ТИТУЛЬНЫЙ ЛИСТ

РЕФЕРАТ

Выпускная квалификационная работа бакалавра состоит из 46 страниц, 9 рисунков, 3 таблиц, 18 использованных источников, 1 приложения.

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ, РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ, МОНИТОРИНГ

В выпускной квалификационной работе бакалавра была спроектирована платформа анализа и оценки производительности распределённых вычислительных систем с целью создания инструментария для более полного и глубокого анализа подобных систем, а также разработан прототип платформы, собирающий информацию о взаимодействии вычислительных узлов посредством gRPC фреймворка, который был протестирован на разработанном тестовом приложении.

Данная платформа уже позволяет значительно ускорить вхождение в существующий проект без документации новых работников, таких как архитектор.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей выпускной квалификационной работе применяют следующие термины с соответствующими определениями:

Архитектурный ландшафт	граф, описывающий взаимодействие вычислительных узлов или сервисов распределённой вычислительной системы
Контейнер	приложение, изолированное на уровне ядра операционной системы с помощью средств контейнеризации
Метрика мониторинга	численное представление информации, измеренной через промежутки времени, которая исходит от использования ресурсов и поведения системы
Микросервисы	это достаточно небольшие, автономные, совместно работающие сервисы
Надёжность	совокупность свойств, характеризующая способность программного средства сохранять заданный уровень пригодности в заданных условиях в течение заданного интервала времени [1]
Наработка	продолжительность или объём работы объекта
Приложение	программа, предназначенная для выполнения определённых задач и рассчитанная на непосредственное взаимодействие с пользователем
Производительность	это характеристика вычислительной мощности системы, определяющая количество вычислительной

	работы, выполняемой системой за единицу времени [2]
Производственная среда	виртуальная среда, в которой находится работающая система, с которой взаимодействует пользователь
Распределённая вычислительная система	это набор автономных вычислительных элементов, которые представляется своим пользователям как единая целостная система [3]
Экономичность	свойство системы сохранять установленные экономические показатели при заданных требованиях к надёжности и эффективности

ВВЕДЕНИЕ

В настоящее время, трудно представить современный мир без распределённых вычислительных систем. Они приобрели весьма широкое распространение в отрасли. Так, например, когда мы хотим отправить кому-то сообщение в социальной сети, наше послание, скорее всего, будет обработано распределённой вычислительной системой. Когда заказываем еду через приложение, наш запрос, скорее всего, будет обработан распределённой вычислительной системой. Но с чего всё начиналось?

Начиная с 1945 года, начали появляться первые компьютеры. Они были большими и очень дорогими и работали независимо друг от друга, т.к. не было способов взаимодействия между ними. Но примерно с середины восьмидесятых годов прошлого века все начало меняться после появления мощных микропроцессоров, сначала 8-битных, а затем 16-ти, 32-ух и 64-битных, и высокоскоростных компьютерных сетей, LANs и WANs. [3]

Сначала, примерно с середины восьмидесятых годов, были системы, состоящие из малопроизводительных терминалов и мэйнфреймов, более мощных вычислительных машин, к которым подключались данные терминалы. Затем начали писать программные системы, которые способны работать на нескольких компьютерах одновременно, поскольку было экономически выгоднее использовать систему, состоящую из дешёвых компьютеров, которая обладает более высоким соотношением производительности/цена, по сравнению с системой, состоящей из одного высокопроизводительного компьютера.[4] И после роста интернета, и появления в начале нулевых крупных центров обработки данных, которые состояли из тысяч компьютеров, распределённые вычислительные системы получили широкую распространённость, которую они имеют и сегодня.

Сейчас появились новые технологии, позволяющие более эффективно разрабатывать, разворачивать и поддерживать эти системы. Самыми значимыми из них являются технологии контейнеризации и оркестрации. Благодаря контейнеризации мы теперь можем разрабатывать отдельные

сервисы не задумываясь сильно об окружении конечного пользователя, так как они позволяют упаковывать приложение в единый образ, который можно будет развернуть на любом вычислительном узле, на котором есть средства управления контейнерами. [5] Самым ярким примером такой технологии является технология Docker. По данным datadog уже в январе 2018 года 35% компаний активно использовали Docker контейнеры, и количество таких кампаний только увеличивалось. [6] Но за всей простотой их развертывания скрывается сложность их обслуживания при увеличении количества контейнеров. Поэтому для уменьшения сложности обслуживания разработали так называемые оркестраторы, которые позволяют контролировать, что все необходимые контейнеры и их реплики работают в штатном состоянии, и в случае необходимости заново поднимать и разворачивать упавшие контейнеры или заменять устаревшие на новые прямо на лету, то есть не останавливая всю систему, чтобы заменить один или нескольно устаревших контейнеров, благодаря чему упростились процессы разработки отказоустойчивых и масштабируемых систем.

Это стало одной из причин, благодаря которой сервис-ориентированная архитектура (SOA) эволюционировала в “Гранулированный SOA” или, говоря более современным языком, в микросервисную архитектуру (MSA), которая стала весьма популярной в наше время. Так в 2016ом году International Data Corparation предсказывала, что к концу 2021 года 80% облачного ПО будет разрабатываться с использованием микросервисов[7], и судя по имеющейся статистике [8], показанной на рисунке 1, данное предсказание оказалось весьма близким к реальной ситуации, поскольку на момент 2021 года 85% крупных компаний (от 5000 сотрудников) используют микросервисы, 84% менее крупных компаний (от 3000 до 4999 сотрудников) и 75% небольших (от 1000 до 2999 сотрудников).

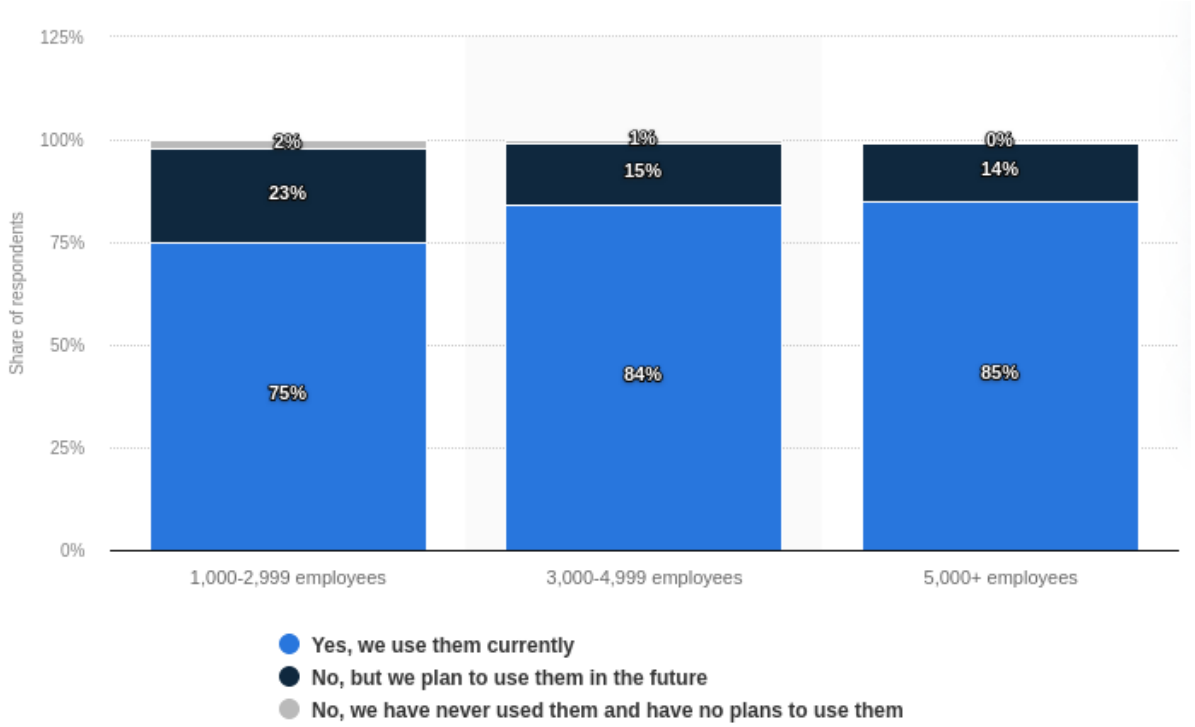

Рисунок 1 - Статистика использования микросервисов [8]

MSA - это принцип разработки программного обеспечения, когда мы разбиваем наше приложение на микросервисы, то есть на “достаточно небольшие”, слабо связанные и легко изменяемые модули, для того, чтобы можно было легко масштабировать в рамках распределённых систем и при необходимости легко их заменять или использовать заново.

