Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра теоретических основ информатики

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р техн.наук, доцент
_____А.В. Замятин
«_05_» _июня_ 2020 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

ПЛАНИРОВЩИК ЗАДАЧ НА ДИНАМИЧЕСКИ РАСШИРЯЕМОМ КЛАСТЕРЕ MESOS

по основной образовательной программе подготовки бакалавров
«Фундаментальная информатика и информационные технологии»
направления подготовки 02.03.02 Фундаментальная информатика и информационные технологии

Скрипник Максим Андреевич

Руководитель ВКР
доцент, канд. техн. наук
_____А.Л. Фукс
«_02_» _июня_ 2020 г.

Консультант
руководитель отдела разработки
ООО "Битворкс"
_____В.А. Михальчук
«_02_» _июня_ 2020 г.

Автор работы
студент группы № 1461
_____М.А. Скрипник

Томск – 2020

Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра теоретических основ информатики

УТВЕРЖДАЮ
Руководитель ООП
д-р ~~техн.наук, доцент~~
_____ А.В. Замятин
«16» декабря 2019 г.

ЗАДАНИЕ

по подготовке ВКР бакалавра студенту Скрипнику Максиму Андреевичу группы №1461

1. Тема ВКР «Планировщик задач на динамически расширяемом кластере Mesos»
2. Срок сдачи студентом выполненной ВКР:
а) на кафедре 05.06.2020
б) в ГЭК 11.06.2020
3. Исходные данные к работе. Цель работы: разработать систему, позволяющую запускать вычислительные задачи на кластере Mesos и динамически изменять его размер в зависимости от вычислительной нагрузки.
4. Краткое содержание работы:
    1. Проанализировать предметную область и существующие решения.
    2. Ознакомиться с инструментами для реализации и выбрать подходящие.
    3. Спроектировать систему.
    4. Реализовать систему.
    5. Создать документацию по системе.
5. Организация, по заданию которого выполняется работа:
    Национальный исследовательский Томский государственный университет
6. Дата выдачи задания «16» декабря 2019 г.

Руководитель ВКР
доцент, ТГУ, канд.физ.-мат. наук _____ А.Л. Фукс

Консультант
руководитель отдела разработки,
ООО “Битворкс” _____ В.А. Михальчук

Задание принял к исполнению 16 декабря _____

РЕФЕРАТ

Выпускная квалификационная работа, 67 страниц, 11 рисунков, 1 таблица, 35 источников.

Ключевые слова: Распределённые вычисления, кластер, масштабирование, Apache Mesos, Scala, Akka, OpenAPI, Digital Ocean.

Цель работы: Разработать систему, позволяющую запускать задачи на вычислительном кластере Mesos и автоматически увеличивать или уменьшать размер этого кластера в зависимости от потребностей запускаемых задач.

Результат работы: Разработана система, позволяющая запускать задачи на кластере Mesos и изменять размер этого кластера в зависимости от нагрузки, реализовывать различные стратегии изменения размеров кластера и реализовывать поддержку различных облачных провайдеров в качестве источников вычислительных ресурсов. Реализована стратегия для системы, которая поддерживает размер очереди задач не больше определённого значения путём добавления и удаления ресурсов при необходимости. Реализована поддержка облачного провайдера Digital Ocean для системы для изменения размеров кластера.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время распределённые вычисления пользуются большой популярностью, поскольку они предоставляют возможность обрабатывать гораздо большие объёмы данных, нежели одиночные вычислительные устройства. Нередко высоконагруженные системы могут включать в себя десятки, сотни и даже тысячи серверов, объединяясь в кластеры и образуя сложные топологии. Эти кластеры сложно не только администрировать, но и эффективно распределять доступные ресурсы кластера в зависимости от текущей нагрузки и имеющихся потребностей в работе приложений. Поэтому возникает необходимость в создании централизованных отказоустойчивых систем для управления кластером, которые полностью берут на себя функции оркестрации запускаемых в нём приложений. Таким образом, написание приложений для кластера становится намного проще, так как в них реализуется только необходимая бизнес-логика, а управлением необходимыми приложению ресурсами занимается уже другая система.

Также за последние годы сильно увеличилась доля использования виртуальных серверов, предоставляемых так называемыми облачными провайдерами. В первую очередь это связано с тем, что благодаря таким сервисам удается достичь значительного уменьшения затрат на приобретение вычислительных ресурсов на ранних этапах проекта. Аренда виртуального сервера на несколько месяцев обойдётся гораздо дешевле, чем покупка полноценного мощного сервера, средств на который на старте проекта может не быть. Экономить также помогает гибкость, заключающаяся в возможности довольно сильно кастомизировать конфигурацию арендуемых серверов: увеличивать или уменьшать объём мощностей по требованию, без дорогостоящих ручных миграций. Кроме того, тенденцию на использование таких виртуальных машин можно объяснить простотой использования:

настройка и обслуживание собственного сервера является весьма трудоемкой задачей, в то время как облачные провайдеры берут эти активности на себя.

Компания, в которой работает автор данной работы, занимается реализацией и поддержкой зарубежного коммерческого проекта, в котором также возникла необходимость запускать и отслеживать процессы на кластере, работающем с использованием мощностей виртуальных машин. Для решения этих задач, в проекте используется Apache Mesos (далее - Mesos). Mesos представляет собой дополнительный слой абстракции, инкапсулирующий детали расположения и доступа к конкретным вычислительным узлам, представляя весь кластер как набор ресурсов: центральных и графических процессоров, оперативной памяти, дискового пространства и так далее. Mesos широко используется многими коммерческими гигантами, такими как Apple, Twitter, eBay, Uber и другими [1]. Для выполнения некоторых сложных задач требуется большое количество ресурсов, поэтому в проекте иногда возникает проблема нехватки ресурсов кластера. Только после добавления администратором кластера новых ресурсов сложные задачи могут запуститься. Но после их завершения возникает ещё одна проблема - добавленные ресурсы могут не использоваться проектом довольно продолжительное время. Сам Mesos автоматически не изменяет размер кластера, поскольку не анализирует нагрузку процессов, работающих в кластере. Таким образом, в условиях переменной и трудно прогнозируемой вычислительной нагрузки может оказаться, что программам в определенный момент времени необходимо больше ресурсов, чем доступно в кластере или наоборот, программы утилизируют кластер Mesos недостаточно эффективно, то есть ресурсы простаивают. Очевидно, что долю простаиваемых ресурсов необходимо минимизировать, ведь они неизбежно ведут к финансовым издержкам, особенно при использовании облачных провайдеров. Как правило, виртуальные

машины тарифицируются поминутно, а значит каждая минута простоя это потраченные впустую деньги.

Целью данной работы является разработка системы, позволяющей запускать задачи на вычислительном кластере Mesos и автоматически увеличивать или уменьшать размер этого кластера в зависимости от потребностей запускаемых задач. Поскольку облачный провайдер можно рассматривать как условно неограниченный источник ресурсов, то именно у него система будет автоматически арендовать ресурсы. Создавая виртуальные машины только когда необходимо и удаляя их (т.е. возвращая провайдеру), когда необходимость в них пропадает, система позволяет значительно снизить расходы на вычислительные ресурсы.

1 Обзор Apache Mesos

Mesos - это отказоустойчивая масштабируемая система, позволяющая запускать процессы на кластере и осуществляющая наблюдение за этими процессами. Главными элементами Mesos являются Mesos Master и Mesos Agents. Mesos Master - это процесс, запущенный на особом, главном узле кластера, через который приложения, желающие использовать вычислительные ресурсы кластера, запускают прикладные процессы. Mesos Agent - процесс, запущенный на узле кластера, который предполагается использовать для запуска прикладных процессов. Агент настраивается на работу с определённым мастером и после прохождения регистрации в кластере Mesos, реагирует на запросы запуска процессов от мастера. Разумеется, почти всегда работают сразу несколько агентов, количеством и параметрами которых и определяется вычислительная мощность всего кластера.

Приложение, желающее запустить процессы на кластере Mesos, также регистрируется через мастер (команда Subscribe), как фреймворк-планировщик (в терминологии Mesos чаще просто фреймворк) [3]. Для запуска прикладных процессов Mesos использует механизм так называемых “офферов” (англ. offer - предложение). Суть этого механизма заключается в том, что Mesos Master сам регулярно, поочерёдно и с достаточно высокой периодичностью, “предлагает” вычислительные ресурсы кластера всем зарегистрированным фреймворкам-планировщикам. Получивший набор офферов фреймворк, должен в ответ принять решение по каждому из них, основываясь на потребностях процессов, которые желает запустить. Оффер может быть

либо отклонён (команда Decline), если ресурсы, которые он содержит, не представляют интереса для фреймворка, либо принят (команда Accept) с указанием данных, необходимых для запуска процесса: путь к исполняемому файлу, аргументы командной строки, переменные окружения и так далее. Таким образом, приложения не запрашивают ресурсы у кластера Mesos самостоятельно, а только реагируют на офферы. Это позволяет разработчикам приложений не реализовывать стратегии периодического опроса (так называемый polling) кластера на предмет текущих ресурсов, а также переносит задачу предоставления конкурентного доступа к ресурсам кластера несколькими приложениями (фреймворками) в зону ответственности самого Mesos.

Далее будем называть процессы, которые запускаются приложением на кластере Mesos, - задачами (Job). Получив от фреймворка решение принять оффер с параметрами задачи, мастер запускает соответствующий процесс на вычислительном узле, соответствующем офферу.

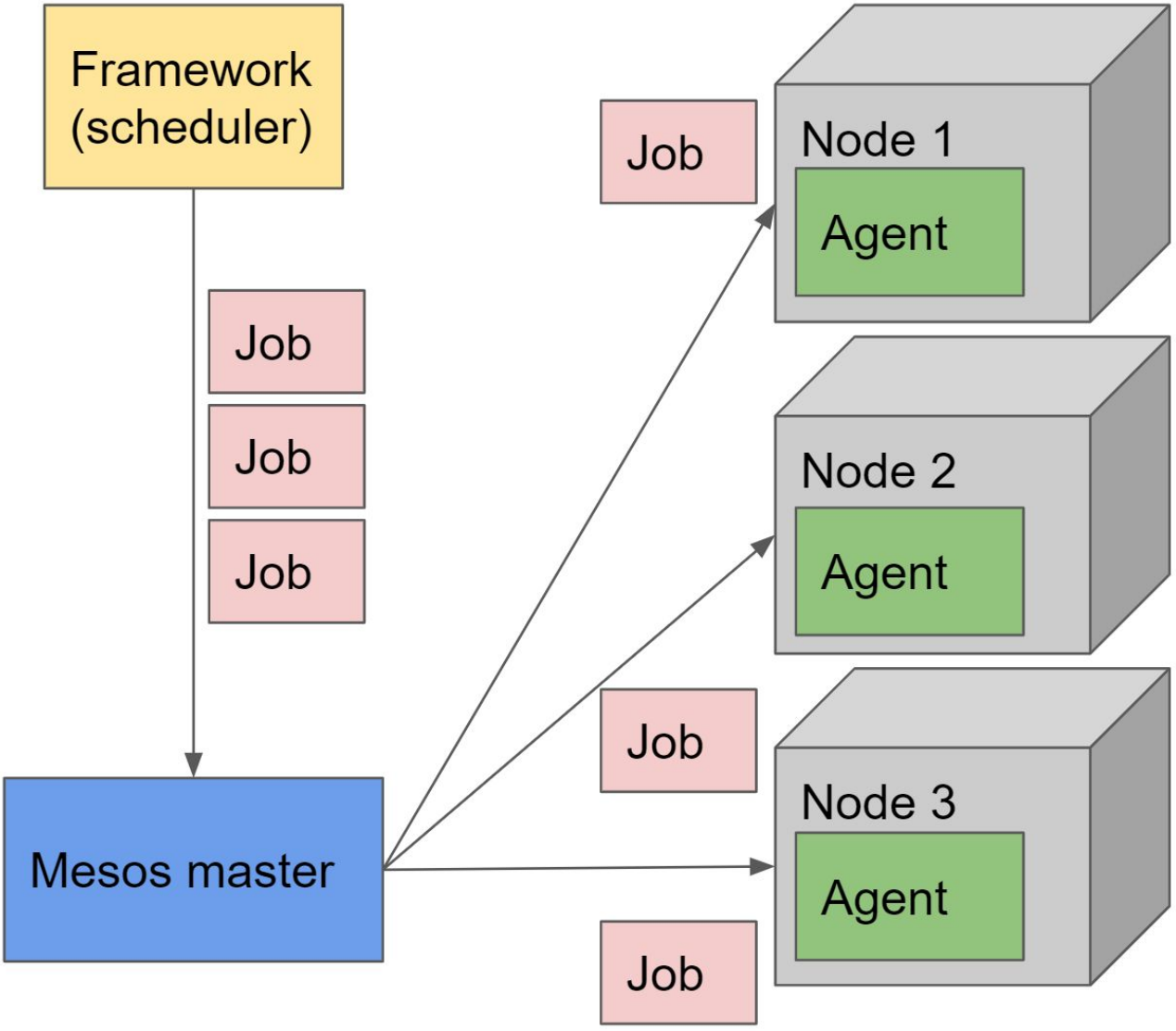


Рисунок 1 - схема взаимодействия приложения (Framework) с кластером Mesos

Помимо офферов, фреймворк также регулярно получает информацию о состоянии запущенной задачи, например, об успешном завершении или завершении с ошибкой. Получив соответствующее событие, фреймворк должен подтвердить факт его обработки (команда Acknowledge), иначе Mesos будет продолжать регулярно отправлять его фреймворку.

