скоростному базовому уровню хранения.

Чтобы быть рентабельным и обеспечивать эффективное использование данных, кэши должны быть относительно небольшими. Тем не менее кэши зарекомендовали себя во многих областях вычислений, потому что типичные компьютерные приложения получают доступ к данным с высокой степенью локальности ссылок (англ. locality of reference). Такие шаблоны доступа демонстрируют временную локальность (англ. temporal locality), где запрашиваются данные, которые уже были запрошены недавно, и пространственную локальность (англ. spatial locality), где запрашиваются данные, которые физически хранятся рядом с данными, которые уже были запрошены [7].

2.2.1. Политика записи

Когда система записывает данные в кэш, она должна в какой-то момент записать эти данные в резервное хранилище. Время этой записи контролируется политикой записи (англ. write policy). Существует два основных подхода [6]:

- Сквозная запись (англ. Write-through): запись выполняется синхронно как в кэш, так и в резервное хранилище.

- Обратная запись (англ. Write-behind): изначально запись выполняется только в кэш, а запись в резервное хранилище откладывается до тех пор, пока измененное содержимое не будет заменено другим блоком кэша.

Кэш с обратной записью сложнее реализовать, поскольку он должен отслеживать, какие из его местоположений были перезаписаны, и отмечать их как грязные для последующей записи в резервное хранилище. Данные в этих местах записываются обратно в резервное хранилище только тогда, когда они удаляются из кэша, что называется отложенной записью. По этой причине промах при чтении в кэше с обратной записью часто требует двух обращений к памяти для обслуживания: один для записи замененных данных из кэша обратно в хранилище, а затем один. для получения необходимых данных.

2.2.2. Политика промаха записи

Поскольку при операциях записи запрашивающей стороне данные не возвращаются, необходимо принять решение при промахах записи (англ. write misses), будут ли данные загружаться в кэш или нет. Это определяется этими двумя подходами:

- Распределение с записью (англ. Write allocate): данные в месте пропущенной записи загружаются в кэш, после чего выполняется операция записи-попадания (англ. write-hit operation). При этом подходе промахи при записи аналогичны промахам при чтении.

- Распределение без записи (англ. No-write allocate): данные в месте пропущенной записи не загружаются в кэш, а записываются непосредственно в резервное хранилище. При этом подходе данные загружаются в кэш только при промахах чтения.

Как политики сквозной записи, так и политики обратной записи могут использовать любую из этих политик пропуска записи, но обычно они используются следующим образом [10]:

- Кэш с обратной записью использует распределение для записи, надеясь на последующие записи (или чтения) в то же место, которое теперь кэшируется.

- Кэш со сквозной записью использует распределение без записи. Здесь последующие записи не имеют преимущества, поскольку их все равно нужно записывать непосредственно в резервное хранилище.

Сущности, отличные от кеша, могут изменять данные в резервном хранилище, и в этом случае копия в кэше может стать устаревшей. В качестве альтернативы, когда клиент обновляет данные в кэше, копии этих данных в других кэшах становятся устаревшими. Для поддержания согласованности данных необходимы протоколы связи между диспетчерами кеша, они называются протоколами согласованности.

2.2.3. Согласованность кэша

Согласованность кэша — это единообразие данных общих ресурсов, которые хранятся в нескольких локальных кэшах. Когда клиенты в системе поддерживают кеши общего ресурса памяти, могут возникать проблемы с некогерентными данными, что особенно характерно для центральных процессоров в многопроцессорной системе.

В многопроцессорной системе с распределенной памятью и с отдельной кэш-памятью для каждого процессора может существовать несколько общих данных: одна копия в основной памяти и одна в локальном кэше каждого процессора, который ее запросил. В случае, когда одна из копий данных изменяется, другие копии должны получить, а затем применить это изменение. Таким образом, согласованность кэша — это дисциплина, которая обеспечивает своевременное распространение изменений значений общих данных по всей системе [15].

Для согласованности кэша существуют следующие требования [13]:

- Распространение записи (англ. Write Propagation): изменения данных в любом кэше должны распространяться на другие копии.

- Сериализация транзакций (англ.): операции чтения или записи в одну ячейку памяти должны просматриваться всеми процессорами в одном и том же порядке.

2.3. Кластеризация

Кластер — это группа узлов, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой.

Один из первых архитекторов кластерной технологии Грегори Пфистер дал кластеру следующее определение: «Кластер — это разновидность параллельной или распределённой системы, которая:

- состоит из нескольких связанных между собой компьютеров.
- используется как единый, унифицированный компьютерный ресурс».

Обычно различают следующие основные виды кластеров:

- отказоустойчивые кластеры (англ. High-availability clusters);
- кластеры с балансировкой нагрузки (англ. Load balancing clusters);
- вычислительные кластеры (англ. High performance computing clusters);
- системы распределенных вычислений (англ. Grid computing systems).

В большинстве случаев, кластеры серверов функционируют на раздельных компьютерах. Это позволяет повышать производительность за счёт распределения нагрузки на аппаратные ресурсы и обеспечивает отказоустойчивость на аппаратном уровне.

Принцип организации кластера серверов позволяет исполнять по нескольку программных серверов на одном аппаратном, что используется при разработке и тестировании кластерных решений, а также при необходимости обеспечить доступность кластера только с учётом частых изменений конфигурации серверов — членов кластера, требующих их перезагрузки в условиях ограниченных аппаратных ресурсов.

2.4. Высокопроизводительные вычисления

Термин «высокопроизводительные вычисления» стал очень популярным в мире информационных технологий в последнее время. Под высокопроизводительными вычислениями понимается практика агрегирования вычислительной мощности таким образом, чтобы она обеспечивала гораздо более высокую производительность для решения больших задач в науке, технике или бизнесе.

Высокопроизводительные вычисления используются не только для моделирования сложных физических явлений, таких как погода или астрономические вычисления, но и для улучшения приложений, снижения производственных затрат и сокращения времени разработки. Помимо этого, с увеличением способности собирать большие данные увеличивается и потребность в анализе уже собранных данных [5].