И если остановиться на этом моменте, то может показаться, что распределённые системы это “лучшее изобретение человечества”. Но, как и многие инженерные решения, распределенные вычислительные системы имеют свои недостатки, в частности связанные с их разработкой, отладкой и сопровождением.

Во-первых, очевидно, что вся диагностическая информация распределённой системы может находиться на разных вычислительных узлах, что затрудняет процесс мониторинга, локализации ошибки и определения причины её возникновения. Особенно этот процесс может усложниться, если произошла цепочка связанных между собой программных вызовов, которые повлекли за собой ошибку в работе системы или её отказ.

Во-вторых, количество вычислительных узлов, а также их конфигурация формируется динамически. Это также усложняет поиск ошибок и разработку архитектуры в целом. Из-за динамической конфигурации системы затрудняется поиск “тонких мест”, которые могут повлечь критические ошибки, или просто достаточно сильно снижать эффективность системы, оставаясь незамеченными.

Данные проблемы кажутся не слишком критичными в относительно небольших приложениях, но ситуация приобретает ужасающие масштабы, когда мы смотрим на современные микросервисные приложения, когда у нас тысячи вычислительных узлов, на которых автоматически разворачиваются десятки и сотни контейнеров, заменяются упавшие вследствие ошибки реплики и масштабируется их количество, для обеспечения постоянной доступности и максимальной производительности системы в целом. В данной ситуации приходится следить за десятками или сотнями тысяч контейнирезированных приложений, что может быть весьма затруднительно.

Существуют различные платформы, для сбора метрик с распределённых вычислительных систем, но они не дают все желаемые функции, продемонстрированные в таблице 1.

Таблица 1 - Сравнение функционала

Функционал	ZABBIX	New Relic
сбор метрик	+	+
задание сигнализации	+	+
поиск аномалий	–	+
обнаружение зависимостей между цепочками вызовов	–	–

Таким образом, мы пришли к необходимости проектирования платформы, позволяющей нам оценивать эффективность распределённых вычислительных систем, посредством сбора и анализа метрик,

характеризующих эти системы, с целью упрощения разработки, тестирования и сопровождения данных систем.

Для выполнения данной задачи было сформулировано шесть подзадач, которые позволят разработать теоретическое и практическое решение поставленной задачи. А именно:

1) определить требования к платформе;
2) определить метрики, необходимые для анализа и отладки распределённых вычислительных систем, и методы их анализа;
3) спроектировать архитектуры платформы;
4) разработать:
    a) минимальные базовые компоненты платформы;
    b) тестовый программный продукт на базе микросервисной архитектуры;
5) протестировать прототип платформы на тестовом программном продукте.

В конечном результате был разработан прототип платформы, который собирает информацию о взаимодействии сервисов между собой с помощью gRPC фреймворка, сохраняет собранную информацию в Elasticsearch и визуализирует с помощью Grafana и инструментарий для интеграции платформы в существующее приложение.

Также прототип был протестирован с тестовым приложением на предмет его работоспособности и влияния на эффективность платформы. Результаты показали, что самый долгий вызов увеличил время работы с 8,2 милисекунд до 23,2, что является увеличением работы почти в 4 разы, но поскольку тестовое приложение было запущено на одном компьютере, сетевые задержки, которые, обычно, составляют значительное время запроса, были минимальны, что так значительно повлияло на результаты тестирования. Однако, даже на данный момент, платформа ускоряет процесс введения в курс дела архитектора в проект без заранее построенной архитектуры приложения.

1 ПОСТАНОВКА ЗАДАЧИ И ТЕОРЕТИЧЕСКИЕ ПРЕДПОСЫЛКИ

Итак, перед нами стоит задача разработки прототипа платформы, собирающей информацию о синхронном взаимодействии сервисов между собой.

Также необходимо разработать тестовый продукт, на котором будет апробирован наш прототип. Поскольку микросервисы становятся всё более часто используемыми, а Docker стал стандартом в IT-индустрии, то будем использовать данные технологии при развёртывании тестового продукта.

1.1 Пользовательские сценарии

Для того, чтобы определить требования к нашей платформе, были проработаны пользовательских сценариев.

Для начала определим основные роли целевых пользователей:

- архитектор - работник, занимающийся планированием и разработкой высокоуровневой архитектуры приложения;
- разработчик - работник, занимающийся разработкой конкретных частей приложения;
- тестировщик - работник, занимающийся проверкой качественности и надёжности приложения;
- DevOps/SRE-инженер - работник, занимающийся поддержкой CI/CD процессов и общедоступности приложения;
- системный администратор - работник, занимающийся аппаратной поддержкой системы, на которой работает приложение.

1.1.1 Построение архитектурного ландшафта

Участники: авторизованный пользователь, который имеет одну из ролей: архитектор, программист или тестировщик.

Предусловие: существует система, которая работает сейчас в производственной среде.

Постусловие: полный граф сервисов и их взаимодействие показано пользователю на веб-странице.

Сценарий:

1) пользователь заходит на вкладку портала платформы "MDS-platform";
2) система предлагает пользователю выбор среды, на которой необходимо построить архитектурный ландшафт;
3) пользователь выбирает необходимую ему среду;
4) пользователь нажимает кнопку: "Построить".

1.1.2 Сравнение состояний системы

Участники: авторизованный пользователь, который имеет одну из ролей: архитектор, программист, тестировщик, DevOps/SRE - инженер.

Предусловие: существует система, за которой происходит наблюдение, и в компоненты которой внесли изменения.

Постусловие: граф сервисов, на котором выделены значительные изменения.

Сценарий:

1) пользователь заходит на сайт портала платформы "MDS-platform";
2) система предлагает пользователю выбор среды, состояния которой надо сравнить;
3) пользователь выбирает среду;
4) пользователь нажимает кнопку “сравнить состояния”;
5) пользователь вводит два диапазона времени, между которыми произошли изменения;
6) пользователь нажимает кнопку “построить”.

Альтернативный сценарии:

5) пользователь выбирает версии ключевых сервисов, при работе которых необходимо сравнить состояния системы.
5) пользователь выбирает теги, которыми отмечаются состояния системы.

1.1.3 Задание сигнализации

Участники: авторизованный пользователь, который имеет роль: программист, системный инженер/администратор, DevOps/SRE - инженер, тестировщик.

Предусловие: существует система, которая работает в тестовой, промежуточной или производственной среде.

Постусловие: настроенное правило, в случае срабатывания которого, платформа отправляет уведомление одному или нескольким пользователям.

Сценарий:

1) пользователь заходит на сайт портала платформы "MDS-platform";
2) система предлагает пользователю выбор среды, состояния которой надо сравнить;
3) пользователь выбирает среду;
4) пользователь нажимает кнопку “установить сигнализацию”;
5) выбирает список сервисов (или всю систему), на который будет распространятся заданное правило;
6) пользователь смотрит доступные для использования метрики;
7) пользователь задаёт правило, используя доступные метрики, с помощью языка задачи правил платформы;
8) пользователь задаёт частоту проверки данного правила
9) пользователь задаёт список контактов (номера телефонов, названия почт, имена пользователей в мессенджерах), на которые будет выслано уведомление.

1.1.4 Отслеживание пути возникновения ошибки

Участники: авторизованный пользователь, который имеет одну из ролей: разработчик, тестировщик, DevOps/SRE - инженер.

Предусловие: построенный архитектурный ландшафт системы, работающей в тестовой, промежуточной и производственной среде.

Постусловие: выделенный путь сервисов и линий взаимодействия сервисов, вследствие которого произошла ошибка.

Сценарий:

1) пользователь выбирает и нажимает на интересующий его сервис;
2) пользователь выбирает подраздел “ошибки” в появившемся окне;
3) пользователь выбирает конкретную ошибку;
4) пользователь нажимает кнопку отследить.

1.1.5 Обзор состояния системы

Участники: авторизованный пользователь, который имеет одну из ролей: разработчик, тестировщик, DevOps/SRE - инженер.

Предусловие: существует система, которая работает сейчас в производственной среде.

Постусловие: графики и диаграммы метрик за определённый временной диапазон.

Сценарий:

1) пользователь заходит на сайт портала платформы "MDS-platform";
2) система предлагает пользователю выбор среды, состояния которой надо сравнить;
3) пользователь выбирает среду;
4) пользователь нажимает кнопку “обзор”;
5) платформа предлагает пользователю выбрать диапазон, за который визуализировать собранные метрики;
6) пользователь выбирает необходимый ему диапазон (или оставляет диапазон по умолчанию);
7) пользователь нажимает кнопку построить.