Функциональность Mesos’а этим не ограничивается: он предоставляет различные точки расширения и может быть интересен не только для разработки фреймворков-планировщиков. Однако это выходит за рамки данной работы, поэтому эти возможности Mesos’а мы рассматривать не будем.

2 Обзор аналогов

Описанный выше механизм взаимодействия с Mesos является относительно низкоуровневым, хоть и достаточно гибким, поэтому множество приложений (вероятно, большинство) на самом деле не работают с интерфейсом Mesos Master напрямую, а используют уже готовые фреймворки для прикладных задач, предоставляющие более высокоуровневые интерфейсы и заточенные под более специфические варианты использования. Разрабатываемая в рамках работы система также является фреймворком-планировщиком в терминологии Mesos и позволяет приложениям запускать задачи на кластере Mesos без прямого взаимодействия с Mesos Master. В связи с этим, в этой главе будут кратко описаны существующие системы, решающие похожие задачи.

Сайт проекта Mesos приводит следующую классификацию популярных фреймворков, написанных на его основе [4]:

- Инструменты DevOps
- Фреймворки для долгоживущих сервисов
- Обработчики больших данных
- Пакетные обработчики
- Хранилища данных
- Инструменты для машинного обучения

Рассмотрим подробнее каждую из этих категорий и примеры соответствующих фреймворков.

Под DevOps (англ. development and operations) подразумевают методологию процесса разработки, когда специалисты по развёртыванию и доставки приложений тесно взаимодействуют с разработчиками

соответствующих систем, а разработчики, в свою очередь, во время создания систем, принимают во внимание способы и технологии, которые используют для доставки приложений. Иногда зоны ответственности тех и других вообще объединяются в одну, и тогда разработчики сами организуют процессы поставки систем. Часто, однако, этот термин используют просто для обозначения работ, связанных с развёртыванием приложений. Соответственно, фреймворки для DevOps решают задачи, направленные на автоматизацию процессов развёртывания и доставки компонентов системы. Примером такого фреймворка является vamp [5]. Vamp, в числе прочего, предоставляет удобный интерфейс для автоматизации поставки релизов приложений и описания A/B тестирования и проведения соответствующих тестов. A/B тестирование это подход, при котором несколько версий приложения разворачиваются для конечных пользователей одновременно (версии A и B) с целью выбора более предпочтительной версии на основании анализа различных факторов, которые нельзя оценить в искусственных (тестовых) окружениях, например, отзывов пользователей. Vamp является платным фреймворком, поэтому мы опустим более детальный его разбор в рамках данной работы. Фреймворки-инструменты DevOps решают совершенно другую задачу, нежели разрабатываемая система, не подходят для запуска задач, создаваемых приложениями во время их работы, а также обычно не требуют динамического расширения кластера для выполнения своих функций.

Фреймворки для долгоживущих сервисов являются платформами для оркестрации таких сервисов, запускаемых на кластере Mesos. Они предоставляют достаточно высокоуровневые интерфейсы для запуска сервисов с указанием ресурсов, которые им необходимы, и количеством их

экземпляров для горизонтального масштабирования или достижения отказоустойчивости. Примером такого сервиса может служить сервер для онлайн-магазина, запущенный в трёх экземплярах, каждому из которых для работы необходимы 2.0 ЦПУ, 2 ГБ ОЗУ и 5 ГБ дискового пространства. Другой пример - сервер базы данных, запущенный в одном экземпляре, которому требуется 1.0 ЦПУ, 4 ГБ ОЗУ и 100 ГБ дискового пространства. Одним из таких фреймворков является Marathon, созданный компанией Mesosphere [6]. Работа с ним ведётся посредством HTTP интерфейса. Он следит за состоянием всех сервисов, созданных с его помощью, и если какой-то экземпляр завершился или перестал отвечать, он автоматически перезапустит его.

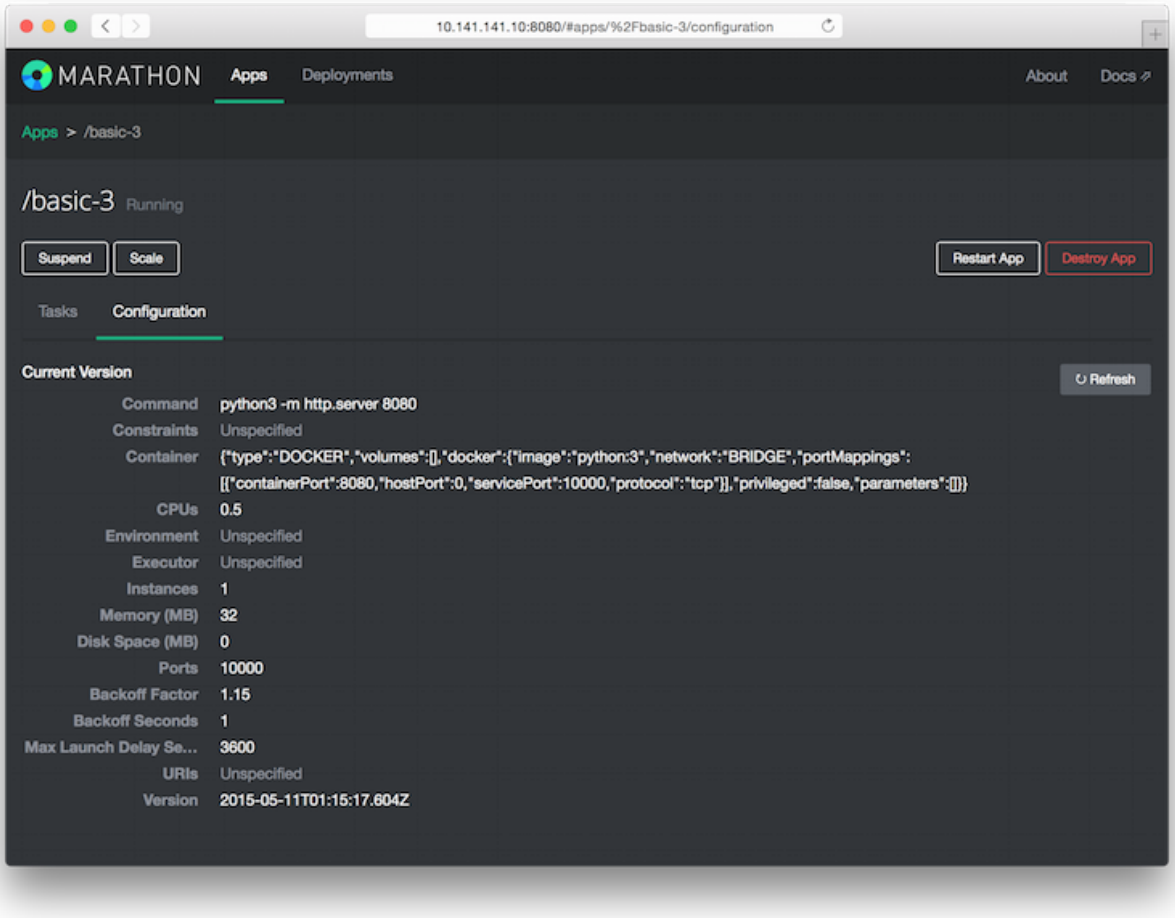


Рисунок 2 - графический интерфейс Marathon, показывающий информацию о запущенном HTTP сервере под управлением языка Python

Разрабатываемая система не подразумевает запуск задач, которые должны работать постоянно и в нескольких экземплярах в целях масштабирования, и этим она отличается от фреймворков такого типа.

Фреймворки-обработчики больших данных созданы для эффективной обработки массивов данных, которые целиком не помещаются в ОЗУ одного компьютера, но допускающие параллельную обработку. Они стараются использовать максимум доступных ресурсов кластера, чтобы минимизировать время обработки. Примером такого

фреймворка является Apache Spark, который предоставляет библиотеки с высокоуровневым интерфейсом для языков программирования Scala и Python, и среду исполнения для программ, написанных с их использованием. Эти библиотеки позволяют описывать преобразования, которые необходимо произвести с данными, не детализируя, как именно данные будут распределяться по узлам кластера. Разрабатываемая в рамках данной работы система не предназначена для работы с задачами такого типа.

Фреймворки, занимающиеся пакетной обработкой, предназначены для запуска таких задач, которые необходимо единожды выполнить в соответствии с их вычислительными требованиями, ради достижения какого-то внешнего эффекта, например, обработки набора данных и сохранения результата обработки во внешней базе данных. Примером фреймворка такого типа является Metronome, который как и Marathon, создан компанией Mesosphere [7]. Он тоже предоставляет достаточно высокоуровневый HTTP интерфейс для запуска задач с указанием ресурсов, которые им необходимы. Задачи, которые регистрируются в Metronome, будут запускаться регулярно, согласно заданному расписанию. Этим он похож на Unix утилиту CRON, что отражено на главной странице сайта проекта. Примером задачи, которую удобно запускать с помощью Metronome, является скрипт, который раз в сутки проверяет число заболевших определённым вирусом в мире, на основании открытых данных с разных источников, группирует её по стране, и обновляет информацию в БД, чтобы потом отобразить её на сайте. Фреймворки этого типа работают с тем же типом задач, что и разрабатываемая система, что позволяет рассматривать их, как самые близкие аналоги из рассмотренных в этой главе, однако на сегодняшний день не существует публичного и

широко известного фреймворка, который следит за тем, хватает ли ресурсов кластера Mesos для запуска входных задач и самостоятельно изменяет размер этого кластера при необходимости. В случае с Metronome, например, может возникнуть ситуация, что запланированный для запуска скрипт потребует ресурсов больше, чем свободно в кластере в очередной момент запуска. Тогда Metronome его просто не запустит, пока соответствующие ресурсы не освободятся или оператор вручную не расширит Mesos кластер, запустив агента на новом сервере. Разрабатываемая же система будет принимать во внимание тот факт, что задачам на протяжении какого-то времени не хватает ресурсов для запуска, и добавит в кластер соразмерное количество ресурсов.

Фреймворки-хранилища данных сильно отличаются от разрабатываемого решения, поскольку процессы, которые они запускают на кластере, имеют довольно узкое назначение - они направлены на организацию доступа к данным, распределённых по кластеру. Хранилище данных ElasticSearch [8] является примером такого фреймворка, и одна из ключевых его функциональностей - поддержка эффективного по времени полнотекстового поиска. Разрабатываемая система не направлена на предоставление структур и алгоритмов хранения данных.

Фреймворки-инструменты машинного обучения также решают специфические задачи, связанные с алгоритмами и структурами искусственного интеллекта, поэтому оптимизированы на эффективное использование ресурсов для организации соответствующих вычислений. Одним из таких фреймворков является TFMesos [9], который предоставляет высокоуровневый интерфейс для запуска задач, работающих под управлением платформы TensorFlow [10], и использования графических процессоров Nvidia [11] для ускорения

обработки данных. Разрабатываемая система не направлена на решение задач машинного обучения.

Таким образом, несмотря на многообразие фреймворков, написанных на основе Mesos, только один их тип - Mesos фреймворки для пакетной обработки, решает часть задач, на которые нацелена разрабатываемая система. Несмотря на это, ни один из рассмотренных фреймворков, вне зависимости от категории, не занимается изменением размеров кластера Mesos, на котором запускаются вычислительные задачи. Именно это является ключевой особенностью разрабатываемого решения и делает его предпочтительнее существующих решений для ряда приложений.