Для увеличения производительности рабочей станции в 2-3 мы можем объединить несколько компьютеров в одну сетку. Однако, если нам нужно достичь производительности более чем в 10-20 раз, нам нужно найти другую парадигму или решение: вычисления в памяти.

Вычисления в памяти (In-memory computing) — это технология обработки данных, хранящихся в базе данных в оперативной памяти [11].

Но почему же это так популярно? Потому что в последнее время цены на память сильно упали. Вычисления в памяти теперь можно использовать для ускорения обработки больших объемов данных. Наилучшие варианты использования вычислений в памяти следующие:

- Обработка ACID транзакций большого объема;
- CaaS [6];
- Кэширование базы данных;
- Комплексная обработка событий для IoT [7]-проектов;
- Аналитика в реальном времени.

[6] Cache as a Service
[7] Internet of Things

2.5. Используемые технологии

2.5.1. Java

Наиболее популярными языками программирования для разработки слоя бизнес-логики современных приложений являются языки C++, C#, Java, Kotlin, Python, JavaScript и PHP [12].

Для реализации системы был выбран язык программирования Java по следующим причинам:

- Кроссплатформенность: кроссплатформенность позволяет не задумываться об аппаратной частью и архитектурой операционной системы;
- Статическая типизация: статическая типизация позволяет отлавливать значительное количество ошибок на этапе компиляции;
- Баланс между скоростью разработки и производительностью;
- Популярность: Во всех рейтингах самых популярных языков программирования Java стабильно занимает место в ТОП-3, что говорит о том, что у языка огромная активная база сторонников, большое количество руководств, а также о том, что он развивается;
- Опыт работы: последние несколько лет я программирую на Java и могу назвать его моим основным языком, на котором веду разработку.

2.5.2. Spring Framework

Для реализации REST API были рассмотрены следующие фреймворки:

- Micronaut;
- Quarkus;
- Spring Framework.

Все перечисленные инструменты позволяют легко создать работающую версию приложения за короткое время, однако у каждого из них есть ряд недостатков. В результате был выбран фреймворк Spring, который является де-факто средой разработки для создания приложений на основе Java, следовательно у него огромное сообщество разработчиков, готовых ответить на все возникающие вопросы. Spring имеет подробную документацию с примерами, что встречается не у всех фреймворков из этого списка. Помимо этого Spring позволяет с легкостью интегрироваться с другими инструментами, необходимыми при разработке.

Были использованы следующие компоненты фреймворка Spring:

- Spring Boot;
- Spring MVC;
- Spring Data;
- Spring Test.

Центральной частью Spring является контейнер Inversion of Control, который предоставляет средства конфигурирования и управления объектами Java с помощью рефлексии. Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и конфигурирование объектов путем связывания их между собой.

Объекты, создаваемые контейнером, также называются управляемыми объектами (beans). Обычно, конфигурирование контейнера, осуществляется путём внедрения аннотаций (начиная с 5 версии J2SE), но также, есть возможность, по старинке, загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.

2.5.3. PostgreSQL

Для хранения данных о задания и их статусе обработки в системе я решил использовать СУБД PostgreSQL по следующим причинам:

- Открытое ПО соответствующее стандарту SQL: PostgreSQL — бесплатное ПО с открытым исходным кодом.

- Большое сообщество: существует довольно большое сообщество в котором вы запросто найдёте ответы на свои вопросы.

- Большое количество дополнений: несмотря на огромное количество встроенных функций, существует очень много дополнений, позволяющих разрабатывать данные для этой СУБД и управлять ими.

- Расширения: существует возможность расширения функционала за счет сохранения своих процедур.

- Объектность: PostrgreSQL это не только реляционная СУБД, но также и объектно-ориентированная с поддержкой наследования и много другого.

2.5.4. Apache Ignite

Для хранения моделей, построенных по аудио файлая, в кэше было принято решение использовать Apache Ignite. Apache Ignite — это распределенная база данных и вычислительная платформа в оперативной памяти с открытым исходным кодом. Первоначально он был разработан компанией GridGain Systems и выпущен в 2007 году. Исходный код Ignite был открыт GridGain Systems в конце 2014 года и в том же году принят в программу Apache Incubator. Проект Ignite завершился 18 сентября 2015 года [3] [5].

Ignite благодаря архитектуре долговременной памяти (durable memory architecture) позволяет хранить и обрабатывать данные и индексы как в памяти, так и на диске, подобно виртуальной памяти операционных систем.

В распределенном хранилище Apache Ignite каждому узлу принадлежит часть общих данных. Он хранит данные в виде пар ключ-значение (key-value pairs) в распределенной разделенной хэш-мапе, хранящейся в

памяти. Data Grid реализует спецификацию JCache (JSR 107), которая обеспечивает поддержку основных операций с кешем, API-интерфейсов ConcurrentMap, совместной обработки, событий и показателей и т.д.

Также Ignite предоставляет возможность совмещать вычисления с данными или данные с данными для повышения производительности — аффинной коллокацией (англ. affinity collocation). В распределенных совместно размещенных соединениях (англ. collocated joins) операция соединения выполняется локально для каждого узла и агрегируется на стороне клиента. Поскольку данные объединяются только локально, возможно, что будут возвращены только частичные результаты, потому что соответствующий ключ отсутствует на том же узле. С другой стороны, в распределенных нераспределенных соединениях (non-collocated joins) каждый узел будет отправлять широковещательные запросы другим узлам в кластере для извлечения недостающих данных. Это более дорогая операция соединения, поскольку она включает в себя дополнительный широковещательный обмен сообщениями и перемещение данных.

У Ignite есть два типа узлов кластера: серверные и клиентские.

- Серверные узлы — это узлы, которые содержат данные, участвуют в кэшировании, вычислениях, потоковой передаче и которые могут быть частью задач Map-Reduce.