1.1.6 Авторизация в систему

Участники: любой неавторизованный.

Предусловие: пользователь зарегистрирован на платформе.

Постусловие: пользователь получил доступ к персональным возможностям.

Сценарий:

1) пользователь заходит на сайт портала платформы “MDS-platform”;

2) система предлагает пользователю ввести логин и пароль и сообщает, что есть 3 попытки;
3) пользователь вводит правильный логин и пароль.

Альтернативные сценарии:

4) система отправляет пользователю код авторизации;
5) пользователь вводит полученный код.
3) пользователь вводит неправильный логин или пароль;
4) система предлагает ввести заново логин и пароль и сообщает оставшееся число попыток;
4) попыток не осталось, и система предлагает попробовать авторизоваться позже.

1.1.7 Интеграция в существующий продукт

Участники: разработчик, DevOps/SRE - инженер.

Предусловие: существующая система.

Постусловие: существующая система, со встроенными в неё сборщиками метрик.

Сценарий:

1) пользователь скачивает с платформы библиотеку, содержащую все необходимые компоненты;
2) пользователь заменяет необходимые части сборщика на их аналоги;
3) запускает заново сборку и развёртывание системы.

Таким образом можно определить несколько требований к нашей платформе:

- возможность собирать метрики с распределённых сервисов и систем;
- возможность построения архитектурного ландшафта;
- отслеживание зависимостей между компонентами распределённой вычислительной системы;
- фиксирование состояния системы за определённые промежутки времени, или по определённым признакам (например, версии

конкретных сервисов);

- установка правил, при срабатывании которых происходит оповещение конкретных пользователей;
- возможность написания расширений сторонними лицами, для увеличения возможностей платформы.

На данном этапе может показаться, что наша платформа будет аналогом Zabbix и/или New Relic, но главной особенностью нашей платформы будет возможность отслеживать, как происходят конкретные взаимодействия между сервисами и как сильно влияют изменения в сервисах приложения на производительность приложения в целом, но в целом можно согласиться, что наша платформа занимается в мониторингом в некотором роде.

1.2 Мониторинг распределённых вычислительных систем

Мониторинг это совокупность действий, включающая в себя сбор, интерпретацию и отображение информации о параллельно работающих процессах. [9] Можно выделить 6 практик мониторинга, упоминаемых в серой литературе:

- управление логами;
- отслеживание ошибок/исключений;
- API проверки работоспособности;
- логирование развёртывания;
- аудиторский след;
- распределённое отслеживание. [10]

Каждая практика состоит из нескольких действий. Так, например, управление логами это создание, сбор, централизация, анализ, передача, архивирование логов, поставляемые сервисами на серверах, сетевыми устройствами и другими компонентами.

Поэтому, основываясь на сформулированных выше требованиях можно предположить, что наша платформа будет представлять из себя платформу, использующую разные практики такие как: распределённое отслеживание,

аудиторский след, отслеживание ошибок.

1.3 Метрики

Было сформулировано то, что требуется получить от платформы. Теперь сформулируем то, каким образом мы будем оценивать системы, за которым происходит мониторинг. Лучший способ оценивания конкретных частей системы это с помощью метрик, поскольку благодаря им возможно сравнивать различные состояния системы и определять лучшее из них.

Так, например, очевидно, что можно предположить, что лучшая система, в плане производительности, будет та, которая обладает наибольшим показателем производительности, например, таким как числом обработанных запросов в единицу времени.

1.3.1 Что такое метрика?

Метрики - это численное представление данных, измеренных в течение времени, которые описывают, как правило, использование ресурсов или поведение системы.

Существует большое число различных метрик, так как есть общие метрики, которые можно применять ко всем или большинству систем, и специфичные метрики, которые описывают какие-нибудь частные особенности используемого языка, например, дополнительный объём памяти, используемый сборщиком мусора, или же описывающие поведение бизнес модели приложения, так называемые SLA-метрики, например, время сеанса пользователя.

Можно выделить 12 популярных групп метрик, упоминаемых в серой литературе, а конкретно это:

- использование ресурсов;
- балансировка нагрузки;
- доступность;
- подключений к базе данных;
- треды;

- ошибки и исключения;
- успешные запросы;
- latency базы данных;
- открытые файлы;
- состояние сервиса;
- специфичные метрики языка.

Поскольку это достаточно общие группы метрик, то можно конкретизировать их до определённых параметров, чтобы можно было разрабатывать и использовать в нашей платформе. Так, например, в качестве определённых показателей использования ресурсов можно использовать нагрузку на CPU и использование оперативной памяти.

Но для упрощения работы с данными группами метрик, разобьём их на три класса метрик, которые описывают разные свойства эксплуатационные свойства системы. Так можно выделить свойства:

- экономичность;
- надёжность;
- производительность.

Итак, поясним выше описанные группы метрик, чтобы можно было легко понять, что за что отвечает. Метрики, описывающие экономичность, позволяют охарактеризовать то, как много ресурсов наблюдаемая система использует в своей работе. С помощью данных параметров можно будет обнаружить чрезмерное использование ресурсов, которые можно было бы использовать более эффективно. Это может быть очень важно, например, в тех случаях, когда приложение работает на арендуемых серверах, где происходит тариф рассчитывается не из реального времени использования, а из объёма используемых ресурсов. В качестве примера, сюда можно отнести метрики использования ресурсов, подключений к базе данных и треды.

Дальше - надёжность. Надёжность характеризует способность конкретных сервисов сохранять свою работоспособность в течение времени. Данный показатель особо важен в распределённых вычислительных

системах, в частности в микросервисных архитектурах поскольку данные системы рассчитаны на то, что в них может и произойдёт отказ, но, насколько серьёзный или объёмный масштаб последствий будет, помогут описать метрики надёжности. Сюда можно отнести такие метрики как доступность, количество ошибок и исключений, время доступности, время, необходимое для восстановления работоспособного состояния и так далее.

Теперь разберём производительность. Производительность, как характеристика, позволяет определить, как много операций может совершать наша система в единицу времени, или сколько времени уходит на выполнение одной операции или запроса. Это позволяет определить узкие места нашей системы, где, например, не хватает ресурсов для достаточного масштабирования, чтобы удерживать показатели производительности на необходимом уровне. К таким показателям можно отнести частоту запросов, время выполнения запроса на стороне клиента, на стороне сервиса и его время передачи по сети.

Хотелось бы уделить особое внимание метрикам надёжности, поскольку существует несколько свойств, которые характеризуют данное качество, а именно:

- Завершенность программного средства - совокупность свойств программного средства, характеризующая частоту отказов, обусловленных дефектами программного средства.

- Безотказность - свойство объекта непрерывно сохранять способность выполнять требуемые функции в течение некоторого времени или наработки в заданных режимах и условиях применения.

- Отказоустойчивость программного средства - совокупность свойств программного средства, характеризующая его способность поддерживать необходимый уровень пригодности при проявлении дефектов программного средства или нарушении установленных интерфейсов.

- Восстанавливаемость программного средства - совокупность свойств программного средства, характеризующая возможность осуществления,

трудоемкость и продолжительность действий по восстановлению им своего уровня пригодности, а также непосредственно подвергшихся воздействию данных, в случае отказа. [1]

Из данного списка хотелось бы выделить два свойства: безотказность и восстанавливаемость - поскольку метрики данных свойств позволяют достаточно подробно описать поведение системы с точки зрения надёжности.

1.3.2 Безотказность и восстанавливаемость

Итак, почему было выделено два данных свойства? Безотказность и восстанавливаемость это два свойства, которые позволяют построить граф состояния для конкретного сервиса, а также определить коэффициент готовности, то есть вероятность того, что сервис окажется работоспособным в определённый момент времени. [11] Для начала определим показатели, которыми описываются данные свойства.

1.3.2.1 Показатели безотказности

Для безотказности используют понятие наработки, а также такие показатели как:

Вероятность безотказной работы $P(t)$ - вероятность того, что в пределах заданной наработки отказ объекта не возникнет.

Оценка данного показателя считается с помощью формулы (1).

$$\hat{P}(t) = \frac{N_0 - n(t)}{N_0}, \quad (1)$$

где $n(t)$- число отказов к моменту времени t,

N_0 - количество наблюдаемых объектов на чальный момент времени $t = 0$.

Средняя наработка до отказа $T_{ср}$ - математическое ожидание наработки объекта до отказа.