3 Требования к системе

В ходе разработки вышеупомянутого коммерческого проекта, к системе были предъявлены следующие требования.

3.1 Функциональные требования

1. Система должна предоставлять интерфейс для создания задач для запуска на кластере Mesos, получения списка всех задач, и остановки задачи. Для задач, которые были завершены с ошибкой, должна быть возможность посмотреть сообщение об ошибке, если таковое было предоставлено соответствующим контейнером.
2. При создании задачи необходимо иметь возможность указать следующие параметры:
    a. Docker образ, на основании которого должен быть создан контейнер, содержащий исполняемый код задачи.
    b. Ресурсов, которые необходимы для работы задачи, в виде минимально допустимых значений количества ЦПУ, размеров ОЗУ и дискового пространства.
    c. Переменные окружения для соответствующего задаче процесса.
    d. Путь к исполняемому файлу внутри контейнера, а также аргументы командной строки для этого файла.
3. Создаваемые задачи должны помещаться системой в очередь, при этом первая в очереди задача запускается на исполнение на кластере

при получении первого оффера от Mesos, состав вычислительных ресурсов которого соответствует требованиям задачи.

4. Для принятия решения об изменении размеров кластера, другими словами, стратегии изменения кластера, должен быть программный интерфейс, предусматривающий альтернативные реализации этой стратегии на основании следующих параметров:
    a. Текущей нагрузки на систему (т.е. очереди задач)
    b. Текущие временные ресурсы, выделенные системой
5. Для добавления и удаления выделенных временных вычислительных ресурсов должен быть программный интерфейс, предусматривающий альтернативные реализации процесса изменения размеров кластера. В первую очередь, этот интерфейс должен быть удобен для реализации поддержки разных облачных провайдеров.
6. Система должна быть способна сохранять информацию о запущенных задачах и, что куда более важно, о выделенных ресурсах. При перезапуске системы, эта информация должна быть полностью восстановлена.

3.2 Нефункциональные требования

1. Интерфейс системы должен быть спроектирован с использованием протокола HTTP в соответствии с подходом REST API [12]. Это требование обусловлено тем, что HTTP является достаточно популярным протоколом, для него существует множество реализаций как для разработки серверных, так и клиентских приложений, практически на каждом популярном языке программирования. Подход REST API к проектированию HTTP

интерфейса также является достаточно удобным для разработки многих клиентских приложений. Некоторые из рассмотренных выше аналогичных систем, а именно Marathon и Metronome, предоставляют именно такой HTTP интерфейс и широко используются, что свидетельствует об удобстве подобного подхода для разработчиков. Также в целевом коммерческом проекте взаимодействия между некоторыми подсистемами организованы с помощью аналогичного подхода, поэтому использование его и для разрабатываемой системы позволит использовать уже привычные инструменты для взаимодействия с ней.

2. Система должна быть реализована с использованием языка программирования Scala. Это требование обусловлено тем, что этот язык широко используется и компилируется в байт-код JVM, что позволяет ему быть запущенным на любой из поддерживаемых JVM платформе. Также для языка Scala существует популярный и достаточно удобный, в виду своей высокоуровневости, набор библиотек Akka. Akka позволяет создавать масштабируемые, высокопроизводительные и поддерживаемые системы с использованием модели акторов [13]. Модель акторов будет более подробно разобрана в главе 4.2. Помимо этого, большая часть целевого коммерческого проекта написана на этом языке.
3. В качестве хранилища данных система должна использовать Sqlite. Это требование обусловлено тем, что Sqlite является достаточно простой библиотекой для работы с файлом, как с SQL базой данных [14]. Файлового хранилища достаточно, поскольку доступ к нему необходим только от одного процесса, однако поскольку количество задач, запускаемых с помощью системы, может быть относительно

большим, то возникает необходимость в использовании формата данных, позволяющем организовывать доступ к этим данным эффективно, что предоставляет Sqlite. С другой стороны, использование полноценной СУБД с клиент-серверной архитектурой выглядит слишком громоздко и избыточно для использования системой, нуждающейся в хранении всего нескольких типов сущностей.

4 Проектирование системы

4.1 HTTP интерфейс

Для спецификации HTTP интерфейса был использован стандарт OpenAPI 3.0.2 [15]. Полная спецификация приведена в приложении А. Таблица ниже содержит описание ключевых типов запросов, которые поддерживаются системой. Все представленные команды спроектированы в соответствии с принципом REST API и предполагают использование формата JSON [16] для HTTP тел запросов и ответов. Описание структуры ответов приведено только в случае кода возврата 200 - OK.

Таблица 1 - Запросы HTTP интерфейса

Метод + путь	JSON структура тела:
POST /jobs Создать задачу	**Запрос:** *"id"*: строка - идентификатор *"resources"*: объект - требуемые ресурсы *"resources.mem"*: целое число - количество ОЗУ в мегабайтах *"resources.cpus"*: число с плавающей точкой - количество ЦПУ *"resources.disk"*: целое число - количество дискового пространства в мегабайтах *"dockerImage":* строка - название docker образа *"cmd"*: массив строк - команда для запуска и аргументы командной строки *"env"*: объект, содержащий строки - переменные окружения

Продолжение таблицы 1

	Ответ: *"id"*: строка - идентификатор *"resources"*: объект - требуемые ресурсы *"resources.mem"*: целое число - количество ОЗУ в мегабайтах *"resources.cpus"*: число с плавающей точкой - количество ЦПУ *"resources.disk"*: целое число - количество дискового пространства в мегабайтах *"dockerImage":* строка - название docker образа *"cmd"*: массив строк - команда для запуска и аргументы командной строки *"env"*: объект, содержащий строки - переменные окружения *"status"*: строка - статус *"created"*: строка - дата и время создания задачи *"updated"*: строка - дата и время последнего обновления задачи *"completed"*: строка - дата и время обновления задачи

Продолжение таблицы 1

GET /jobs получить список задач	**Ответ:** *"count"*: целое число - количество задач *"items"*: массив объектов - задачи. Структура объектов та же, что и в ответе на запрос POST /jobs
POST /jobs/<id>/cancel отменить выполнение задачи с идентификатором id	Не содержит тела
DELETE /jobs/<id> удалить задачу с идентификатором id	Не содержит тела

4.2 Модель акторов

Особенности интерфейса Mesos, а также асинхронная и даже поддающаяся распараллеливанию природа процесса наблюдения за исполнением задач привели к использованию модели акторов для реализации системы. В связи с этим, в этой главе приводится краткий обзор модели акторов. Модель акторов это математическая модель, которая была создана, чтобы формализовать распределённые вычисления, и служит в качестве теоретической базы для ряда программных инструментов [17]. Эти инструменты используются, например, для разработки различных приложений, работающих с асинхронными интерфейсами или использующих параллельные вычисления. Имея под собой теоретическую базу, эти приложения работают с некоторыми

высокоуровневыми концепциями, что упрощает разработку и поддержку таких приложений без потери преимуществ, которые дают асинхронные и параллельные модели.

В основе модели лежат понятия актора и сообщения. Актор это универсальная вычислительная сущность, то есть сущность, которая производит вычисления. Сообщение это набор данных, который актор может получить от другого актора или отправить другому актору. Акторы постоянно обмениваются друг с другом сообщениями, выполняют какие-то действия в ответ на получаемые сообщения, создают других акторов, меняют свои состояния и “умирают”, когда больше не нужны. Будучи сущностью с состоянием и поведением, актор очень похож на объект из мира ООП. Сообщения же достаточно близки по своей сути к одноимённому понятию сообщения из ООП. В современном и повсеместном ООП чаще говорят “вызвать метод”, нежели “отправить сообщение”, но такую терминологию ещё можно застать в более “классическом” (раннем) ООП, которое можно найти в таких языках, как Smalltalk, где вообще ООП возведено в абсолют: даже управляющие конструкции типа if не существуют, а реализованы через получение сообщений.

Тем не менее, модель акторов существует отдельно от ООП не просто так, и одним из фундаментальных свойств акторов является то, что они обрабатывают только одно сообщение в один момент времени. В отличие от метода класса, который можно из нескольких потоков вызвать несколько раз одновременно, что сильно усложняет разработку распределённых систем на ООП, актор обрабатывает сообщения по очереди и никогда не делает это параллельно - новые сообщения попадают в “ящик” (mailbox) и только когда актор будет готов, начинают им

обрабатываться. Также акторы могут менять своё поведение в ответ на полученное сообщение. Этого в принципе можно добиться и в ООП с помощью различных паттернов, но в модели акторов это свойство первого порядка, и в том числе на его основе доказываются теоремы модели акторов, имеющие практическое значение. Это свойство также делает акторов удобным инструментом для моделирования автоматов.

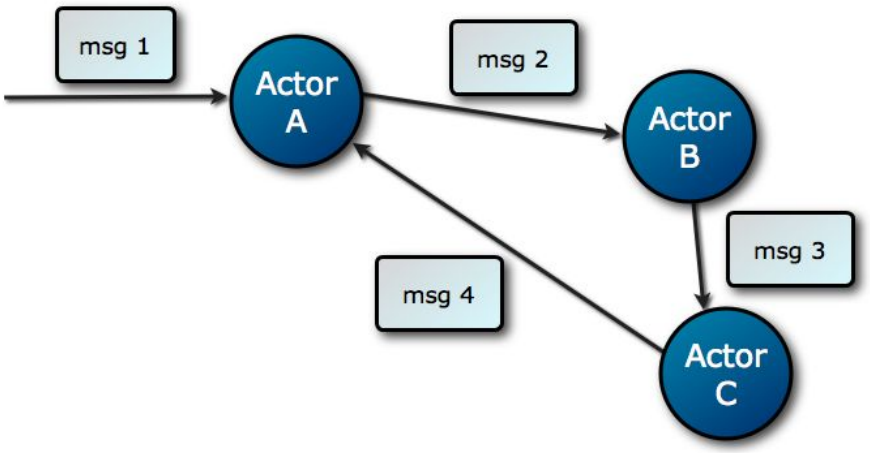


Рисунок 3 - пример диаграммы, изображающей набор акторов и сообщения, посылаемые ими друг другу [18]

Для работы с моделью акторов в разрабатываемой системе используется инструментарий Akka, предоставляющий набор программных примитивов и исполняемую среду для создания акторов и передачи сообщений между ними.

4.3 Подсистема запуска задач на кластере Mesos

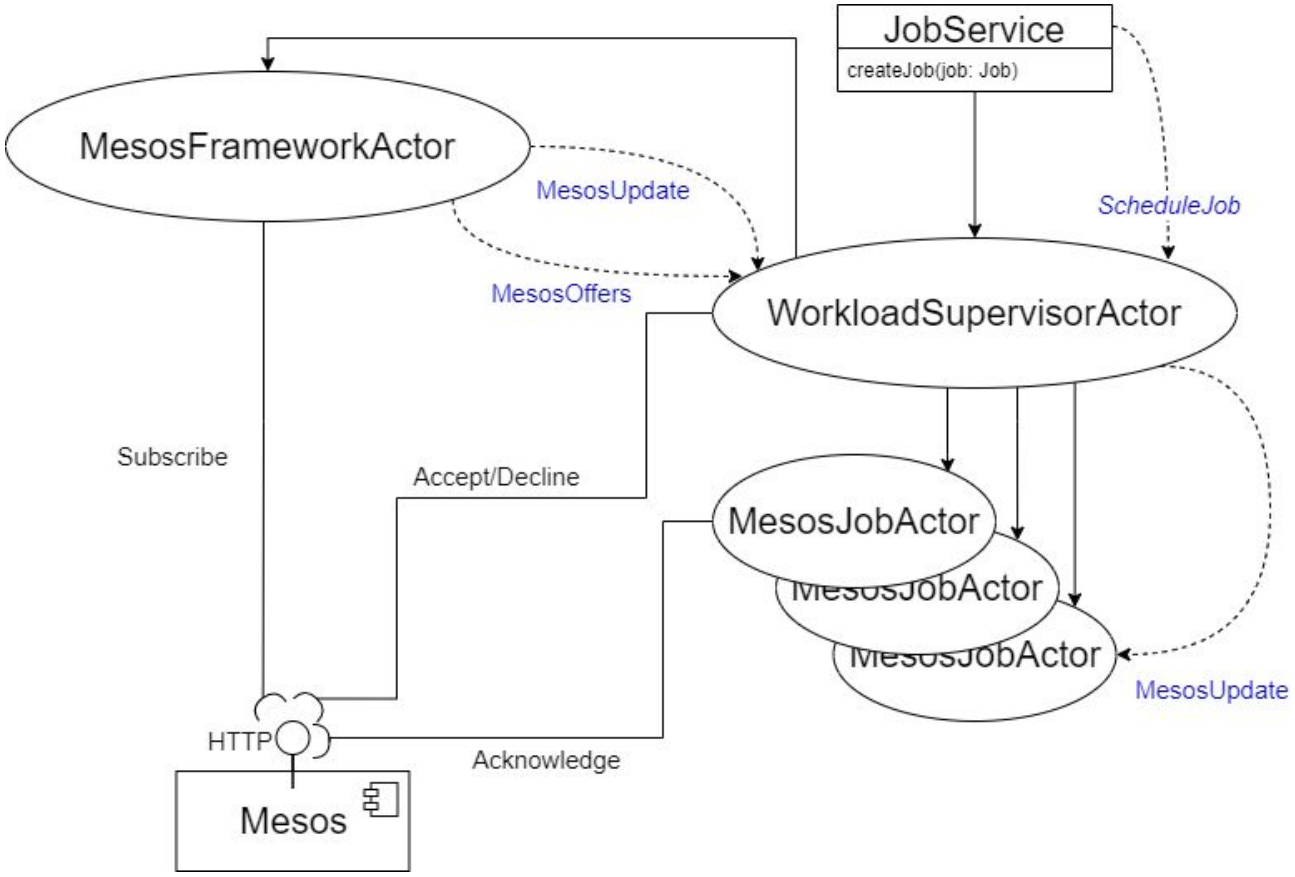


Рисунок 4 - диаграмма сущностей подсистемы запуска задач на кластере Mesos и связей между ними и самим Mesos