- Клиентские узлы — это узлы, которые обеспечивают возможность удаленного подключения к серверам для размещения элементов в кэш или получения элементов из кэша. Они также могут хранить части данных (рядом с кэшем), которые представляют собой меньший локальный кэш, в котором хранятся самые последние и наиболее часто используемые данные.

Клиенты могут отправлять запросы на чтение или запись на любой узел кластера [5].

Функции кластеризации Ignite [4]:

- Обнаружение узла: способность обнаруживать узлы с помощью обнаружения Multicast IP или Static IP.

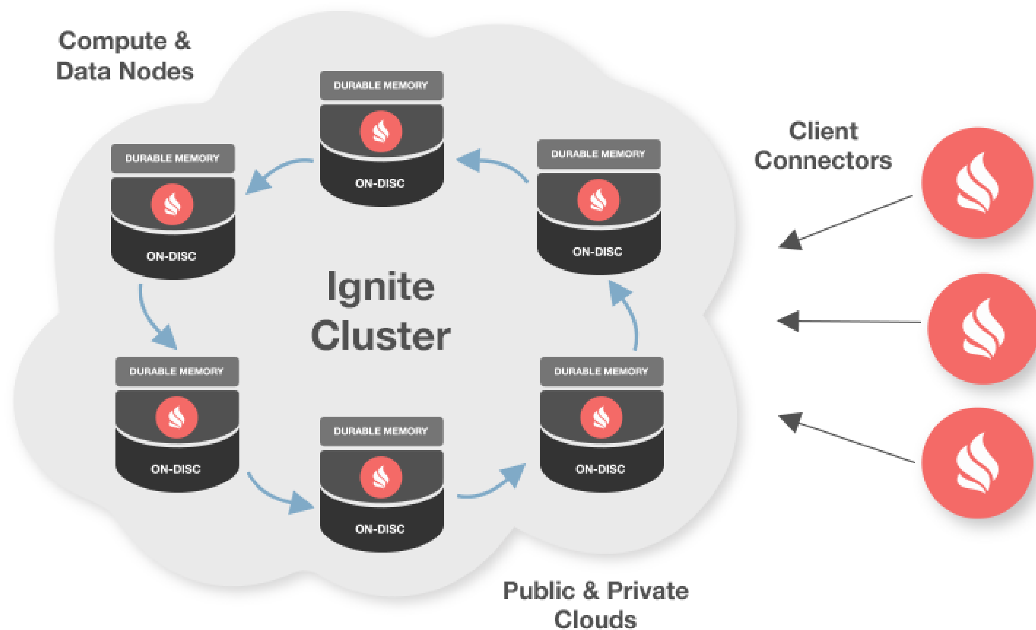

Рис. 1: Ignite кластер[8]

- Варианты развертывания кластера: возможность развертывания кластера локально или в облаке, на физических серверах или виртуальных средах.
- Группировка кластеров: возможность создавать логические группы узлов в кластере, что дает возможность назначать определенные задания или задачи только подмножеству узлов.
- Выборы лидера: возможность выбирать самые старые или самые молодые узлы в кластере, в ситуациях, когда для определенных задач требуется узел-координатор.
- Одноранговая загрузка классов: специальный распределенный загрузчик классов (англ. ClassLoader) обеспечивает нулевое развертывание, избегая явного повторного развертывания кода на узлах каждый раз, когда он изменяется.

[8] https://ignite.apache.org/docs/latest/clustering/clustering

3. Архитектура системы

3.1. Функциональные требования к системе

Разрабатываемая система должна удовлетворять следующим функциональным требованиям:

- Масштабируемость;
- Отказоустойчивость;
- Сервис обработки аудиофайлов:
  - Построение голосовой модели по аудиофайлу;
  - Сравнение между уже построенных голосовых моделей;
  - Фильтрация результатов сравнения, исходя из пороговых значений.
- Сервис взаимодействия системы с внешними клиентами:
  - Получение и передача запросов на обработку голосовой модели в сервис обработки;
  - Обработка и выполнение запросов на получение информации об уже построенных моделях;
  - Обработка и выполнение запросов на получение информации о кластере.

3.2. Модули системы

Модули, которые содержатся в проекте можно разделить на следующие группы:

- Вспомогательные модули:
  - common: базовый модуль, содержащий интерфейсы, базовые реализации сервисов и вспомогательные утилиты;

– storage: модуль взаимодействия с различными хранилищами;

– transport-core: базовый модуль передачи данных;

– transport-http: модуль передачи данных по HTTP;

– ts-db: модуль, содержащий сущности БД;

– voice-engine: модуль взаимодействия с VG SDK.

- Модули взаимодействия с клиентом:

  – rest: модуль взаимодействия с внешним клиентом посредством REST запросов.

- Модули обработки:

  – voice: модуль обработки аудиофайлов.