Оценка данной величины считается по формуле (2).

$$\hat{T}_{cp} = \frac{1}{N_0}\sum_{i=1}^{N_0} t_{pi}, \quad (2)$$

где t_{pi} - время наработки i-го объекта до отказа из N_0 отказавших объектов.

Интенсивность отказов $\lambda(t)$ - условная плотность вероятности возникновения отказа объекта, определяемая при условии, что до рассматриваемого момента времени отказ не возник. Или выражаясь более математически:

$$\lambda(t) = \frac{q(t)}{P(t)}, \quad (3)$$

где $q(t) = \frac{dQ(t)}{dt}$ - плотность вероятности возникновения отказа

$$Q(t) = 1 - P(t) \quad (4).$$

Решив уравнение (3) относительно $P(t)$, можно получить связь между вероятностью безотказной работы $P(t)$ и $\lambda(t)$:

$$P(t) = exp\left(-\int_0^t \lambda(t)\, dt\right) \quad (5)$$

Статистическая оценка интенсивности отказов может быть получена с помощью формулы (6).

$$\hat{\lambda}(t) = \frac{N(t) - N(t+\Delta t)}{N(t)\cdot \Delta t}, \quad (6)$$

где $N(t)$, $N(t + \Delta t)$ - число объектов работоспособных к моменту времени t и $t + \Delta t$ соответственно.

Параметр потока отказов $\omega(t)$ - предел отношения вероятности возникновения отказа восстанавливаемого объекта за достаточно малый интервал времени к длительности этого интервала, стремящейся к нулю.

1.3.2.2 Показатели восстанавливаемости

Показатели восстанавливаемости аналогичны показателям безотказной работы. Они используют понятие время до восстановления — время от момента отказа до восстановления работоспособного состояния объекта и включают в себя:

Вероятность восстановления $P_В(t_{зад})$ - вероятность того, что время до восстановления работоспособного состояния объекта не превысит заданное значение.

Среднее время до восстановления $T_{дв}$ - математическое ожидание времени до восстановления.

Интенсивность восстановления $\mu(t)$- условная плотность вероятности восстановления работоспособного состояния объекта, определённая для рассматриваемого момента времени при условии, что до этого момента восстановление не было завершено.

1.3.2.3 Граф состояний системы и коэффициент готовности

Итак, имея в распоряжении такие показатели как интенсивность отказов $\lambda(t)$ и интенсивность восстановления $\mu(t)$, можно составить граф состояний, показанные на рисунке 1.1. [12]

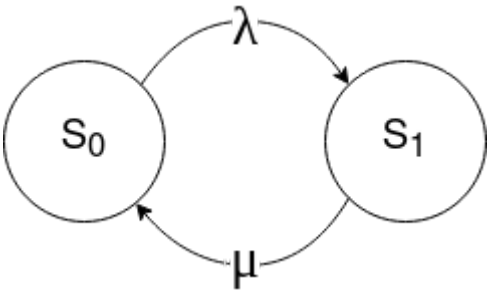

Рисунок 1.1 - Граф состояний системы

Здесь S_0 - работоспособное состояние сервиса, а S_1 - неработоспособное. Сервис переходит из состояния S_0 в состояние S_1 с интенсивностью отказов λ и обратно из S_1 в S_0 с интенсивностью восстановления μ.

Поскольку нахождение в одном из двух состояний это случайное событие, тогда мы можем определить с какой вероятность P_0 сервис находится в работоспособном состоянии S_0 и с какой вероятность P_1 сервис находится в неработоспособном состоянии S_1. Для решения данной задачи опишем данный граф системой дифференциальных уравнений Колмогорова:

$$\begin{cases} \frac{d}{dt}P_0(t) = -\lambda \cdot P_0(t) + \mu \cdot P_1(t) \\ \frac{d}{dt}P_1(t) = \lambda \cdot P_0(t) - \mu \cdot P_1(t) \\ P_0(t) + P_1(t) = 1 \end{cases} \quad (6)$$

Решение данной системы уравнений для P_0 имеет вид:

$$P_o(t) = \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} \cdot exp(-(\lambda + \mu) \cdot t) \quad (7)$$

Итак, мы получили вероятность того, что наш сервис находится в работоспособном состоянии, или, иначе говоря, коэффициент готовности $К_г(t)$ - вероятность того, что объект окажется в работоспособном состоянии в данный момент времени. [13]

1.3.3 Итог по метрикам

Итак, было показано, какие существуют метрики и предложили некоторые необычные метрики из теории надёжности, которые редко можно встретить в разных готовых решениях. Но все выше перечисленные метрики использовать не стоит, так как пользователю они могут быть не нужны и не очень понятны, но, если понадобятся какие-то специфичные метрики, всегда будет возможность расширить функционал платформы для сбора, обработки и анализа метрики, не предоставленной по умолчанию в платформе.

Таким образом, можно перечислить базовые метрики, которые будут предоставлены пользователям, как минимально необходимые, и это:

- частота запросов к конкретному сервису конкретных запросов или групп запросов;
- время запроса, потраченное на стороне клиента, на стороне сервиса и в процессе передачи по сети;
- количество ошибок во время запросов и их частота;
- количество и частота отказов конкретных сервисов;
- время недоступности этих сервисов и их восстановления;
- интенсивности отказов и восстановления сервисов;
- коэффициент готовности сервиса.

1.4 Виды взаимодействий в распределённых вычислительных системах

Теперь стоит рассказать о том, как могут передаваться данные внутри распределённых систем. Это важная тема, поскольку достаточно много времени может тратиться на ожидание выполнения передачи и/или получения данных. Все взаимодействия можно разделить по признаку использования хранилища, показанные на рисунке 1.2. [14]

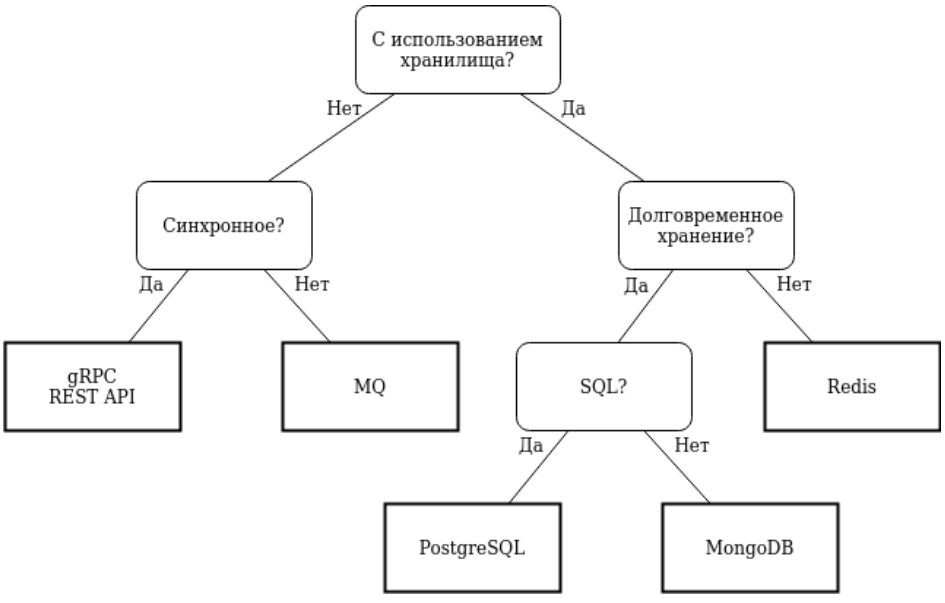

Рисунок 1.2 - Виды взаимодействий

1.4.1 Без использования хранилища

Взаимодействие без использования хранилища это прямое взаимодействие между сервисами как на разных вычислительных узлах, так и на одном. Данный вид взаимодействия зачастую нужен, чтобы один сервис отправил запрос на выполнение определённых действий другому сервису, и, возможно, получил ответ.

В некоторых случаях, сервису инициатору не всегда необходим результат своего запроса в момент вызова, поэтому взаимодействие без использования хранилища можно разделить на синхронное и асинхронное.

1.4.1.1 Синхронное

Ярким примером синхронного взаимодействия можно назвать Representational State Transfer (REST) и Remote Procedure Call (RPC). Так, например, REST API удобно использовать, когда сторонние приложения должны работать с нашим приложением, как при взаимодействии сайтов и веб-приложений с сервером. Если необходимо, чтобы наш сервис мог делать запросы к другому нашему сервису, то также возможно использовать REST API, но написание такого кода будет значительно сложнее. Вместо этого будет значительно проще использовать RPC. Например, gRPC, который работает на HTTP/2, благодаря чему обмен данными в 7-10 раз быстрее, чем REST, позволяет описать RPC - методы и сообщения, которые будут использоваться, с помощью Prototcol Buffers (ProtoBuf), и генерировать код на основе этого описания. Данный код может быть сгенерированным для разных языков программирования, поэтому существует возможность использовать в разных сервисах специально заточенные под какие-то задачи языки, не сильно задумываясь о их совместимости.