На представленной диаграмме содержатся основные элементы подсистемы, отвечающей за запуск задач на кластере Mesos. Точкой входа в эту подсистему для остальных модулей является класс JobService,

имеющий метод createJob, принимающий объект класса Job - представление вычислительной задачи, которая создаётся в системе. MesosFrameworkActor представляет собой тип актора, который регистрируется как фреймворк-планировщик для экземпляра Mesos, с которым работает система. Mesos предоставляет HTTP интерфейс с точкой входа Subscribe, сделав соответствующий запрос к которой, клиент становится фреймворком и непрерывно получает события, пока не оборвёт соединение. Существует несколько типов таких событий, но здесь интересны два: Offers и Update. Получив информацию о событии от Mesos, актор типа MesosFrameworkActor оборачивает его в соответствующий ему тип сообщения и посылает тому актору, который на него подписан. Этим актором является актор типа WorkloadSupervisorActor - центральный и самый сложный элемент системы, в обязанности которого входят, среди прочего:

- Управление очередью задач. Получив сообщение типа ScheduleJob, содержащей информацию о задаче, актор добавляет задачу в конец очереди.
- Обновление информации о задачах в хранилище задач.
- Принятие или отклонение очередных офферов от Mesos в виде сообщений типа MesosOffers в зависимости от требований первой задачи в очереди. Если оффер принимается, то актор посылает Mesos HTTP запрос с командой Accept и создаёт дочернего актора типа MesosJobActor, обязанности которого будут описаны далее. В случае, если оффер не принимается, актор типа WorkloadSupervisorActor отправляет Mesos HTTP запрос с командой Decline.

- Перенаправление сообщений типа MesosUpdate с информацией об обновлённом статусе процесса на кластере Mesos соответствующему дочернему актору типа MesosJobActor.

Последний тип актора - MesosJobActor, наблюдает за прогрессом задачи посредством реакции на сообщения типа MesosUpdate. Сообщение содержит информацию об обновлённом статусе процесса, где статус это одно из значений перечисления (enum), согласно интерфейсу Mesos. Это перечисление содержит более десятка возможных значений с разной семантикой, и актор обрабатывает их, в соответствии с этой семантикой. Реакцией на изменение статуса является обновление соответствующих данных о задачи в хранилище. Завершающим этапом обработки сообщения типа MesosUpdate является отправка Mesos HTTP запроса с командой Acknowledge, согласно протоколу взаимодействия с Mesos.

4.4 Подсистема изменения размеров кластера Mesos

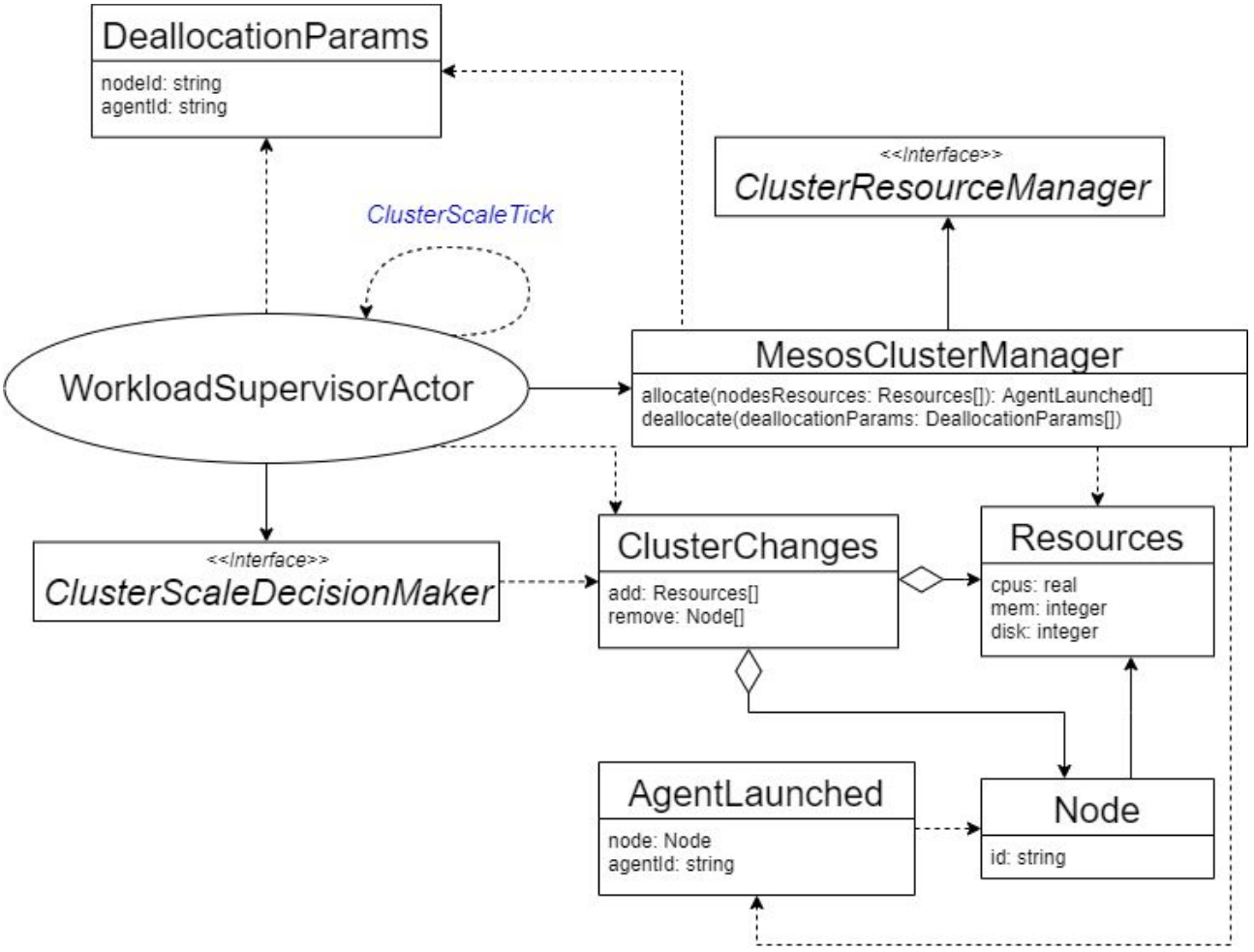


Рисунок 5 - диаграмма сущностей подсистемы изменения размеров кластера Mesos и связей между ними

На представленной диаграмме содержатся основные элементы подсистемы, отвечающей за изменение размеров кластера Mesos. Поясним сперва некоторые простые классы, являющиеся примером использования объектного паттерна “объект-значение” и представляющие собой объектные представления сущностей предметной области или обёртки параметров или возвращаемых значений:

- Resources - представление вычислительных ресурсов в виде количества ЦПУ, размера ОЗУ и дискового пространства.
- Node - представление вычислительного узла (в данном случае, конкретнее, виртуальной машины), обладающим некоторыми ресурсами и идентификатором.
- ClusterChanges - представление решения об изменении ресурсов кластера в виде выделения некоторого количества ресурсов (атрибут add) и/или удаления некоторого количества узлов (атрибут remove).
- AgentLaunched - представление результата успешного запуска Mesos агента на некотором узле.
- DeallocationParams - представление параметров удаляемого из кластера вычислительного узла с соответствующим ему идентификатором Mesos агента.

С определённым периодом (задаваемым перед стартом системы) актор типа WorkloadSupervisorActor сам себе посылает сообщение типа ClusterScaleTick. Реакцией на это сообщение является обращение к объекту, реализующему интерфейс ClusterScaleDecisionMaker, обязанностью которого является принятие решения об изменении размеров кластера на основании ряда параметров, среди которых текущий размер очереди задач. В целях упрощения диаграммы, сигнатура соответствующего метода, который будет вызван актором, опущена. Эта сигнатура будет представлена и подробно разобрана в главе 4.5.1. Результатом вызова этого метода является, опционально, объект типа ClusterChanges. В том случае, если объект был возвращён, то есть было принято решение изменить размер кластера, WorkloadSupervisorActor обращается к объекту класса MesosClusterManager, вызывая у него методы

allocate и deallocate для добавления и удаления ресурсов в кластер Mesos соответственно. Обязанностью класса MesosClusterManager является управление агентами Mesos, будь то создание новых агентов, если было принято решение расширить кластер, или завершение действующих агентов, если было принято решение сузить кластер. Однако MesosClusterManager не производит создание и удаление самих серверов (обычно, виртуальных машин) самостоятельно - для этого существует интерфейс ClusterResourceManager. Этот интерфейс реализуется для соответствующего источника вычислительных ресурсов, например, облачного провайдера. Методы этого интерфейса также опущены в целях упрощения и без того громоздкой диаграммы и будут подробно рассмотрены в главе 4.5.2. Если необходимо добавить ресурсы в кластер, то вызвав соответствующий метод ClusterResourceManager и получив информацию о созданных вычислительных узлах, объект типа MesosClusterManager запускает на них Mesos агент, чтобы закончить процесс регистрации новых ресурсов в кластере. Стоит отметить, что для этого используется немного другая часть интерфейса Mesos; она предназначена не для фреймворков-планировщиков, а для так называемых программ-операторов Mesos, которые управляют ресурсами кластера. Если необходимо удалить ресурсы из кластера, то объект типа MesosClusterManager сперва останавливает соответствующий агент в кластере Mesos и только после этого вызывает метод для удаления сервера интерфейса ClusterResourceManager. Такой порядок важен, поскольку если сперва удалить сервер, что обычно означает удаление виртуальной машины в облаке, то, мастер Mesos, потеряв связь с агентом, не удалит его из кластера, а пометит как временно недоступный - Unreachable и даже продолжит предлагать фреймворкам его ресурсы. Так происходит,

поскольку Mesos также обеспечивает отказоустойчивость кластера, и для него внезапно переставший работать сервер не обязательно значит, что его специально удаляют: в ряде случаев ожидается, что сервер восстановит работу и агент снова будет запущен. Такое поведение полезно во многих ситуациях, но в нашем случае требуется именно полное удаление сервера из кластера, поэтому приходится явно сообщать об этому мастеру Mesos. MesosClusterManager также обновляет информацию о работающих узлах в соответствиями с изменениями кластера. Как уже было упомянуто в требованиях, это важно, поскольку при перезапуске системы крайне неприятным будет потеря информации о созданных виртуальных машинах, ведь пока их кто-то не удалит вручную, они будут тарифицироваться облачным провайдером, что может привести к катастрофическим денежным потерям.

4.5 Точки расширения системы

Согласно функциональным требованиям, система должна обладать программными интерфейсами, позволяющими:

- Реализовывать разные стратегии изменения размеров кластера на основании информации о текущей очереди задач и уже выделенных ресурсах
- Реализовывать работу с разными источниками вычислительных ресурсов для их добавления в систему или удаления из системы при необходимости

Два этих требования прямо отображаются в интерфейсы, уже упомянутые в главе 4.4. Эти интерфейсы предназначены для разработчиков и позволяют расширять функционал системы при

необходимости. В этой главе будет проведён более подробный разбор этих интерфейсов.