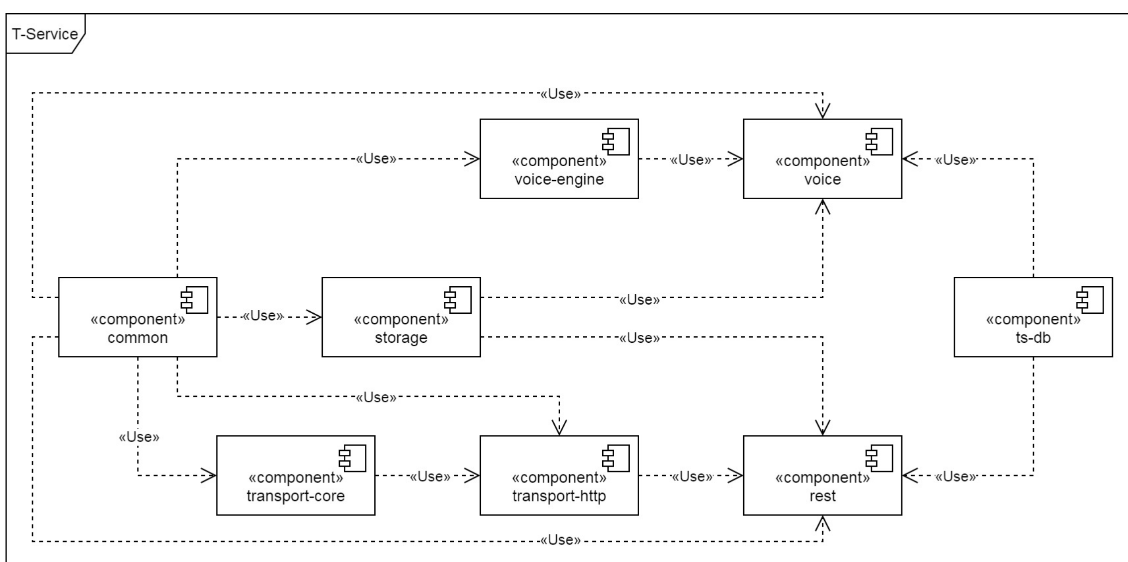

Рис. 2: Схема зависимостей модулей

Вспомогательные модули созданы для вынесения части логики из основных модулей. К данным модулям можно отнести следующие модули: common, storage, transport-core, transport-http, ts-db, voice-engine.

К модулям, отвечающим за общение, относится на данный момент только модуль rest, через который клиент может делать запросы в систему на создание выполнение задания, команды или запроса.

Модули обработки представлены только модулем voice, который отвечает за все задания, команды и запросы, направленные на создание моделей, сравнение уже построенных моделей, а также добавление моделей в горячие списки или базу идентификации. Об этом подробнее будет рассказано в разделе, посвященном этому модулю.

3.3. Описание API

API системы имеет три базовых конечных точки (англ. endpoint):

- `/api/v1/job/`, которая обрабатывает POST запросы на выполнение заданий. Задание (англ. job) — это задача, которая могу занять достаточно большое количество времени: построение модели, сравнение моделей между собой;

- `/api/v1/cmd/`, которая принимает POST запросы на выполнение команд. Команда (англ. command) — это задача, которая направлена на обновление, измение горячих листов, изменения базы идентификации и других задач, которые не очень сложны вычислительно, но требуют поле запроса;

- `/api/v1/query/`, которая обрабатывает GET запросы на выполнение запросов. Запрос (англ. query) — это задача, которые направлена на получение информации о системе.

3.3.1. Схема обработки задания

Каждый вычислительный модуль должен уметь работать с заданиями. Запросы JSON на создание нового задания принимают клиентские узлы.

Клиентский узел передает запрос[9] на обработку на сервис издатель[10], который валидирует запрос, сохраняет его в базу данных и возвращает клиенту id задания. Сервис доставки[11] выбирает задания

[9] JobRequest
[10] PublisherService
[11] PullerService

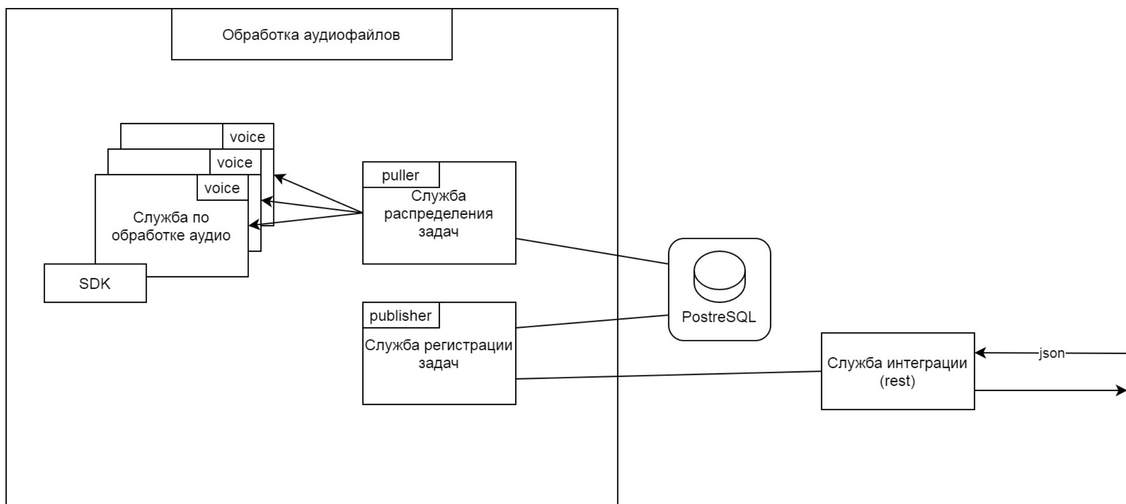

Рис. 3: Архитектура

для выполнения из репозитория, перекладывает задания в распределенную блокирующую очередь. сервис обработки[12] выполняет задания, которые он берет из очереди заданий своей группы узлов. После завершения обработки он сохраняет результат в базу данных, из которой клиент получает

3.3.2. Схема обработки команд

Клиентский узел передает запрос на выполнение на сервис, который отвечает за выполнение команд[13], который валидирует запрос, выполняет его и возвращает клиенту результат выполнения.

3.3.3. Схема обработки запросов

Клиентский узел переводит строку запроса в формат JSON и отправляет на выполнение на сервис, который отвечает за выполнение запросов[14], который валидирует запрос, выполняет его и возвращает клиенту результат выполнения.