1.4.1.2 Асинхронное

В противовес синхронному взаимодействию можно поставить асинхронное, которое позволяет отправлять запросы, не блокируя вызывающий процесс до получения. Ярким примером для асинхронного

взаимодействия можно назвать Message Queue (MQ). [15] Технологии построенные на основе MQ можно назвать, например, RabbitMQ, ZeroMQ. Данные технологии также имеют возможность использования сервисами, написанными на разных языках.

1.4.2 С использованием хранилища

Помимо прямого взаимодействия между сервисами без использования хранилища можно выделить косвенный способ взаимодействия через различные хранилища. Данный вид взаимодействия удобно использовать, когда сервисы получают какую-то информацию в течение времени, накапливают её и используют самостоятельно и/или другими сервисами. Хранение данной информации можно разделить на долговременное и кратковременное.

1.4.2.1 Долговременные хранилища данных

Долговременные хранилища удобно и стоит использовать, когда необходимо хранить данные продолжительное количество времени и сохранять их между перезапусками. Они используют жёсткие диски для хранения данных, за счёт чего достигается долговременность и надёжность, но также делает использование подобных баз данных (БД) более медленным в сравнении с кратковременными хранилищами.

Поскольку существуют разные потребности к хранилищам данных, можно выделить реляционные БД и нереляционные или, иначе говоря, SQL и NoSQL.

Реляционные базы данных содержат в себе наборы данных, представленные в виде таблиц, которые связаны между собой связями. Данный вид баз данных использует чаще всего в отрасли, так как они созданные для того, чтобы хранить структурированные данные и поддерживать их согласованность и долговечность. Ярким примером реляционной БД можно назвать PostgreSQL.

Но порой возникают ситуации, в которых необходимо хранить неструктурированные данные, например, когда нужно работать с научными статьями. В данном случае вместо SQL базы данных лучшим выбором будет база данных из семейству документоориентированных, которые уже относятся к NoSQL. Один из популярных представителей такого семейства это MongoDB.

1.4.2.2 Кратковременные хранилища данных

Но когда нет нужды в долговременном хранении и необходим быстрый доступ к данным стоит использовать хранилища, использующие вместо жёстких дисков оперативную память. В качестве примера такой базы данных можно привести Redis, NoSQL, который позволяет хранить данные в формате ключ-значение с задержкой до 1 миллисекунды.

1.5 Методы анализа распределённых вычислительных систем

Далее хотелось бы описать методы анализа, применимые к распределённым вычислительным системам, которые позволяют лучше понять состояние системы и обнаружить её недочёты.

1.5.1 Поиск аномалий

Первое, что приходит в голову для анализа распределённых вычислительных систем, это поиск аномалий или поиск выбросов. Наша платформа собирает различные показатели, такие как время обработки запроса, которые можно считать случайной величиной, поскольку даже идентичные запросы выполняются за разные промежутки времени от того, например, что происходят различные прерывания.

Итак, одним из простых и часто используемых методов обнаружения аномалий в нормально распределённых данных является правило трёх сигм. Данное правило состоит в том, что если значение случайной величины отклоняется от её математического ожидания больше, чем на три среднеквадратичных отклонений, то данное значение считается выбросом или аномалией. Так, например, для нормального распределения, вероятность

того, что реализация случайной величины будет лежать далее трёх сигм от математического ожидания, равна 0,0027. [16]

Также существует множество алгоритмов машинного обучения, позволяющих обнаруживать аномалии. Например, изолирующий лес и одноклассовый метод опорных векторов.

1.5.2 Обнаружение связи между вызовами

Также одним из способов анализа распределённых вычислительных систем является обнаружения последовательностей связанных вызовов. Данный способ полезен для определения зависимостей между сервисами, или выявления цепочек вызовов, приходящих к ошибкам или даже отказам.

В качестве оценки вероятности того, что пара вызовов вызов-А и вызов-Б взаимосвязаны, можно использовать частоту вызова вызова-Б во время обработки вызова-А на одном вычислительном узле.

2 РАЗРАБОТКА

Получив общее представление о том, какие требования выдвинуты к нашей платформе, и какие существуют методы оценки эффективности и анализа распределённых вычислительных систем, можно начинать разрабатывать архитектуру платформы и базовые компоненты платформы. Пусть наша платформа называется Monitoring Distributed Systems - platform или MDS - platform.

2.1 Высокоуровневая архитектура

Поскольку платформе необходимо как-то собирать информацию о системе, за который происходит мониторинг, то в ней должен быть процесс, отвечающий за её получение и хранение. Также необходим процесс, отвечающий за обработку полученных событий, с возможностью использования кастомных метрик. Под эти правила, а также под вычисляемые метрики нужно своё хранилище. Предположим на данном этапе, что данная информация поступает на платформу из какого-то стороннего источника.

Итак, теперь есть хранилища, в который хранятся информация о метриках и событиях, произошедших в системе, показанные на рисунке 2.1.

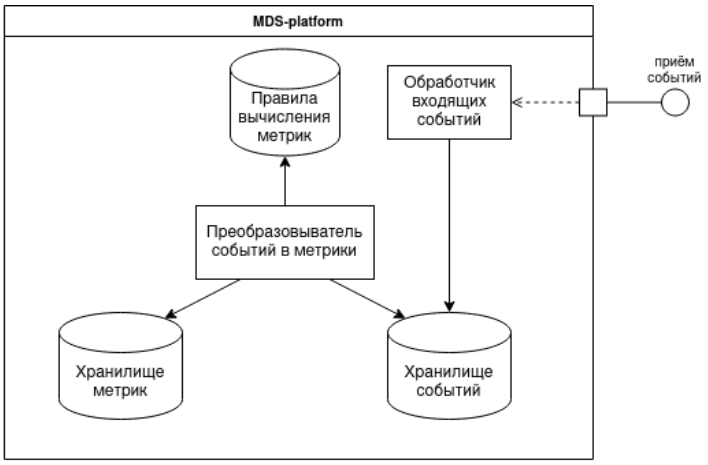

Рисунок 2.1 - архитектура платформы, этап 1

Поскольку пользователи должны иметь доступ к полученным метрикам, будет существовать сервис, отвечающий за работу с клиентами: регистрацию, авторизацию и передачу запросов другому сервису,

отвечающему за обработку запросов таких, как создание правил, получение метрик. За получение метрик и передачу их пользователю будет отвечать другой сервис. Также будет необходимо хранить информацию о пользователях, которая будет использоваться для авторизации, информацию для кастомизации, правила сигнализации и правила вычисления метрик. Доступ к платформе будет через клиенты с GUI и API, показанным на рисунке 2.2.

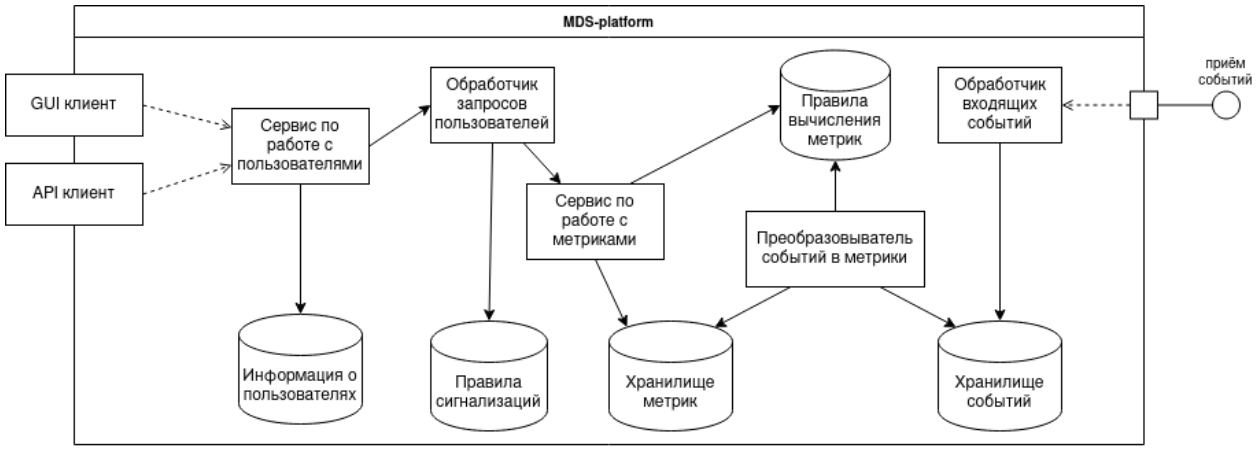

Рисунок 2.2 - Архитектура платформы, этап 2

Теперь у наших пользователей есть возможность получать доступ к вычисляемым метрикам на основе собранной информации. Но необходимо добавить последний процесс, показанный на рисунке 2.3, который будет отвечать за наблюдение за текущими показателями и в случае необходимости отправлять уведомления на нужные адреса.