4.5.1 Поддержка новых стратегий изменения размеров кластера

Для принятия решения об увеличении или уменьшении мощностей кластера может возникнуть потребность в специфической или сложной стратегии, где текущую очередь или выделенные ресурсы надо учитывать особым образом. Например, согласно бизнес-логике, необходимо уделять особое внимание задачам, описываемым определённым Docker образом. Или обеспечивать как можно меньшее время аренды серверов с большим количеством ресурсов. В таких случаях предлагается реализовать интерфейс ClusterScaleDecisionMaker.

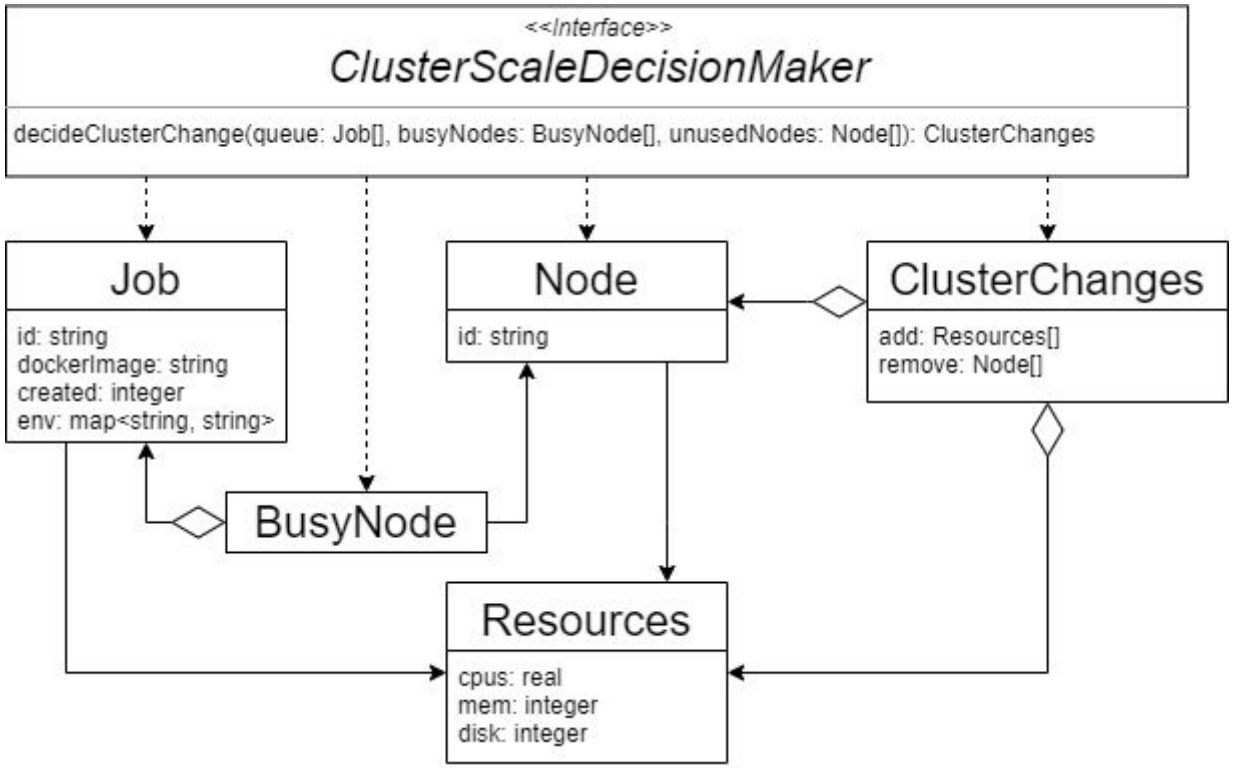


Рисунок 6 - диаграмма, описывающая интерфейс ClusterScaleDecisionMaker и связанные сущности

На диаграмме видно участие уже упомянутых в предыдущих главах сущностей. Единственный новый класс - BusyNode, является объектным представлением вычислительного узла, на котором в текущий момент запущены задачи.

Интерфейс ClusterScaleDecisionMaker состоит из одного метода - decideClusterChange, который принимает на вход три аргумента:

1. queue - очередь задач. Представляет собой список объектов класса Job.
2. busyNodes - выделенные системой узлы, на которых в данный момент работают задачи. Представляет собой список объектов класса BusyNode.
3. unusedNodes - выделенные системой узлы, на которых в данный момент нет работающих задач. Представляет собой список объектов класса Node.

В качестве возвращаемого значения выступает объект типа ClusterChange, который описывает, какие ресурсы должны быть добавлены в кластер, а какие вычислительные узлы - удалены. При этом может быть принято решение не менять размер кластера вообще - этот случай предлагается представлять возвращением “пустого” значения вместо объекта. Понятия пустого значения в разных языках программирования представлены по-разному. В языке Scala для этого фактически используется тип Option[ClusterChanges], где Option[T] - класс-шаблон языка Scala, который представляет собой возможно отсутствующее значение типа T.

Таким образом, разработчик новых стратегий изменения размера кластера волен использовать следующие данные, которые будут

предоставлены актором типа WorkloadSupervisorActor во время работы системы:

- Текущие задачи в очереди, причём не только их количество и порядок, но также метаданные о любой задаче, такие как название Docker образа (атрибут dockerImage), дату создания (атрибут created) или даже переменные окружения (атрибут env, представляет собой Map - отображение (или словарь) из строки в строку).
- Уже выделенные системой узлы. Причём эти узлы разнесены на два множества - утилизируемые задачами и неутилизируемые. Реализация интерфейса может отслеживать, как долго/часто какие-то узлы не утилизируются и принять решение удалить их. Стоит отметить, что по понятным причинам нельзя возвращать объекты из списка unusedNodes в атрибуте remove объекта ClusterChanges (тип BusyNode тривиально переводится в тип Node), что семантически значит запрос на удаление из кластера узлов, на которых ещё не закончились задачи. Система страхует от подобной ошибки разработчика и не станет удалять такие узлы, даже если они входили в состав объекта с результатами: в этом случае в журнал системы будет отправлено соответствующее сообщение с пометкой WARNING.

4.5.2 Поддержка новых источников вычислительных ресурсов

В мире существует множество облачных провайдеров, и новые появляются регулярно, а также сами приложения могут мигрировать от одного провайдера к другому. В связи с этим возникает необходимость в механизме, который позволил бы разрабатываемой системе работать с новыми поставщиками вычислительных ресурсов. Для этих целей был создан интерфейс ClusterResourceManager.

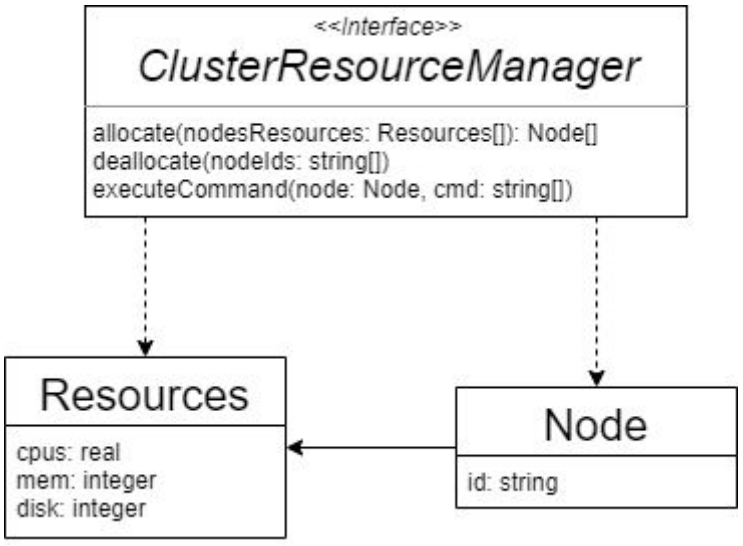


Рисунок 7 - диаграмма, описывающая интерфейс ClusterResourceManager и связанные сущности

На представленной диаграмме отображено три метода, которые необходимо реализовать для поддержки нового источника ресурсов: allocate, deallocate и executeCommand. Эти методы будут использоваться классом MesosClusterManager во время работы системы, когда будет принято решение изменить размер кластера Mesos. Рассмотрим каждый из них подробнее.

Метод allocate принимает список наборов ресурсов и возвращает список узлов. Таким образом, задача этого метода - превратить набор запрошенных ресурсов в набор конкретных узлов, вычислительные характеристики которых удовлетворяют запросу. При этом узел также характеризуется идентификатором, который должен быть уникальным в контексте источника данных. Как правило, крупные облачные провайдеры, предоставляющие API для управления ресурсами аккаунта, предоставляют способы уникальной идентификации абстракции виртуальной машины или физического сервера. Отметим, что не всегда возможно запросить конкретный набор вычислительных ресурсов: зачастую облачные провайдеры предоставляют определённый набор тарифов, которым соответствуют те или иные характеристики арендуемой машины. Любая разумная реализация интерфейса должна это учитывать, округляя запрошенные ресурсы вверх при выборе соответствующего тарифа. Например, при запросе следующего набора ресурсов:

- Resources(cpus = 1.5, mem = 2 * 1024, disk = 10 * 1024),

и наличии исключительно следующих возможных вариантов создания узлов (тарифов):

- lite - 1.0 CPUs, 2GB RAM, 10GB SSD,
- medium - 2.0 CPUs, 4GB RAM, 40GB SSD,

соответствующая реализация интерфейса должна выбрать вариант medium, несмотря на то, что вариант lite удовлетворяет требованиям по памяти и дисковому пространству.

Также отметим, что размер результирующего списка узлов не обязательно должен быть равен размеру входящего списка ресурсов. В некоторых случаях может оказаться, что выгоднее выделить один узел для

покрытия сразу нескольких наборов ресурсов. Например, при запросе следующих наборов ресурсов:

- Resources(cpus = 1.0, mem = 1 * 1024, disk = 10 * 1024),
- Resources(cpus = 2.0, mem = 2 * 1024, disk = 20 * 1024),
- Resources(cpus = 4.0, mem = 4 * 1024, disk = 40 * 1024)

и наличии исключительно следующих возможных вариантов создания узлов (тарифов) с соответствующими стоимостями:

- micro - 1.0 CPUs, 1GB RAM, 10GB SSD - 0.02$ / hour,
- lite - 2.0 CPUs, 2GB RAM, 20GB SSD - 0.03$ / hour,
- medium - 4.0 CPUs, 4GB RAM, 40GB SSD - 0.04$ / hour

возможны следующие варианты результирующего набора узлов:

1.
    - Node(id = "micro1", Resources(cpus = 1.0, mem = 1 * 1024, disk = 10*1024))
    - Node(id = "lite1", Resources(cpus = 2.0, mem = 2 * 1024, disk = 20*1024))
    - Node(id = "medium1", Resources(cpus = 4.0, mem = 4 * 1024, disk = 40*1024))
2.
    - Node(id = "medium1", Resources(cpus = 4.0, mem = 4 * 1024, disk = 40*1024))
    - Node(id = "medium2", Resources(cpus = 4.0, mem = 4 * 1024, disk = 40*1024))

Общая стоимость варианта 1 составляет 0.02 + 0.03 + 0.04 = 0.09$ в час. Общая стоимость варианта 2 составляет 0.04 + 0.04 = 0.08$ в час. В этом случае предпочтительнее вариант 2, поскольку он не только полностью удовлетворяет требованиям по запрашиваемым ресурсам (и даже

предоставляет больше в итоге), но и обойдётся дешевле. Более “наивная” реализация, рассматривающая каждый набор запрашиваемых ресурсов независимо, выбрала бы вариант 1, в котором каждый из наборов ресурсов находит естественное идеальное отображение в соответствующий тариф. Такая реализация хоть и отвечает требованиям, но влечёт за собой больше расходов.

Также на выделяемые узлы накладывается требование по наличию окружения, в котором присутствуют исполняемые файлы для запуска агентов Mesos. Хорошим способом удовлетворить это требования является использование функциональности так называемых снимков (англ. snapshots) виртуальных машин, предоставляемой многими крупными облачными провайдерами. Суть этого механизма в том, что экземпляр виртуальной машины создаётся на основании указанного шаблона, в котором уже подготовлено определённое окружение. Это довольно часто используется для облегчения расширения облачной инфраструктуры для проекта: вместо ручной настройки окружения на каждой вновь создаваемой машине, создаётся образ, где установлены все нужные пакеты, выставлены необходимые параметры и так далее. Этот образ позже используется в качестве базового для вновь создаваемой виртуальной машины, позволяя сразу вводить её в эксплуатацию в соответствующем проекте. В нашем случае это очень удобно, поскольку требуется конкретное окружение, которое позволяет использовать вновь созданный узел в качестве Mesos агента. При отсутствии такого механизма, задача усложняется и требует собственной реализации процесса подготовки окружения, о чём должен озаботиться разработчик, реализующий интерфейс для соответствующего источника данных.