[12]ProcessService
[13]CommandService
[14]QueryService

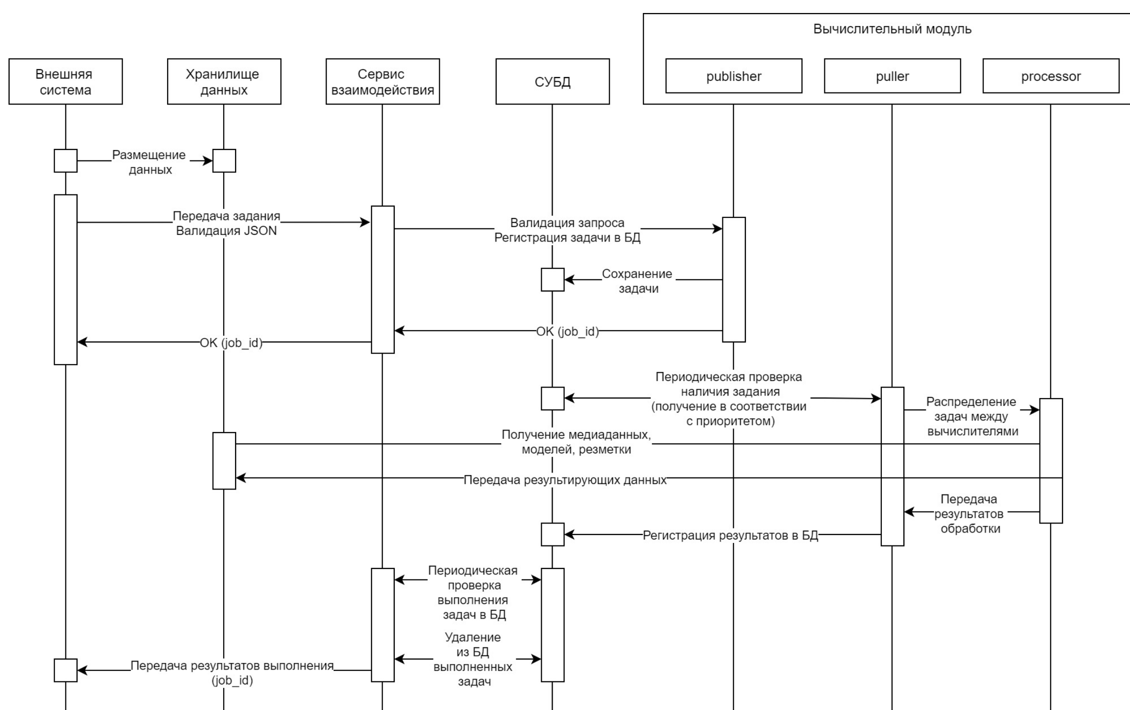

Рис. 4: Схема обработки задания

4. Реализация системы

Данный раздел подробнее рассказывает о модулях проекта, которые являются можно считать важными для системы.

4.1. Описание API заданий

4.1.1. Описание запроса на создание задания

Запросы имеют формат JSON (спецификация RFC 8259).

Дублирование имен полей одного объекта возможно, но не желательно. В таких случаях будет использовано последнее значение.

Запрос на создание задания для вычислителей состоит из 2х частей:

- header — не изменяется;
- body — варьируется в зависимости от типа задания;

Для создания задание необходимо отправить POST запрос на /api/v1/job. В случае успешного принятия задания вернется статус 201 (CREATED), а в теле ответа будет содержаться id созданного задания. Если поле

"reply_to" было заполнено, то по окончанию обработки ответ будет автоматически отправлен клиенту. И в случае подтверждения получения ответа, результат будет удален из БД. Если поле "reply_to" было пустым, то клиент сам запрашивает статус обработки, результат обработки, запрос на удаления результата обработки из БД.

Описание полей в блоке header для создания задания:

Ключ	**Обязательно**	**Описание**
service	Да	Тип сервиса, на котором будет выполнятся работа
reply_to	Да	Адрес ответа для возврата результата работы. Если ответ не требуется, то это поле остается пустым("")
priority	Да	Чем больше число, тем выше приоритет. Может принимать значения от 1 до 32767

Таблица 1: Описание полей в блоке header в ответе о выполненной работе

```
1    {
2        ''header'':{
3            ''service'': ''voice'',
4            ''reply\_to'': ''http://localhost:9090/response/42''
5        },
6        ''body'':{
7            // тело зависит от задания
8        }
9    }
```

Listing 1: Пример запроса.

4.1.2. Описание ответа о выполненном задании

Ответ о выполненной работе состоит из двух частей:

- header — Заголовок ответа, не изменяется;
- body — Результат вычислений, варьируется в зависимости от результата;

Ключ	Обязательно	Описание
id	Да	Идентификатор задания
status	Да	Успешно ли завершилась работа

Таблица 2: Описание полей в блоке header в ответе о выполненной работе

```
1  {
2      ''header'':{
3          ''id'': ''42'',
4          ''status'':''OK''
5      },
6      ''result'':{
7          // результат зависит от задания
8      }
9  }
```

Listing 2: Пример ответа о выполнении задания.

4.2. Описание API команд

Запросы имеют формат JSON (спецификация RFC 8259). Дублирование имен полей одного объекта возможно, но не желательно. В таких случаях будет использовано последнее значение.

Чтобы выполнить команду необходимо отправить POST запрос на /api/v1/cmd. В результате выполнения команды возвращается либо результат успешно выполненного команды, либо сообщение об ошибке.

Ключ	**Обязательно**	**Описание**
service	Да	Тип сервиса, на котором будет выполнятся работа
command	Да	Название команды, которая отправлена на выполенение

Таблица 3: Описание полей запроса выполнения команды

```
{
    ''service'': ''service\_name'',
    ''command'': ''command\_name'',
    // поля зависят от команды
}
```

Listing 3: Пример запроса на выполение команды.

Ключ	**Обязательно**	**Описание**
status	Да	Статус по выполнению команды
payload	Да	Результат выполнения команды

Таблица 4: Описание полей в ответе о выполненной команде

```
{
    ''status'': ''STATUS''
    ''payload'': {
        // результат зависит от задания
    }
}
```

Listing 4: Пример ответа о выполнении команды.

4.3. Описание API запросов

Запросы имеют формат URI.

Дублирование ключей возможно, но не желательно. В таких случаях будет использовано последнее значение.

Для создания запроса необходимо отправить GET запрос на /api/v1/query. В результате выполнения запроса возвращается либо результат успешно выполненного запроса, либо сообщение об ошибке.

Ключ	**Обязательно**	**Описание**
service	Да	Имя сервиса на котором будет выполнятся работа
query	Да	Название запроса

Таблица 5: Описание полей в строке запроса

```
GET /api/v1/query/{service}/{query}?{key_1}={value_1}& ...
```

Listing 5: Пример запроса.