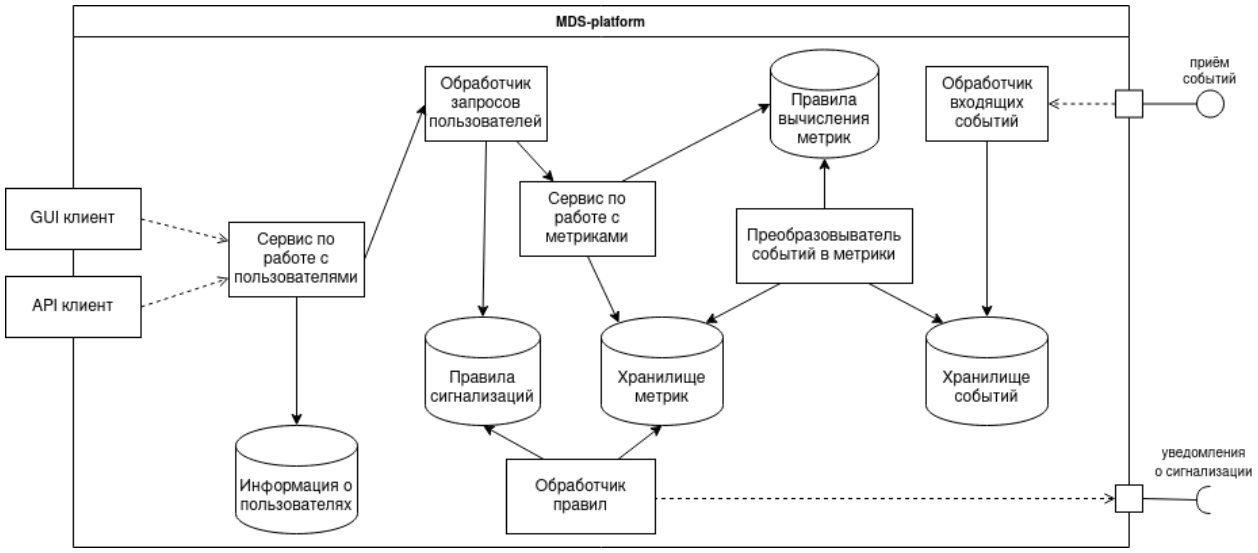

Рисунок 2.3 - Архитектура платформы, этап 3

Таким образом, мы получили базовую архитектуру платформы, которая удовлетворяет поставленным требованиям, но способная претерпеть изменения в будущем, если возникнут новые потребности.

2.2 Разработка отдельных компонентов

Итак, теперь можно приступать к низкоуровневой разработке MDS–platform. Поскольку объём работы, необходимый для полной разработки платформы, слишком велик для рамок ВКР, то будут разработаны только определённые компоненты, а именно:

- компонент, отвечающий за сбор информации о синхронном взаимодействии между собой с помощью RPC;
- компонент, отвечающий за получение, обработку и хранение информации;
- компонент, следящий за срабатыванием правил пользователей, но без возможности кастомизации правил пользователями.

Так же будет реализовано простейшее визуализирование метрик и архитектурного ландшафта с помощью уже готовых решений.

2.2.1 Используемый стек технологий

Стоит заранее упомянуть, какие технологии будут использоваться в нашей работе, почему были выбраны именно они, а не что-то другое.

Начнём с языка программирования. Был выбран python 3, потому что он является одним из самых популярных языков программирования для backend разработки, достаточно прост в освоении и поддерживает различные парадигмы программирования, такие, как объектно-ориентированное программирование (ООП) и аспектно-ориентированное программирование (АОП), которые позволяют более удобно собирать информацию о состоянии системы и происходящих внутри операциях, не изменяя логику используемых функций.

Для взаимодействия между сервисами был выбран RPC подход, поскольку он наиболее удобен для внутреннего взаимодействия, а в качестве

фреймворка был выбран gRPC, так как данный фреймворк можно использовать для разных языков, то есть клиент и сервер могут быть написаны на разных языках программирования. Также он работает на http/2, что позволяет более быстро обмениваться вызовами, и обладает достаточно удобной и подробной документацией[17].

Для хранения получаемой информации будем использовать ElastickSearch, так как это частый выбор для хранения логов, временных рядов и подобной информации, поскольку позволяет хранить огромные объёмы информации, достаточно легко, быстро и эффективно анализировать их и при необходимости масштабировать.

Для визуализации данных будем использовать Grafana вместо Kibana из известного и популярного ELK стэка, поскольку Grafana предназначена больше для визуализации метрик, таких как нагрузка на процессор, в отличии от Kibana, которая больше рассчитана на визуализацию логов, хотя они обе способны визуализировать как метрики, так и логи.

2.2.2 Отслеживатель gRPC вызовов

Для отслеживания gRPC вызовов был написана специальная библиотека, содержащая в себе 2 класса ClientTracer и ServerTracer, которые являются наследниками классов interceprtor-ов (перехватчиков) - существующих классов в gRPC библиотеке, предназначенные для оборачивания вызовов на стороне клиента, а также в дополнительной библиотеке, для оборачивания вызовов на стороне сервера.

Во время инициализации одного из классов перехватчиков создаётся новый процесс, который отвечает за сбор информации о происходящих вызовах и её отправку на платформу. На данный процесс поступает информация через очередь из библиотеки, описанная в таблице 1.

Далее этот процесс, отслеживающий gRPC вызовы, накапливает полученные события, пока их не наберётся нужное количество, а затем отправляет одним сообщением накопленные данные на платформу в виде

json строки. Формат сообщения о событиях, связанных с вызовом gRPC метода, имеет следующий вид, показанный на таблице 2.

Таблица 2 - Формат сообщения о событии

Название поля	Возможные значения	Пояснение
hostname	любое	Имя хоста, позволяющее идентифицировать узел, с которого пришло сообщение.
script	любое	Имя главного исполняемого файла/скрипта.
type	gRPC-client-call-send, gRPC-client-call-receive, gRPC-server-call-send, gRPC-server-call-receive	Тип события, который произошёл.
method	/{service}/{method}	Имя вызванного метода.
function_path	/{…}/{func}	Последовательность функций, из которых был вызван данный метод.
argument	любое	Параметры, с которыми был вызван метод.
GUID	uuid	GUID строка для идентификации вызова.
time0	“yyyy-mm-dd hh:mm:ss.ssssss”	Время начала вызова/обработки метода.
status	grpc.StatusCode	Код завершения вызова.
details	любое	Текст вызванного исключения, если таковое было вызвано.

Во время вызова удалённого метода на клиенте, сервисе А, с помощью ClientTracer замеряется локальное время t_0. При получении на сервере аналогично замеряется t_1. Далее происходит выполнение метода на сервере, сервисе Б, и при завершении его, на сервере замеряется время t_2 и результат

отправляется обратно клиенту. Там при получении ответа замеряется время t_4. Схема взаимодействия двух сервисов изображена на рисунке 2.4.