Метод deallocate принимает список идентификаторов узлов, которые требуется удалить, т.е. вернуть источнику данных. Поскольку этот метод будет вызван уже после удаления из кластера Mesos соответствующих агентов, то реализация не требует ничего поверх отправки запроса на удаление виртуальных машин или чего-то аналогичного.

Метод executeCommand принимает узел и команду вместе с аргументами командной строки, которую необходимо запустить. Этот метод используется для запуска агента Mesos с определёнными параметрами, которые сформирует система. Во многих случаях этот метод может быть реализован с использованием протокола SSH, но некоторые облачные провайдеры предоставляют возможность запускать команды собственными средствами API. Для большей гибкости разработчику предоставляется возможность использовать любой из этих подходов, или возможно какой-то иной, для реализации этого метода.

4.6 Структура пакетов

В этой главе будут описаны основные пакеты, группирующие классы и другие сущности системы, и связи между ними.

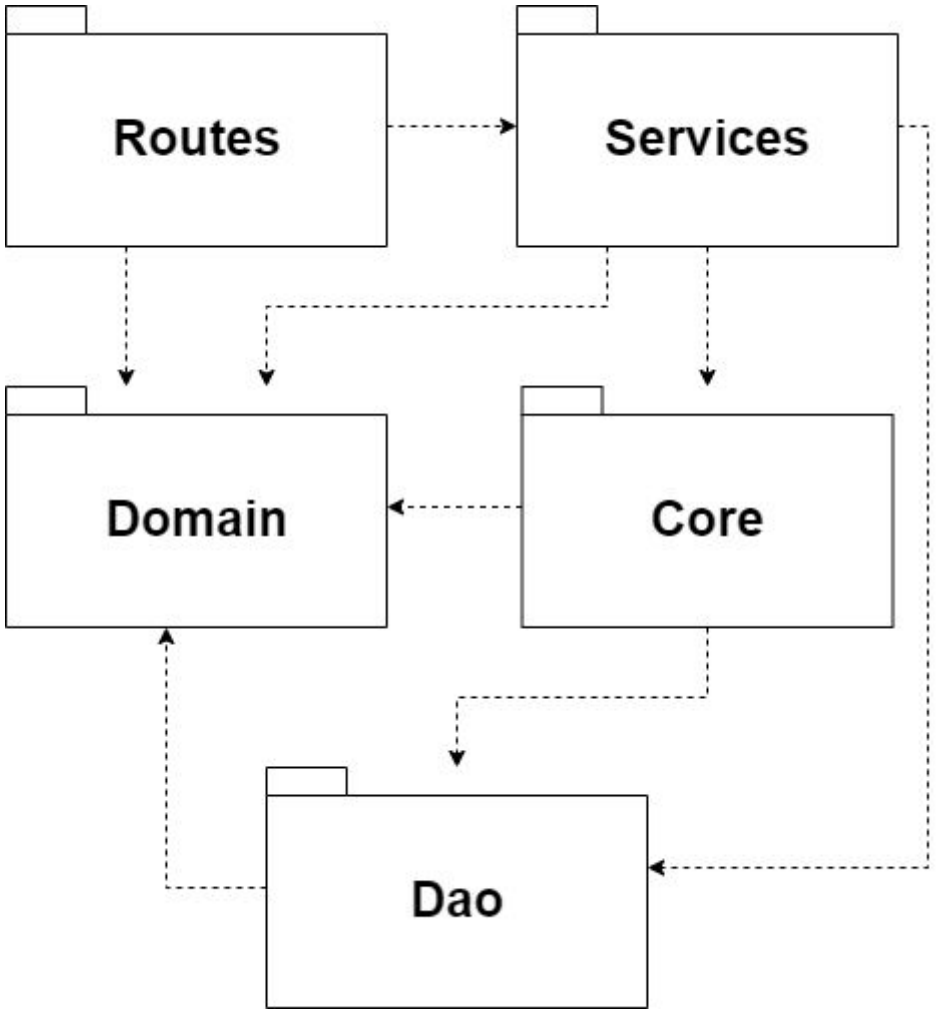


Рисунок 8 - диаграмма, описывающая пакеты системы и связи между ними

Ниже приведено описание каждого из пакетов, представленных на диаграмме:

- Routes - содержит код, реализующий HTTP интерфейс системы. Написан с использованием библиотеки Akka HTTP, позволяющей создавать HTTP серверы с использованием удобного DSL.
- Services - содержит классы, реализующие бизнес-логику системы и служащие точкой входа для различных внешних интерфейсов системы (в данный момент - только HTTP интерфейса). Сюда входит класс JobService, предоставляющий CRUD методы для сущности Job.
- Domain - содержит классы, представляющие сущности предметной области. Сюда входят такие классы, как Resources, Job, Node, BusyNode, и другие.
- Core - содержит классы и описания акторов, реализующие всю работу с Mesos и внешними источниками ресурсов. Сюда, среди прочего, входят такие классы, как MesosClusterManager, типы акторов MesosFrameworkActor, WorkloadSupervisorActor и MesosJobActor, а также интерфейсы ClusterScaleDecisionMaker и ClusterResourceManager.
- Dao - содержит классы-шлюзы для хранения информации о созданных задачах и текущих выделенных узлах.

5 Реализация системы

В этой главе будет представлен обзор некоторых особенностей реализации системы. Будут более подробно описаны некоторые инструменты, с использованием которых велась работа над системой. Также будут рассмотрены конкретные реализации вышеописанных программных интерфейсов, представляющих собой точки расширения системы, по одной на каждый, которые можно использовать в качестве документации для разработки новых реализаций.

5.1 Подготовка окружения. Travis CI

При разработке информационных систем, в том числе в рамках данной работы, важно контролировать качество кода, выражающееся в:

1. Покрытии кода тестами. За последние года многие проекты пришли к понимаю, что автоматизированное тестирование сильно упрощает разработку сложных и развивающихся систем. В связи с этим получили широкое распространение инструменты, вычисляющие соотношение покрытого тестами кода к его общему объёму. Рассчитывая эту величину каждый раз при внесении изменений в код проекта, можно поддерживать её на определённом уровне, например, 90%. Когда есть уверенность, что 90% функционала покрыта тестами, то это позволяет считать, что большинство ошибок выявляются до развёртывания приложения.
2. Соответствие кода выбранным стилистическим соглашениям. Сложные информационные системы часто подразумевают, что работа над ними будет вестись не одним человеком. При этом

многие программисты имеют собственные представления о читаемости кода, которые не всегда совпадают. Если позволить каждому разработчику следовать личным предпочтениям в таких вопросах, как, например, подход к именованию переменных или функций, размеру отступа при переносе строк и другим параметрам, то со временем база кода превратится в тяжело читаемую смесь различных стилей. Такой код сложнее поддерживать, добавляя новые функции в систему. Для решения этой проблемы часто вводят так называемые визуальностилистические руководства (англ. style guide), которые однозначно регламентируют подходы к оформлению кода. После обсуждений, порой бурных и продолжительных, разработчики приходят к компромиссному варианту, учитывающего интересы их всех, и обязуются соблюдать правила такого варианта. Поскольку ручной аудит кода на соответствие принятым соглашением выполнять долго и обычно неинтересно, были созданы соответствующие инструменты статического анализа кода, занимающиеся именно этим аудитом.

3. Отсутствии в коде различных антипаттернов, которые можно обнаружить при статическом анализе. Этот пункт похож на предыдущий, но отличие в том, что речь здесь идёт не о стилистике и личных предпочтениях, а в объективно вредных для работы и поддержки системы участков кода. Примером такого участка может быть вызов устаревшей функции, недостижимый код или инструкция по включению в компиляцию модуля, который фактически не используется. Большинство таких проблемных участков можно также отловить до исполнения кода, для чего были

созданы инструменты, проводящие статической анализ кода на предмет наличия таких антипаттернов.

Эти показатели качества кода следует оценивать регулярно, а лучше непрерывно, при любых изменениях. Для этого также были созданы соответствующие инструменты, позволяющие описать набор команд, которые следует выполнить в отношении кода проекта по наступлению определённого события. Практика использования таких инструментов называется непрерывная интеграция (англ. continuous integration), и в данном проекте она также применяется. В качестве системы непрерывной интеграции использовался платный облачный сервис Travis CI [19] с интеграцией с сервисом хранения кода GitHub [20], поскольку все остальные части коммерческого проекта также использовали их по решению заказчика.

Travis CI работает с системой контроля версий git, являющейся самой популярной системой такого рода [21]. В виду популярности git и наличия большого количества открытых ресурсов с его подробным описанием, не будем подробно на нём останавливаться. Скажем лишь, что Travis CI настраивается путём добавления в репозиторий с кодом специального конфигурационного файла .travis.yml (или .travis.yaml), который инструктирует облачный сервис запускать команды из этого файла при каждой фиксации изменений в репозитории (git commit). Содержимое файла составляется в соответствии с декларативным языком конфигурации Travis CI.

При разработке системы для непрерывной интеграции были дополнительно использованы следующие технологии:

- sbt - инструмент для разработки проектов на языке Scala [22]. Функционал включает в себя, помимо прочего, скачивание

зависимостей проекта, компиляцию проекта, запуск тестов и сборку исполняемых артефактов для запуска на JVM. Соответствующие sbt команды sbt - update, compile, test и packageBin. При этом во время компиляции проводится анализ на наличие потенциально проблемных или бесполезных участков кода - например, инструкции import, добавляющие в зону видимости .scala файла пространства имён, которые по факту не используются.

- scoverage - инструмент для определения уровня покрытия тестами программ, написанных на язые Scala [23]. Для интеграции с sbt был использован плагин sbt-scoverage [24], добавляющий команды coverage и coverageReport. Он отслеживает, какие участки кода были вызваны при запуске тестов и формирует ряд отчётов в различных форматах, в том числе в HTML.

```
32
33   def create(
34     id: String,
35     resources: Resources,
36     dockerImage: String,
37     cmd: Option[List[String]],
38     env: Option[Map[String, String]]
39   ): Future[Job] =
40     for {
41       existingJob <- dao.get(id)
42       _ = if (existingJob.isDefined) throw new RuntimeException(s"Job with id '$id' already exists")
43       now = Instant.now()
44       job = Job(
45         id = id,
46         resources = resources,
47         dockerImage = dockerImage,
48         cmd = cmd.getOrElse(List.empty),
49         env = env.getOrElse(Map.empty),
50         status = JobStatus.Pending,
51         error = None,
52         created = now,
53         updated = now,
54         completed = None
55       )
56       createdJob <- dao.create(job)
57       _ = workloadSupervisorActor ! WorkloadSupervisorActor.ScheduleJob(createdJob)
58     } yield createdJob
59
60   def list(): Future[ListResult[Job]] =
61     for {
62       jobs <- dao.list()
63       count <- dao.count()
64     } yield ListResult(jobs, count)
65
```

Рисунок 9 - пример отчёта scoverage. Зелёные участки кода покрыты тестами. Красные - нет

- scalafmt - инструмент для статической проверки Scala кода на предмет соответствия соглашений по стилю кода [25]. Настройки стиля задаются в конфигурационном файле .scalafmt.conf. Для интеграции с sbt был использован плагин sbt-scalafmt [26], добавляющий команду scalafmtCheckAll, которая показывает все несоответствия кода конфигурации, если таковые имеются.
- scalafix - инструмент для автоматического изменения или анализа Scala кода проекта согласно правилам, прописанным в конфигурационном файле .scalafix.conf [27]. В качестве примера правила, используемого в проекте, можно привести SortImports - при

наличии в файле инструкций import, не отсортированных в алфавитном порядке согласно добавляемого в ими в область видимости пространства имён, - отсортировать их соответствующим образом. Для интеграции с sbt был использован плагин sbt-scalafix [28], добавляющий команду scalafixCheckAll, которая показывает все несоответствия кода конфигурации, если таковые имеются.