В результате выполнения запроса возвращается ответ в формате JSON.

```
{
    // результат зависит от задания
}
```

Listing 6: Пример ответа.

4.4. Модуль взаимодействия с клиентом

Модулем взаимодействия с клиентом в системе является модуль ‘rest‘, который позволяет внешним клиентам отправлять системе запросы на обработку и получение информации о системе.

Для этого в модуле реализованы следующие REST контроллеры:

- HealthController: позволяет клиенту узнать запустился ли ‘rest‘ сервис;
- IgniteController: позволяет клиенту узнать состояние кластера, отдельных узлов, версию Ignite.
- JobController: позволяет клиенту отправить задание на обработку, либо провалидировать запрос;
- VipJobController: позволяет клиенту отправить vip задание на выполнение;
- JobResultController: позволяет клиенту получить результат выполнения задания;
- CommandController: позволяет клиенту отправить команду на выполнение;
- QueryController: позволяет клиенту отправить запрос на выполнение;

4.5. Модуль обработки аудиофайлов

Самым большим, и помимо этого самым важным по функционалу является модуль ‘voice‘. Он отвечает за все задания, команды, запросы, которые отвечают за работу с моделями, построенными по аудиофайлу или аудиофайлам.

4.5.1. Задания voice модуля

Модуль ‘voice‘ содержит следующие задания:

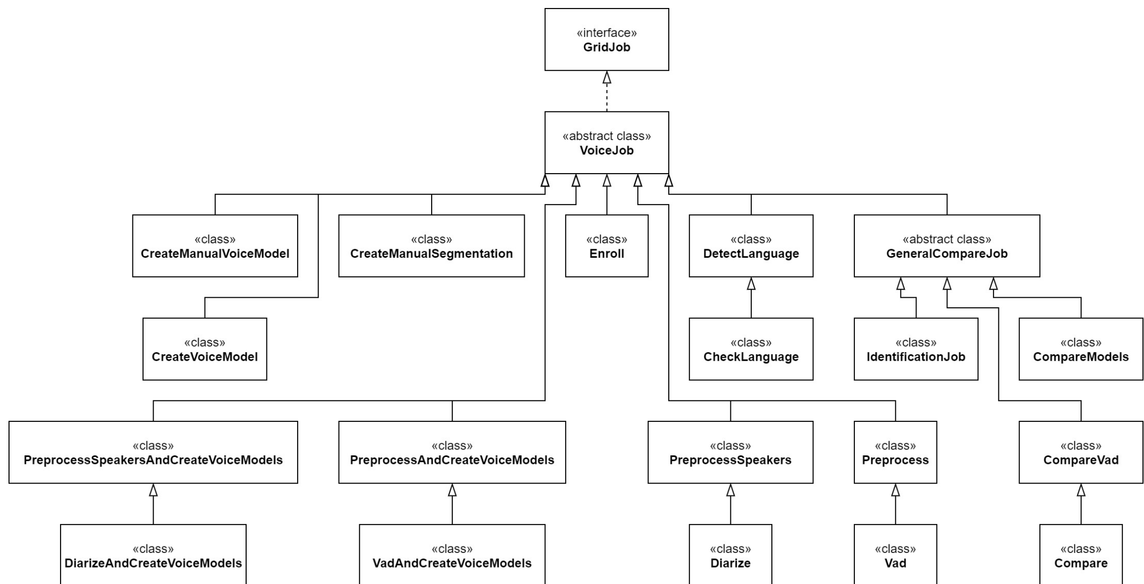

Рис. 5: Схема зависимостей заданий модуля ‘voice‘

- `compare`: сравнение файла с разделением дикторов;
- `identification`: сравнение модели против базы идентификации;
- `compare_vad`: сравнение файла без разделения дикторов;
- `compare_models`: сравнение голосовых моделей;
- `create_voice_model`: построение модели диктора;
- `manual_create_voice_model`: построение модели диктора из ручных сегментаций;
- `create_manual_segmentation`: построение сегментации диктора по ручной разметке диктора;
- `preprocess`: предобработка файла;
- `preprocess_and_create_voice_models`: предобработка файла и построение модели диктора;
- `preprocess_speakers`: предобрабработка с разделением дикторов;
- `preprocess_speakers_and_create_voice_models`: предобрабработка с разделением дикторов и построение модели диктора;

- `diarize`”: разделение дикторов без определения языка;
- `diarize_and_create_voice_models`: разделение дикторов и построение голосовых моделей без определения языка;
- `vad`: предобработка без разделения дикторов;
- `vad_and_create_voice_models`: предобработка без разделения дикторов и построение модели диктора;
- `enroll`: предобработка или разделение дикторов и построение модели диктора или дикторов;
- `detect_language`: определение языка по каждому из каналов;
- `check_language`: определение вероятности присутствия языка;

4.5.2. Команды voice модуля

Модуль ‘voice‘ содержит следующие команды:

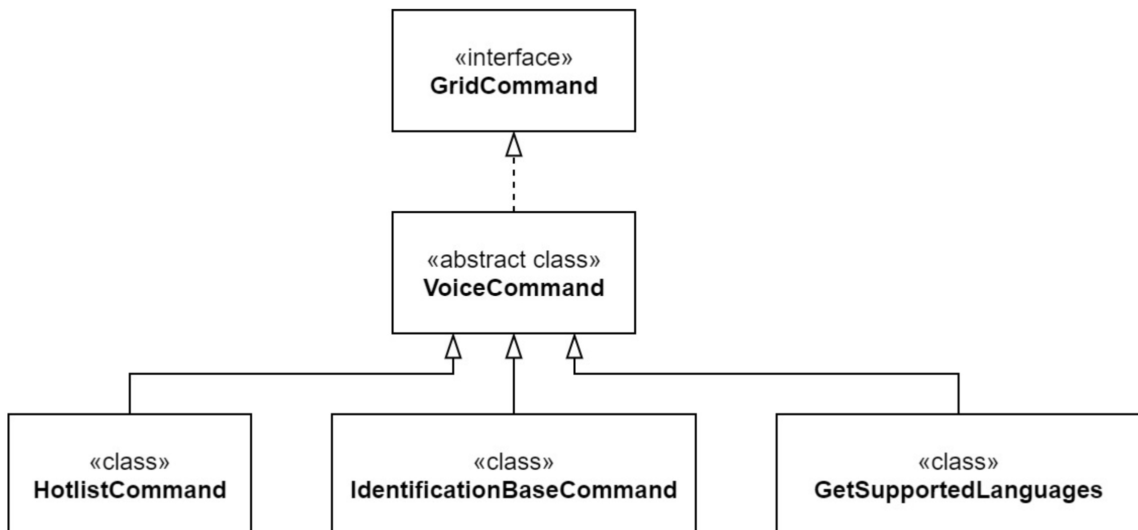

Рис. 6: Схема зависимостей команд модуля ‘voice‘

- `get_supported_languages`: возвращает список языков, которые могут быть распознаны в ходе выполнения заданий;
- `hotlist`: базовая команда для работы с горячими списками;

- `identification_base`: базовая команда для работы с базой идентификации;

4.5.3. Запросы voice модуля

Модуль ‘voice‘ содержит следующтие запросы:

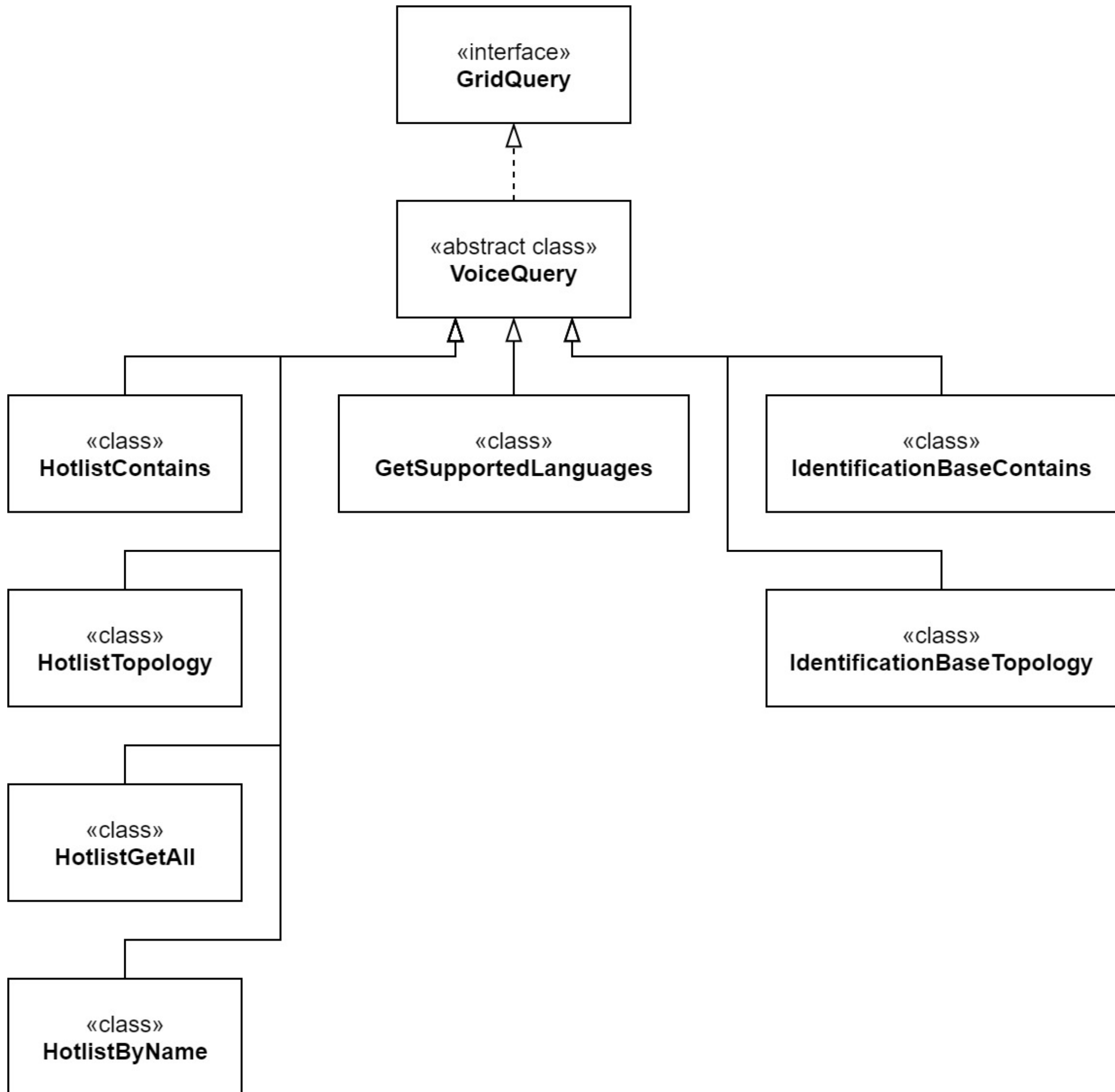

Рис. 7: Схема зависимостей запросов модуля ‘voice‘

- `supported_languages`: возвращает список языков, которые могут быть распознаны в ходе выполнения заданий;

- `hotlist`: возвращает список моделей, которые содержатся в данном горячем списке;

- `hotlist_all`: возвращает все горячие списки и модели, которые в каждом из них содержатся;
- `hotlist_contains`: проверяет содержится ли в данном горячем списке данная модель;
- `hotlist_topology`: возвращается топологию всех горячих списков;
- `identification_base_contains`: проверяет содержится ли в базе идентификации данная модель;
- `identification_base_topology`: возвращается топологию базы идентификации.

4.5.4. Работа с кэшом

Данный модуль, помимо простого построения модели, получения информации о ней и сравнения её с другими, реализует возможность хранения модели в кэше Ignite. Существует два use-case: использование горячих списков и использование базы идентификации.