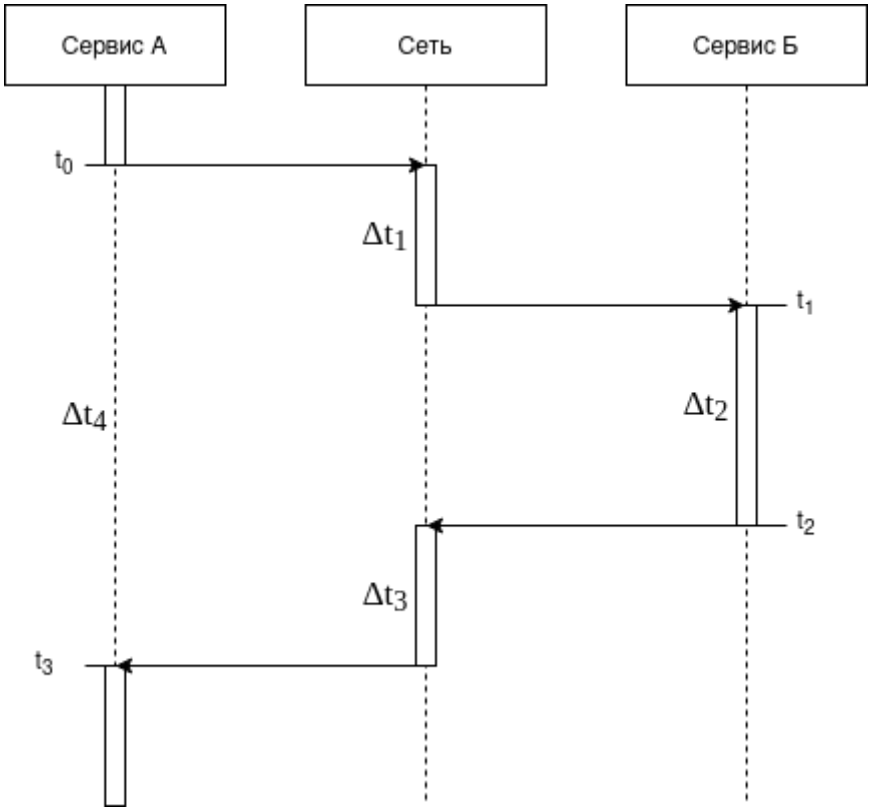

Рисунок 2.4 - Диаграмма последовательности

Таким образом,возможно получить всю необходимую информацию о вызывающих процессах, а также время вызова на клиенте, на сервере и время, потраченное на пересылку. Но в данном подходе есть сложность, состоящая в том, что время на клиенте и сервере может отличаться, поэтому трудно точно посчитать Δt_1 и Δt_3 отдельно. Так как, например, $t_1 - t_0$ может не равняться Δt_1 , поскольку время сервера может сильно отставать или наоборот сильно спешить, что будет создавать неправильное представление о системе.

Поэтому в качестве метрик будет использоваться время обработки запроса на стороне клиента Δt_4, на стороне сервиса Δt_2 и время передачи по сети $\Delta t_1+\Delta t_3$, которое можно выразить как $\Delta t_4 - \Delta t_2$.

Так же можно было заметить, что для идентификации каждого отдельного вызова использовались GUID (globally unique identifier, или глобально уникальные идентификаторы). Это 128-битная символьная строка, которая представляет из себя статистически уникальный идентификатор.

Поскольку возможных значений достаточно много, а если точнее 2^{128}, что примерно равно $3,4 \cdot 10^{38}$, то вероятность того, что будет создано 2 одинаковых идентификатора достаточно мала. Так, например, если создавать миллиард идентификаторов в секунду на протяжении года, то вероятность того, что за это время будет создано 2 одинаковых идентификаторов всего 50%.

2.2.3 Инструмент для встраивания

Поскольку нашу полученную библиотеку могу захотеть использовать на уже существующем приложении, было разработано решение для встраивания в существующие .proto файлы необходимые изменения, конкретно поля для GUID, а также генерация на основе этих файлов кода, с последующим встраиванием в него перехватчиков. Решение использует инструмент для генерации кода protoc.main.

2.2.4 Получатель и обработчик событий

Для получения событий был реализован небольшой http-сервер, который способен обрабатывать только POST запросы на сохранение событий, отмечая время их получения. http-сервер работает в фоновом процессе, получая события и передавая их в родительский процесс.

Родительский процесс сохраняет полученные события в базу данных elasticsearch версии 8.2 с помощью специальной библиотеки elasticsearch-py , являющейся API для более удобной работы с elasticsearch. [18] Сохранение происходит с помощью нескольких тредов, поскольку обращение к базе данных достаточно долгая операция, а сохранение происходит по одному событию за обращение.

2.2.5 Нормализатор событий и обработчик сложных запросов

Поскольку достаточно трудно работать с достаточно разнообразными данными, необходим процесс, который будет нормализировать полученные события, а так же обрабатывать более сложные запросы, такие как

построение графа взаимодействия между процессами систем, за которыми происходит наблюдение. Нормализация и обработка запросов происходит периодически.

Так, например, нормализованные события о grpc вызовах содержут в себе информацию о клиенте, сервере, что за метод вызывался, и время, затраченное на его обработку и время передачи по сети. Данные о вызове складываются в отдельный индекс по идентификатору, соответствующему GUID вызова.

Так же происходит обработка полученных и уже нормализованных событий путём получения из них графа взаимодействия различных узлов, где в качестве узла выступает отдельный процесс на отдельнов вычислительном узле, а в качестве ребёр - информация о взаимодействии пары узлов данного графа.

2.2.6 Визуализация

Визуализация полученных результатов происходит, как упоминалось выше, в Grafana Dashboard, а также с использованием плагина Service Dependency Graph для отображения графа взаимодействия между сервисами.

3 ТЕСТИРОВАНИЕ

Разработав отдельные компоненты платформы, теперь необходимо протестировать её работоспособность на тестовом приложении, которое будет необходимо разработать, а также проверить влияние платформы на производительность системы.

3.1 Разработка тестового приложения

Тестировать наши наработки будем на приложении, которое является классическим интернет магазином. Его архитектура, изображённая на рисунке 3.1.

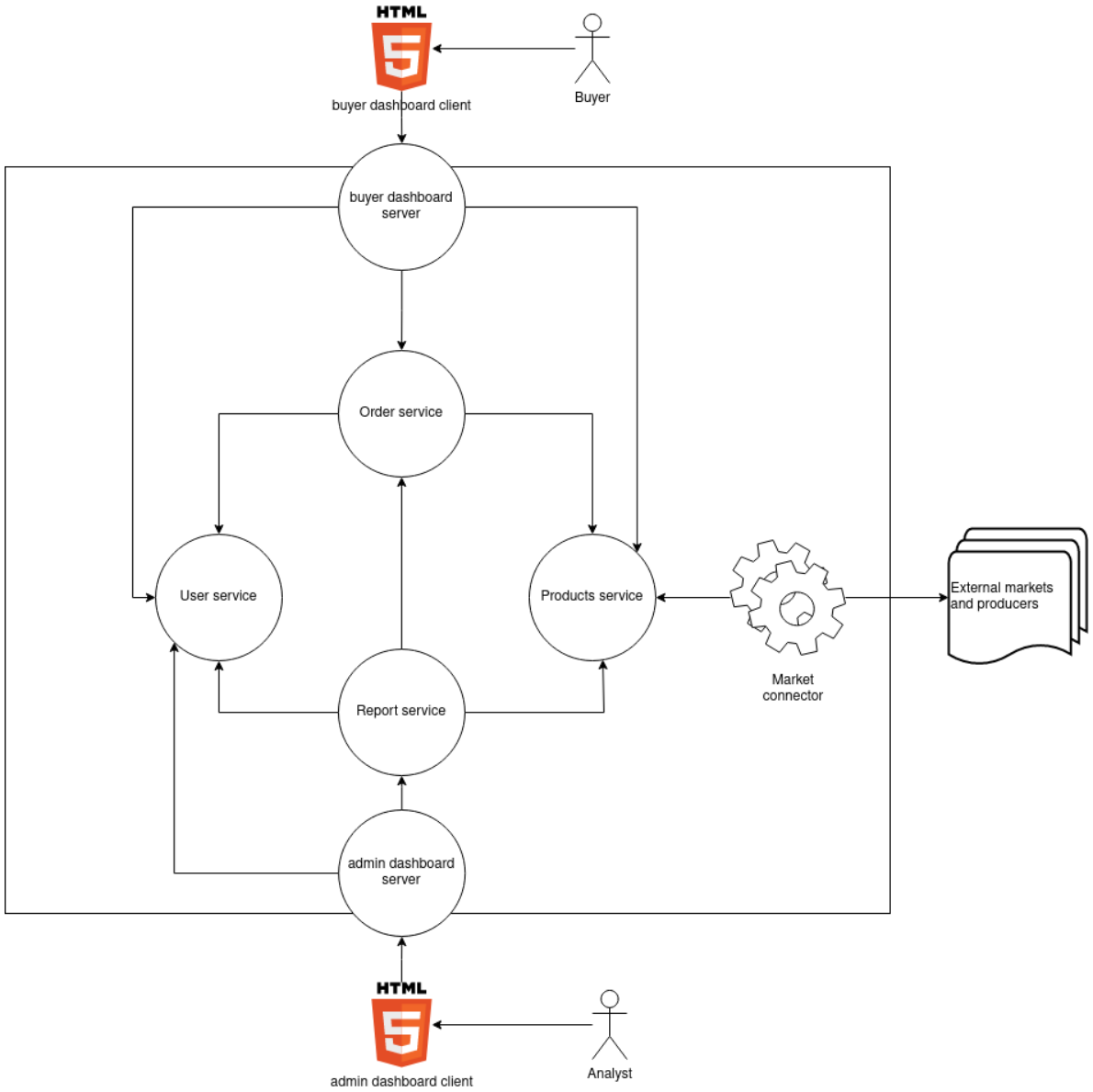

Рисунок 3.1 - Архитектура тестового приложения

Она включает в себя следующие сервисы:

- user service - сервис по работе с пользователями и их данными;
- order service - сервис хранящий и обрабатывающий заказы;
- product service - сервис по хранению, обновлению и выдаче продуктов;
- market connector - сервис, собирающий с внешних источников информацию о продуктах;
- report service - умеет строить различные отчеты и запрашивать данные;
- admin dashboard server|client - сайт, через который аналитик ходит за товарами;
- buyer dashboard server|client - сайт, через который пользователь может посмотреть товары и сделать заказ.