- coveralls - облачный сервис, позволяющий отслеживать изменения процента покрытия кода тестами с возможностью уведомления при падении покрытия ниже установленного минимума или на определённую величину в сравнении с предыдущей версией кода [29]. Для интеграции с sbt был использован плагин sbt-coveralls [30], добавляющий команду coveralls, которая отсылает предварительно сгенерированный отчёт о покрытии кода тестами, и отправляет его на сервер coveralls.

С использованием перечисленных технологий, одной из действий, выполняемых облачным сервисом Travis CI для непрерывной интеграции, является выполнение следующей команды:

sbt scalafmtCheckAll scalafixCheckAll coverage test coverageReport coveralls

которая последовательно проверяет соответствие кода заданным конфигурациям, активирует отслеживание покрытия кода тестами, генерирует соответствующий отчёт и отправляет его на сервер coveralls.

Таким образом, при каждом обновлении кода, проводится целый ряд проверок, что позволяет быть уверенным, что код всегда соответствует определённым стандартам оформления и качества, даже если разработчик забывает проверить это у себя на компьютере вручную.

5.2 Простая стратегия изменения размеров кластера

Для тестирования функционала системы была создана примитивная реализация интерфейса ClusterScaleDecisionMaker в виде класса под названием MaxNJobsScaleDecisionMaker.

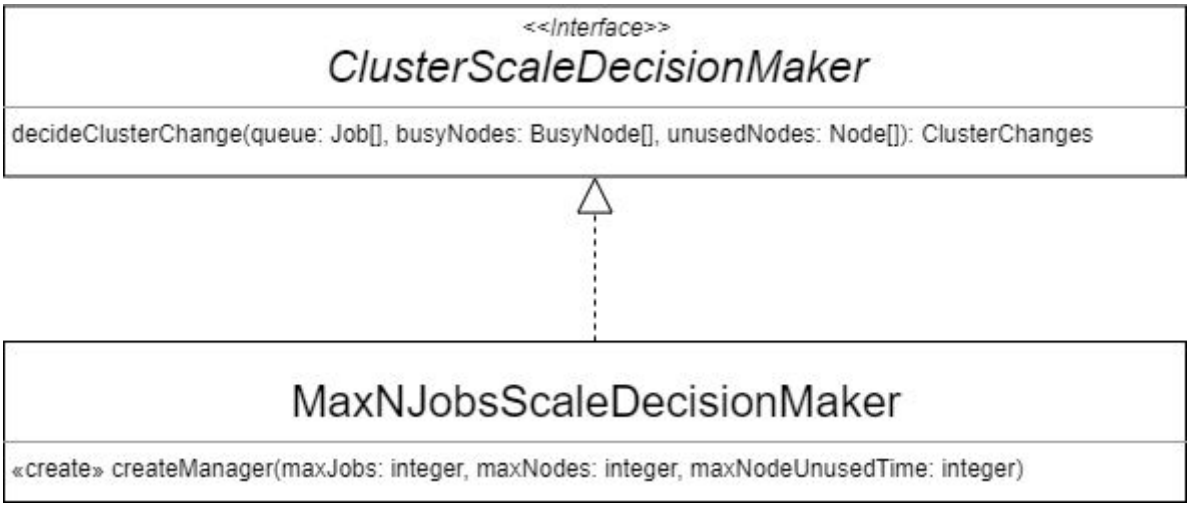


Рисунок 10 - диаграмма, демонстрирующая класс MaxNJobsScaleDecisionMaker

Этот класс содержит конструктор, принимающий три параметра:

1. maxJobs - максимально допустимое количество задач в очереди
2. maxNodes - максимально допустимое количество одновременно работающих узлов, созданных системой автоматически
3. maxNodeUnusedTime - максимально допустимое время в минутах, которое выделенный узел может не быть использован

Стратегия работает по простому алгоритму. Если количество задач в очереди превышает maxJobs и при добавлении новых ресурсов количество узлов не превысит maxNodes - принимается решение добавить количество ресурсов, соответствующее первой в очереди задаче. Все узлы, на которых

задачи не запускались последние maxNodeUnusedTime минут или дольше, выбираются для удаления из кластера.

Таким образом, стратегия MaxNJobsScaleDecisionMaker старается держать размер очереди не больше заданного N, но не превышая установленный предел количества узлов, а также удаляет долго неиспользуемые узлы.

5.3 Поддержка облачного провайдера Digital Ocean

Для тестирования функционала системы была создана полноценная реализация интерфейса ClusterResourceManager для работы с ресурсами облачного провайдера Digital Ocean [31] в виде класса под названием DigitalOceanResourceManager.

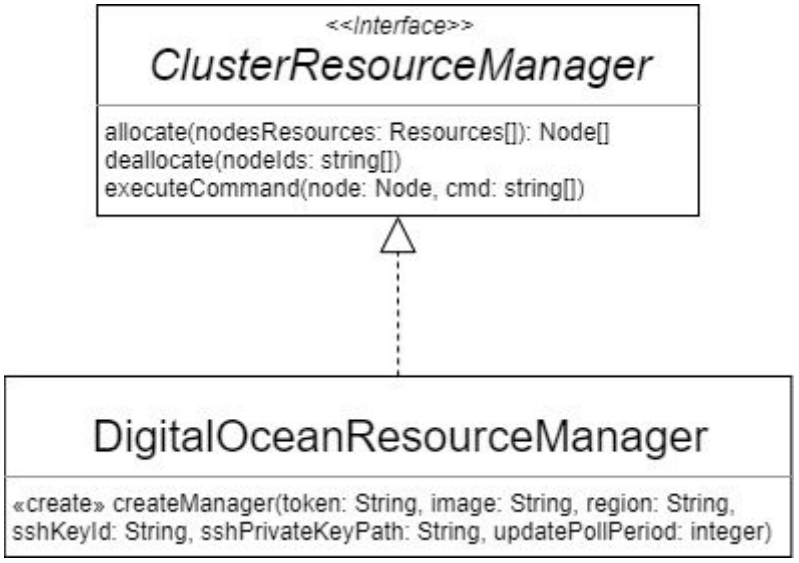


Рисунок 11 - диаграмма, демонстрирующая класс DigitalOceanResourceManager

Класс DigitalOceanResourceManager использует HTTP API Digital Ocean [32]. Для разработки была использована Java библиотека digitalocean-api-client [33], предоставляющая чуть более удобные объектные обёртки над сущностями Digital Ocean и операциями между ними, чем голый HTTP интерфейс. Конструктор класса содержит следующий набор параметров:

1. token - приватный токен для доступа к HTTP интерфейсу Digital Ocean, генерируемый через веб интерфейс провайдера в личном кабинете
2. image - идентификатор образа, используемого в качестве базы при создании виртуальных машин. Должен включать в себя исполняемый файл Mesos агента, доступного для вызова по относительному пути "mesos-agent" при подключении по SSH под пользователем root. Простым вариантом получения такого образа является ручное создание виртуальной машины в личном кабинете Digital Ocean на основе нужного “пустого” базового образа (например, CentOS 7.0) с последующей установкой Mesos и создания конечного образа на основе состояния виртуальной машины
3. region - идентификатор региона Digital Ocean
4. sshKeyId - идентификатор открытой части ssh ключа, заранее добавленного в аккаунт Digital Ocean
5. sshPrivateKeypath - путь к файлу, содержащий закрытую часть ssh ключа, которому соответствует sshKeyId
6. updatePollPeriod - период в секундах, согласно которому будут регулярно запрашиваться состояния виртуальных машин и других сущностей Digital Ocean

Рассмотрим более детально реализацию классом каждого из методов, предоставляемых интерфейсом ClusterResourceManager.

Метод allocate реализован с использованием следующих команд HTTP API Digital Ocean:

1. Создать новый droplet - POST /v2/droplets. Droplet в контексте Digital Ocean это абстракция над виртуальной машиной. При создании нового droplet выставляются следующие атрибуты:
    - name - пользовательское имя, генерируется системой автоматически с использованием глобального идентификатора формата UUID
    - region - в соответствии со значением из конструктора класса
    - size - тарифный план. Выбирается наиболее дешёвый из тех, которые отвечают запрошенным в аргументах метода ресурсам
    - image - в соответствии со значением из конструктора класса
    - ssh_keys - список из одного элемента в соответствии со значением из конструктора класса
2. Получить информацию о droplet по идентификатору - GET /v2/droplets/<id>. Droplet не обязательно доступен для использования сразу после создания соответствующей сущности, поэтому система некоторое время в соответствии с периодом updatePollPeriod регулярно проверяет состояние недавно созданного droplet, пока оно не перейдёт из new в active.
3. Включить droplet - POST /v2/droplets/<id>/actions с установлением JSON поля type в значение power_on. После того, как droplet будет готов к использованию, его необходимо включить, т.к. по-умолчанию он создаётся в выключенном состоянии. После отправки запроса на включение вновь происходит регулярная

отправка запросов на получение актуального состояния droplet до тех пор, пока он не включится.

Метод deallocate реализован с использованием следующих команд HTTP API Digital Ocean:

1. Удалить droplet - DELETE /v2/droplets/<id>.
2. Получить информацию о droplet по идентификатору - GET /v2/droplets/<id>. После получения ответа на запрос на удаление droplet, последний может существовать какое-то время, поэтому система здесь также некоторое время делает запрос на информацию о droplet до тех пор, пока сервер не ответит, что таковой не найден.

Метод executeCommand реализован с использованием протокола ssh без использования интерфейса Digital Ocean. Для работы с ssh в качестве клиента использовалась библиотека scalassh [34], предоставляющая объектный интерфейс для запуска команд по ssh. В виду особенностей протокола ssh, а также того факта, что Digital Ocean не предоставляет способа получить публичную часть ключа сервера при создании droplet, возникает требование использовать только приватную сеть для общения между системой и созданными виртуальными машинами в Digital Ocean. Это требования обусловено соображениями безопасности и отражено в документации с призывом обратить на это особое внимание.

Также в процессе тестирования выяснилось, что даже после включения droplet, соответствующая виртуальная машина не всегда сразу отвечала на запросы SSH. При этом система ожидает, что как каждый объект класса Node из возвращаемого методом allocate списка полностью

готов к запуску Mesos агента в момент возврата соответствующего значения методом allocate. В связи с этим в реализацию метода allocate для класса было дополнительно добавлено следующее: после создания droplet производится несколько попыток запустить на нём SSH команду до тех пор, пока это не сработает.

ЗАКЛЮЧЕНИЕ

В рамках данной работы была разработана система, полностью отвечающая поставленным требованиям. Таким образом, цель работы - разработка системы, позволяющей запускать задачи на вычислительном кластере Mesos и автоматически увеличивать или уменьшать размер этого кластера в зависимости от потребностей запускаемых приложений - достигнута. Система протестирована с использованием облачного провайдера Digital Ocean, а также предоставляет точки расширения, позволяющие реализовать поддержку других облачных провайдеров, а также различные стратегии принятия решений об изменении размеров кластера, исходя из потребностей конкретной предметной области. Эти точки расширения задокументированы для облегчения их использования разработчиками, не имеющими полного представления об архитектуре системы. В настоящее время для внедрения в проект требуется реализация конкретной стратегии изменения размеров кластера, для чего необходимо провести анализ нагрузки в виде задач, которые уже запускаются в проекте. Также в проекте используется другой облачный провайдер, который в рамках данной работы использовать было нецелесообразно ввиду дороговизны его услуг. В то же время, Digital Ocean предоставил некоторое количество предоплаченных услуг в рамках помощи студенческим проектам, за что автор работы премного ему благодарен. Это позволило тестировать частое создание и удаление платных виртуальных машин в процессе отладки системы, что повлекло бы значительные финансовые траты при работе с целевым для проекта облачным провайдером.

Помимо вышеупомянутых точек расширения, система имеет большой потенциал в своём развитии. Как видно из названия работы,

система работает исключительно с Mesos в качестве сервиса оркестрации, однако на рынке существуют другие системы подобного рода, некоторые из которых в последние года приобрели большую популярность, например, Kubernetes [35]. На самом деле, задача запуска процессов на кластере, которые необходимо выполнить как можно быстрее, даже за счёт выделения дополнительных платных ресурсов, может быть рассмотрена в отрыве Mesos, Kubernetes или другой конкретной технологии подобного вида. Теоретически, в разработанной системе можно абстрагировать слой, работающий с Mesos, выделив набор операций с определённой структурой и семантикой, который будет допускать конкретные реализации для различных сервисов оркестрации. Иначе говоря, спроектировать интерфейс и сделать всю работу с Mesos лишь одной из его реализаций. В этом случае получим систему более общего назначения, оставляющую за пользователем право выбора определённого “движка” или даже реализации поддержки собственного. Эта задача сама по себе заслуживает нетривиального анализа многих сложных технологий для проработки не менее сложного интерфейса, но теоретически выглядит реализуемой.