4.5.4.1. Горячие списки Если сравнения с какими-либо моделями происходит достаточно часто, то разумно для экономии времени выполнения задания, хранить эти модели в кэше, таким образом экономя время на загрузке и выгрузке модели при сравнении.

Горячие списки позволяют объединить несколько моделей в один горячий список, а затем сравнивать модели не прописывая в запросе все модели, а только указывая название списка.

Реализованы следующие операции для работы с горячими списками:

- PUT: создает новый горячий список;
- ADD: добавляет в горячий список все доступные модели;
- REMOVE: удаляет из горячего списка модели, указанные в запросе;

- UPDATE: заменяет содержимое горячего списка на список моделей, переданных в теле команды;

- DELETE: удаляет горячий список.

4.5.4.2. База идентификации База идентификации представляет собой горячий список с названием ”internal_hotlist”, взаимодействие с которым происходит через API идентификации, описанное ниже:

По аналогии с горячими списками поддерживаются следующие операции:

- ADD: добавляет в базу все доступные модели;

- REMOVE: удаляет из базы модели, указанные в запросе;

- CLEAR_CACHE: удаляет все модели из базы.

5. Тестирование

Для тестирования функциональности полученного решения были написаны юнит-тесты и интеграционные тести, которые проверяют корректное выполнение задач.

Тип обработки	**Описание**	**Скорость обработки**
enroll	выделение речевой информации, определение количества дикторов и количества речевой информации для найденных дикторов, построение биометрических моделей голосов дикторов, определение пола найденных дикторов, определение языковой принадлежности	30 RT
preprocess	предобработка файла с определением языка, длительности речи, дополнительной разметкой	85 RT
detect_language	определение языка без разделения дикторов	85 RT
diarize	разделение дикторов без определения языка	110 RT
preprocess_speakers	разделение дикторов с определением языка	57 RT
compare_models	сравнение голосовых моделей	11500 сравнений в секунду на ядро

Таблица 6: Результаты тестирования системы

Также в ходе работы над данным проектом было проведено тестирование системы на скорость обработки аудиофайлов.

Тестирование проводилось на компьютере с 64-разрядным шестиядерным процессом Intel® Core™ i7-9750H с базовой частотой 2.60 ГГц, вместимостью ОЗУ[15] 32 ГБ и HDD на 2 ТБ.

Решение, реализованного в данной работе, планируется использовать в будущих продуктах компании ООО "ЦРТ"[16]

[15]Оперативного Запоминающего Устройства

[16]Центр Речевых Технологий

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Исследованы подходы и решения по тематике работы:
  - организация микросервисного ПО.
  - кэширование.
  - высокопроизводительные вычисления.
  - популярные инструменты и библиотеки.
- Спроектирована и разработана распределенная система, позволяющая
  - построить голосовую модель по аудиофайлу;
  - сравнивать между уже построенными голосовыми моделями;
  - выполнять задания, исходя из их приоритета.
- Разработан сервис взаимодействия системы с внешними клиентами, позволяющий
  - отправлять запросы на обработку голосовой модели;
  - получать информацию об уже построенных моделях;
  - получать информацию о кластере.
- Проведено тестирование реализованной системы.

В дальнейшем планируется добавить сервис взаимодействия системы с внешними клиентами через AMQ, а также добавить в систему модули обработки изображений, видеофайлов.

Список литературы

[1] AWS // Обзор кэширования. — 2021. — Режим доступа: https://aws.amazon.com/ru/caching/ (дата обращения: 10.05.2021).

[2] Andrews G.R. Foundations of Multithreaded, Parallel, and Distributed Programming. — Addison-Wesley, 1999.

[3] Apache // What is Apache Ignite? — 2020. — Access mode: https://apacheignite.readme.io/docs/what-is-ignite (online; accessed: 16.12.2020).

[4] Apache // Apache Ignite Documentation. — 2020. — Access mode: https://ignite.apache.org/docs/latest/index (online; accessed: 16.12.2020).

[5] Bhuiyan Shamim, Zheludkov Michael, Isachenko Timur. High Performance In-Memory Computing with Apache Ignite. — Lulu.com, 2017. — ISBN: 1365732355.

[6] Bottomley James // Understanding Caching. — 2004. — Access mode: https://www.linuxjournal.com/article/7105 (online; accessed: 10.05.2021).

[7] Cache hit ratio maximization in device-to-device communications overlaying cellular networks / Liang Zhong, Xueqian Zheng, Yong Liu et al. // China Communications. — 2020. — Vol. 17, no. 2. — P. 232–238.

[8] Chris Richardson. Microservices patterns : with examples in Java. — Shelter Island, 2019. — ISBN: 9781617294549 1617294543.

[9] Fielding Roy Thomas. REST: Architectural Styles and the Design of Network-based Software Architectures : Doctoral dissertation / Roy Thomas Fielding ; University of California, Irvine. — 2000. — Access mode: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[10] Hennessy John L., Patterson David A. Computer Architecture, Fifth Edition: A Quantitative Approach. — 5th edition. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011. — ISBN: 012383872X.

[11] Plattner H., Zeier A. In-Memory Data Management: Technology and Applications. SpringerLink : Bücher. — Springer Berlin Heidelberg, 2012. — ISBN: 9783642295751. — Access mode: https://books.google.ru/books?id=zzlY9glKPfkC.

[12] SkillFactory // Самые популярные языки программирования. Хабратоп 2020 года. — 2020. — Режим доступа: https://habr.com/ru/company/skillfactory/blog/531360/ (дата обращения: 21.05.2021).

[13] Solihin Yan. Fundamentals of Parallel Multicore Architecture. — 1st edition. — Chapman Hall/CRC, 2015. — ISBN: 1482211181.

[14] Tanenbaum Andrew S., Steen Maarten van. Distributed Systems: Principles and Paradigms (2nd Edition). — USA : Prentice-Hall, Inc., 2006. — ISBN: 0132392275.

[15] Thomadakis Michael. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms // JFE Technical Report. — 2011. — 03.