Поскольку данное приложение будет использоваться только с целью тестирования на нём платформы, то html-клиенты разрабатываться не будут, а нагрузка на систему будет создаваться путём случайного выбора одного из доступных методов с равной вероятностью через одинаковые промежутки времени.

Так же реализация самих сервисов не будет содержать предметной части, так как в ней по вышеупомянутым причинам нет необходимости, а вместо неё вычислительная нагрузка будет создавать за счёт подсчёта факториала 1000.

Для придания реалистичности данной системы, каждый сервис будет упакован в docker контейнер и будет запущен и оркестрирован с помощью docker compose, чтобы имитировать работу на кластере. Но время передачи запросов по сети будет значительно меньше реального времени обработки запроса на сервере, поскольку вся система запущена на одном компьютере.

3.2 Тестирование базовых функций платформы

Итак, разработав тестовое приложение, интегрировав наше решение по мониторингу, и запустив его вместе с нашей платформой, с помощью вышеупомянутой grafana можно визуализировать данные, собираемые нашей платформой, как показано на рисунке 3.2.

Рисунок 3.2 - Пример визуализации данных

Также можно определить влияние изменений в системе на отдельные её компоненты. Например, если увеличить вычислительную нагрузку на user service, то можно увидеть, как измениться время обработки запроса на report service. Среднее время выполнения запроса без сетевых задержек можно посчитать по формулам (8) и (9).

$$real_A(B) = total_{AB} - net_A(B), \tag{8}$$

$$net_A(B) = net_{AB} + \sum_C net_B(C) \tag{9}$$

где A - сервис, отправивший запрос AB на сервис B, а C - сервис, на который был отправлен запрос BC во время выполнения запроса AB, net_{AB} сетевые задержки запроса AB, $total_{AB}$ - полное время запроса AB, а $real_{AB}$ - реальное время запроса AB, потраченное на вычисление без сетевых издержек.

Таким образом, время выполнения без сетевых задержек запроса с сервиса admin dashboard на report service до внесения изменений в user service составляло 0.01209, а после них 0.01336 секунд, что составляет увеличение времени выполнения на 10,5%.

3.3 Влияние интеграции платформы на производительность системы

Для определения влияния интеграции платформы на производительность тестируемой системы сравним среднее время выполнения запросов, исходящих с admin dashboard, buyer dashboard и market connector, приведённые в таблице 3.

Таблица 3 - Среднее время выполнения запросов

Сервис	Метод	Среднее время выполнения запроса без интеграции, миллисек.	Среднее время выполнения запроса после интеграции, миллисек.
buyer dashboard	GetUser	1.857	4.445
	GetProduct	2.762	4.199
	GetOrder	4.887	16.922
admin dashboard	GetUser	1.591	4.175
	GetReport	8.204	23.234
market connector	GetProduct	3.676	4.584

Как видно, влияние интеграции платформы в тестовую систему достаточно значительное. Время запросов увеличивается в 3-4 раза, но это можно связать с тем, что сами запросы происходят достаточно быстро, так

как в них не происходит значительно долгих операций, таких как обращение к базам данных и так далее. А так же с тем, что во время создания нескольких классов, например, Stub происходило создание нескольких процессов, что сильно влияло на производительность, особенно в рамках оборудования, на котором проводилось тестирование.

Данную проблему можно исправить тем, что не создавать на каждое создание класса новый процесс, а стараться обходиться минимальным количеством таких процессов.

Весь разработанный код приложен в приложении А.

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе бакалавра была спроектирована платформа для оценки производительности распределённых вычислительных систем и их анализа.

Были решены все поставленные задачи, а именно:

- определены требования, предъявляемые к платформе;
- определены метрики для оценки производительности распределёных вычислительных систем, а так же методы для поиска аномалий в распределённых вычислительных системах и зависимостей между сервисами;
- спроектирован и разработан рабочий прототип целевой платформы;
- проведено тестирование прототипа с оценкой влияния на производительность системы.

Развитие данной платформы может позволить более эффективно собирать информацию о распределённых вычислительных системах и анализировать их. Конечно, на данный момент данная платформа не способна конкурировать с аналогичными продуктами такими, как ZABBIX или New Relic.

Для обеспечения конкурентоспособности необходимо добавить обнаружение зависимостей между цепочками вызовов сервисов, поиск аномалий, уменьшить влияние на проинтегрированные системы и расширить спектр отслеживаемых событий.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 28806-90 Качество программных средств. Термины и определения – 1992. – с 80-87

2. Е.Ю. Климанова, А.Р. Субханкулова, Б.В. Зеленко, О.Ю. Леонтьева, Оценка производительности вычислительных систем // Вестник технологического университета. – 2015. – Т.18, №24, с. 102 – 105.

3. van Steen M., Tanenbaum A.S. A brief introduction to distributed systems // Computing – 2016. – p. 967 – 1009.

4. Распределенные системы. Паттерны проектирования. — СПб.: Питер, 2019. — 224 с.: ил. — (Серия «Бестселлеры O’Reilly»).

5. Shaun F. , Ningchuan X. , GIS Methods and Techniques // Comprehensive Geographic Information Systems- 2018 - p.218 - 245.

6. Datadog, 8 surprising facts about real docker adoption. URL: www.datadoghq.com/docker-adoption/ (дата обращения 2022-05-12)

7. A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture // IEEE Software 33(3):1-1 – 2016.

8. Statista, Microservices usage per organization size. URL: www.statista.com/statistics/1236823/microservices-usage-per-organization-size/ (дата обращения 2022-05-18)

9. Joyce J., Lomow G., Slind K., Unger B. , Monitoring distributed systems// ACM Transactions on Computer Systems 5(2) – p. 121 –150.

10. Muhammad W., Peng L. , Mojtaba S., Amleto D. S. , Gastón M. , Design, Monitoring, and Testing of Microservices Systems: The Practitioners’ Perspective Journal of Systems and Software - 2021.

11. В.А. Балыбердин, А.М. Белевцев, О.А. Степанов, Анализ некоторых подходов к количественной оценке надёжности программных средств // Известия ЮФУ. Технические науки – 2015. – с 157 – 165 .

12. Т.С. Дьячук, Оценка характеристик распределённой системы// ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ Радіоелектроніка, інформатика, управління – 2011 – Том № 2, с. 124 – 131.

13. ГОСТ 27.002–2015 Надёжность в технике. Термины и определения - М., 2016. - 23 с.

14. Хоп Г., Вульф Б. , Шаблоны интеграции корпоративных приложений // Вильямс – 2016

15. Fritsch J.,Walker C., CMQ - A lightweight, asynchronous high-performance messaging queue for the cloud – Journal of Cloud Computing: Advances, Systems and Applications – 2012.

16. В.В.Заляжных, Статистическая обработка результатов испытаний (измерений) — URL: http://www.arhiuch.ru/

17. Документация по gRPC. URL: grpc.github.io/grpc/python/index.html (дата обращения 2022-05-9)

18. Документация по elasticsearch-py URL: https://elasticsearch-py.readthedocs.io/en/v8.2.1/ (дата обращения 2022-05-16)

ПРИЛОЖЕНИЕ А

Разработанный код

Ссылка на хранилище с разработанным кодом: https://github.com/student31415/BachelorDiploma. QR-код с ссылкой представлен на рисунке А.1.

Рисунок А.1 - QR-код ссылка на хранилище