В качестве другой потенциальной точки расширения можно рассмотреть стратегию выбора очередной задачи для запуска на кластере. Если вспомнить рассмотренную архитектуру системы, то можно придать сомнению текущий подход, который WorkloadSupervisorActor применяет для принятия или отклонения очередного оффера от Mesos. А именно, он всегда ориентируется на запросы первой задачи в очереди, что не всегда оправдано. В некоторых случаях задача с большими запросами ресурсов будет причиной отказа от многих офферов, несмотря на то, что после неё в очереди находятся менее требовательные задачи, которые можно было бы запустить до последующего расширения кластера. Процесс принятия

решения в отношении поступившего набора офферов на основании всей очереди задач вполне может быть представлен в виде интерфейса, допускающего различные реализации, в том числе текущую. Поскольку в рамках данной работы не ставилось такого требования, в целях экономии времени такая возможность кастомизации не была реализована, но в целом имеет смысл.

Также в текущем состоянии система не поддерживает горизонтальное масштабирование самого управляющего процесса. Для увеличения пропускной способности в виде количества клиентов системы, обрабатываемой за единицу времени, сейчас можно только увеличить мощность машины, на которой запущен главный процесс - HTTP сервер. Иначе говоря, система позволяет только вертикальное масштабирование. Как правило, от систем подобного рода ожидается возможность горизонтального масштабирования не только для увеличения производительности, но и для обеспечения отказоустойчивости. Поэтому одним из направлением дальнейшего развития целесообразно рассмотреть реализацию такой функциональности.

Некоторые из рассмотренных аналогов предоставляют для пользователя графический веб-интерфейс, на случай, если запускать или просматривать задачи понадобится не программно, а оператором вручную. Это удобно, например, для тестирования или администрирования системы. Поскольку главный интерфейс системы реализован с использован протокола HTTP, то не представляет особого труда разработать соответствующее веб-приложение с помощью повсеместно используемых технологий HTML и JavaScript. Это также можно рассмотреть в качестве очередного вектора развития системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Powered By Mesos [Электронный ресурс] - URL: http://mesos.apache.org/documentation/latest/powered-by-mesos/ (дата обращения: 02.10.2019)
2. Apache Mesos - Building [Электронный ресурс] - URL: http://mesos.apache.org/documentation/latest/building/ (дата обращения: 04.10.2019)
3. Scheduler HTTP API [Электронный ресурс] - URL: http://mesos.apache.org/documentation/latest/scheduler-http-api/ (дата обращения: 06.10.2019)
4. Apache Mesos - Software Projects Built on Mesos [Электронный ресурс] - URL: http://mesos.apache.org/documentation/latest/frameworks/ (дата обращения: 06.04.2020)
5. vamp.io – Cloud-Native Release Orchestration [Электронный ресурс] - URL: https://vamp.io/ (дата обращения: 06.04.2020)
6. Marathon: A container orchestration platform for Mesos and DC/OS [Электронный ресурс] - URL: https://mesosphere.github.io/marathon/ (дата обращения: 07.11.2019)
7. Metronome: Cron Service for DC/OS [Электронный ресурс] - URL: https://dcos.github.io/metronome/ (дата обращения: 07.11.2019)
8. Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic [Электронный ресурс] - URL: https://www.elastic.co/elasticsearch/ (дата обращения: 06.04.2020)
9. douban/tfmesos: Tensorflow in Docker on Mesos #tfmesos #tensorflow #mesos [Электронный ресурс] - URL: https://github.com/douban/tfmesos (дата обращения: 06.04.2020)
10. TensorFlow [Электронный ресурс] - URL: https://www.tensorflow.org/ (дата обращения: 06.04.2020)
11. Artificial Intelligence Computing Leadership from NVIDIA [Электронный ресурс] - URL: https://www.nvidia.com/en-us/ (дата обращения: 06.04.2020)
12. Representational state transfer - Wikipedia [Электронный ресурс] - URL: https://en.wikipedia.org/wiki/Representational_state_transfer (дата обращения: 23.11.2019)

13. Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala [Электронный ресурс] - URL: https://akka.io/ (дата обращения: 01.12.2019)

14. SQLite Home Page [Электронный ресурс] - URL: https://www.sqlite.org/index.html (дата обращения: 06.12.2019)

15. OpenAPI Specification - Version 3.0.2 | Swagger [Электронный ресурс] - URL: https://swagger.io/specification/ (дата обращения: 30.03.2020)

16. JSON [Электронный ресурс] - URL: https://www.json.org/ (дата обращения: 09.01.2020)

17. Модель акторов — Википедия [Электронный ресурс] - URL: https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%BE%D0%B2 (дата обращения: 23.01.2020)

18. Реактивные приложения на Java с Akka | DOU [Электронный ресурс] - URL: https://dou.ua/lenta/articles/Reactive-Applications-in-Java-with-Akka/ (дата обращения: 01.04.2020)

19. Travis CI - Test and Deploy Your Code with Confidence [Электронный ресурс] - URL: https://travis-ci.org/ (дата обращения: 06.04.2020)

20. The world's leading software development platform · GitHub [Электронный ресурс] - URL: https://github.com/ (дата обращения: 10.09.2019)

21. RhodeCode › Version Control Systems Popularity in 2016 [Электронный ресурс] - URL: https://rhodecode.com/insights/version-control-systems-2016

22. sbt - The interactive build tool [Электронный ресурс] - URL: https://www.scala-sbt.org/ (дата обращения: 04.02.2020)

23. Scoverage [Электронный ресурс] - URL: http://scoverage.org/ (дата обращения: 07.03.2020)

24. scoverage/sbt-scoverage: sbt plugin for scoverage [Электронный ресурс] - URL: https://github.com/scoverage/sbt-scoverage (дата обращения: 07.03.2020)

25. Scalafmt · Code formatter for Scala [Электронный ресурс] - URL: https://scalameta.org/scalafmt/ (дата обращения: 07.03.2020)

26. scalameta/sbt-scalafmt: sbt plugin for Scalafmt [Электронный ресурс] - URL: https://github.com/scalameta/sbt-scalafmt (дата обращения: 07.03.2020)

27. Scalafix · Refactoring and linting tool for Scala [Электронный ресурс] - URL: https://scalacenter.github.io/scalafix/ (дата обращения: 07.03.2020)
28. scalacenter/sbt-scalafix: sbt plugin for Scalafix [Электронный ресурс] - URL: https://github.com/scalacenter/sbt-scalafix (дата обращения: 07.03.2020)
29. Coveralls - Test Coverage History & Statistics [Электронный ресурс] - URL: https://coveralls.io/ (дата обращения: 07.03.2020)
30. scoverage/sbt-coveralls: Sbt plugin for uploading Scala code coverage to coveralls [Электронный ресурс] - URL: https://github.com/scoverage/sbt-coveralls (дата обращения: 07.03.2020)
31. DigitalOcean – The developer cloud [Электронный ресурс] - URL: https://www.digitalocean.com/ (дата обращения: 02.02.2020)
32. DigitalOcean API [Электронный ресурс] - URL: https://developers.digitalocean.com/documentation/v2/ (дата обращения: 22.02.2020)
33. jeevatkm/digitalocean-api-java: DigitalOcean API Client in Java [Электронный ресурс] - URL: https://github.com/jeevatkm/digitalocean-api-java (дата обращения: 22.02.2020)
34. sirthias/scala-ssh: Remote shell access via SSH for your Scala applications [Электронный ресурс] - URL: https://github.com/sirthias/scala-ssh (дата обращения: 27.12.2019)
35. Production-Grade Container Orchestration - Kubernetes [Электронный ресурс] - URL: https://kubernetes.io/ (дата обращения: 30.05.2020)

ПРИЛОЖЕНИЕ А
OpenAPI спецификация HTTP интерфейса системы

```yaml
openapi: "3.0.2"
info:
  version: "0.0.3"
  title: "Miknik Scheduler HTTP API"
  contact:
    email: "skripmaxand@gmail.com"
paths:
  /jobs:
    post:
      summary: "Create a Job"
      requestBody:
        description: "Job metadata"
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/JobCreate"
      responses:
        201:
          $ref: "#/components/responses/Job"
        400:
          description: "Bad request"
        500:
          description: "Internal server error"
    get:
      summary: "List Jobs"
      responses:
        200:
          description: "Jobs"
          content:
            application/json:
              schema:
                allOf:
                  - $ref: "#/components/schemas/ListResponse"
                  - type: object
```

```
              properties:
                items:
                  type: array
                  items:
                    $ref: "#/components/schemas/Job"
      500:
        description: "Internal server error"
/jobs/{id}:
  delete:
    summary: "Delete a Job"
    parameters:
      - $ref: "#/components/parameters/JobId"
    responses:
      200:
        description: "Deleted"
      404:
        description: "Not found"
      500:
        description: "Internal server error"
/jobs/{id}/cancel:
  post:
    summary: "Cancel a Job"
    parameters:
      - $ref: "#/components/parameters/JobId"
    responses:
      202:
        description: "Job cancel process started"
      404:
        description: "Not found"
      500:
        description: "Internal server error"
/resources:
  get:
    summary: "List available cluster resources"
    responses:
      200:
        description: "Current cluster resources"
      500:
        description: "Internal server error"
```

```
  /healthcheck:
    get:
      summary: "Check if server is healthy"
      responses:
        200:
          description: "Server healthy"
        500:
          description: "Server is not healthy"
components:
  schemas:
    ListResponse:
      type: "object"
      properties:
        count:
          type: integer
    JobCreate:
      type: "object"
      properties:
        id:
          type: "string"
        resources:
          type: "object"
          properties:
            mem:
              type: "integer"
            cpus:
              type: "number"
            disk:
              type: "integer"
            dockerImage:
              type: "string"
            cmd:
              type: "array"
            items:
              type: "string"
            env:
              type: "object"
          additionalProperties:
            type: "string"
```

```
      required:
        - id
        - resources
        - dockerImage
Job:
  type: "object"
  allOf:
    - $ref: "#/components/schemas/JobCreate"
    - properties:
        status:
          $ref: "#/components/schemas/JobStatus"
        error:
          type: "string"
        created:
          type: "string"
          format: "date-time"
        updated:
          type: "string"
          format: "date-time"
        completed:
          type: "string"
          format: "date-time"
      required:
        - status
        - created
        - updated
JobStatus:
  type: "string"
  enum:
    - pending
    - running
    - completed
    - failed
    - canceled
```

```
parameters:
  JobId:
    in: "path"
    name: "id"
    description: "Job id"
    required: true
    schema:
      type: "string"
responses:
  Job:
    description: "Job response"
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Job"
```

АНТИПЛАГИАТ
ТВОРИТЕ СОБСТВЕННЫМ УМОМ

Отчет о проверке на заимствования №1

Автор: flayerman@yandex.ru / ID: 4464154
Проверяющий: (flayerman@yandex.ru / ID: 4464154)

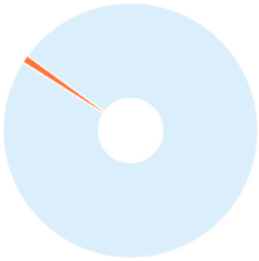
Отчет предоставлен сервисом «Антиплагиат»- http://users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 8
Начало загрузки: 13.06.2020 13:23:35
Длительность загрузки: 00:00:03
Имя исходного файла: Скрипник_ВКР.pdf
Название документа: Скрипник_ВКР
Размер текста: 1 кБ
Символов в тексте: 72402
Слов в тексте: 8866
Число предложений: 414

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Последний готовый отчет (ред.)
Начало проверки: 13.06.2020 13:23:38
Длительность проверки: 00:00:04
Комментарии: не указано
Модули поиска: Модуль поиска Интернет



ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
1,36%	0%	0%	98,64%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Доля в тексте	Источник	Ссылка	Актуален на	Модуль поиска	Блоков в отчете	Блоков в тексте
[01]	0,06%	0,34%	Разработка мобильного при...	http://library.eltech.ru	19 Сен 2019	Модуль поиска Интернет	2	4
[02]	0,25%	0,25%	Дипломная работа: Автомат...	https://referatbank.ru	07 Апр 2020	Модуль поиска Интернет	1	1
[03]	0,21%	0,21%	29122-g40.zip (2/2)	https://3gpp.org	17 Дек 2019	Модуль поиска Интернет	2	2

Еще источников: 16
Еще заимствований: 0,85%