МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра технологий программирования

ЖЕЛТОВСКИЙ

Кирилл Дмитриевич

РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОЙ СИСТЕМЫ, УПРАВЛЯЕМОЙ СОБЫТИЯМИ

Дипломная работа

Научный руководитель:

старший преподаватель кафедры

технологий программирования

М.И. Давидовская

Допущен к защите

«__» ____________ 2021 г.

Зав. кафедрой технологий программирования

доктор технических наук, профессор,

заслуженный деятель науки Республики Беларусь А.Н. Курбацкий

Минск, 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 9

Глава 1 ПРИНЦИПЫ И ПОДХОДЫ К ПРОЕКТИРОВАНИЮ РАСПРЕДЕЛЁННЫХ СИСТЕМ 12

1.1 Микросервисная архитектура 12

1.1.1 Устройство микросервисной архитектуры 12

1.1.2 Преимущества микросервисной архитектуры 13

1.2 Технологии удалённого вызова процедур 14

1.2.1 Общая информация о технологии gRPC 14

1.2.2 Основные концепции удалённого вызова процедур на примере gRPC 15

1.2.3 Использование gRPC 16

1.2.4 Балансирование нагрузки в gRPC 17

1.3 Системы, управляемые событиями 18

1.3.1 Модель «издатель-подписчик» 18

1.3.2 Проблема клиент-серверного взаимодействия 18

Глава 2 ОСНОВНЫЕ ПОНЯТИЯ, КОНЦЕПЦИИ И ВНУТРЕННЕЕ УСТРОЙСТВО APACHE KAFKA 22

2.1 История появления Apache Kafka и области применения 22

2.2. Отличия Apache Kafka от существующих механизмов организации взаимодействия 22

2.2.1 Сравнение Apache Kafka с REST API 23

2.2.2 Сравнение Apache Kafka с Service Bus 23

2.2.3 Сравнение Apache Kafka с базами данных 24

2.2.4 Apache Kafka как стриминговая платформа 24

2.3 Внутреннее устройство Apache Kafka 26

2.3.1 Хранение данных в Apache Kafka 26

2.3.2 Реализация модели «издатель-подписчик» в Apache Kafka 26

2.3.3 Устройство журнала сообщений 27

2.3.4 Линейное масштабирование 28

2.3.5 Репликация данных 29

2.4 Издатели и подписчики в Apache Kafka 29

2.4.1 Издатели в Kafka 30

2.4.2 Подписчики в Kafka 31
2.5 Клиентская библиотека Kafka Streams 31
2.5.1 Понятие времени в системах обработки потоков событий 31
2.5.2 Двойственность потоков данных и таблиц 33
2.5.3 Операции агрегации событий 34
2.5.4 Оконные операции 34
Глава 3 ПРОЕКТИРОВАНИЕ РАСПРЕДЕЛЁННОЙ СИСТЕМЫ, УПРАВЛЯЕМОЙ СОБЫТИЯМИ 36
3.1 Назначение системы 36
3.1.1 Функциональные требования 36
3.1.2 Нефункциональные требования 36
3.2 Общая схема системы 37
3.3 Высокоуровневая архитектура системы 38
3.4 Распределение конфигурации по загрузчикам 40
3.4.1 ZooKeeper как хранилище состояния распределённых систем 40
3.4.2 Организация распределения конфигурации 40
3.4.3 Структура конфигурации 41
3.4.4 Обновление конфигурации загрузчиков 43
3.5 Обработка потоков событий и подсчёт статистики 44
Глава 4 РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОЙ СИСТЕМЫ, УПРАВЛЯЕМОЙ СОБЫТИЯМИ 47
4.1 Решение проблемы синхронизации записи в базу данных и платформу Apache Kafka 47
4.1.1 Необходимость использования базы данных 47
4.1.2 Проблема синхронизации записи в базу данных и Apache Kafka 47
4.1.3 Решение проблемы синхронизации записи в базу данных и Apache Kafka 48
4.2 Обработка и анализ потоков событий деятельности разработчиков с использованием библиотеки Kafka Streams 49
4.3 Реализация возможности осуществления запросов к приложению Kafka Streams 50
4.3.1 Распределённое хранение состояния между экземплярами приложения Kafka Streams 50
4.3.2 Реализация механизма осуществления запросов 51

4.4 Реализованное веб-приложение с использованием фреймворка React 52
4.5 Демонстрация работы системы 57
4.5.1 Добавление нового репозитория 57
4.5.2 Загрузка и обработка потока событий для добавленного репозитория 58
4.5.3 Просмотр результатов обработки данных 58
ЗАКЛЮЧЕНИЕ 61
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 63
ПРИЛОЖЕНИЯ 64

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ

RPC	Remote Procedure Call (Удалённый вызов процедур).
API	Application Programming Interface (Программный интерфейс приложения).
TCP	Transmission Control Protocol (Протокол управления передачей).
HTTP	Hyper Text Transfer Protocol (Протокол передачи гипертекста).
ESB	Enterprise Service Bus (Сервисная шина предприятия).
GPS	Global Positioning System (Система глобального позиционирования).

РЕФЕРАТ

Дипломная работа, 62 с., 29 рис., 9 приложений.

Ключевые слова: РАСПРЕДЕЛЁННЫЕ СИСТЕМЫ, ПОТОКОВАЯ ОБРАБОТКА ДАННЫХ, МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, APACHE KAFKA, JAVA, PROTOBUF, ZOOKEEPER, gRPC.

Объект исследования — объектом исследования являются высоконагруженные распределённые системы и технологии, использующиеся для их разработки и поддержки. В качестве предмета исследования выбираем проектирование, разработку и исследование характеристик распределённой системы для анализа активности разработчиков.

Цели работы — рассмотреть основные понятия, ключевые принципы, алгоритмы и компромиссы, возникающие при разработке распределённых систем, технологии, использующиеся для построения данных систем, изучить основные концепции лежащие в основе Apache Kafka, а также спроектировать распределённую систему, управляемую событиями, предназначенную для анализа активности разработчиков.

Методы исследования — а) теоретические: изучение литературы, посвященной построению высоконагруженных распределённых систем; б) практические: обобщение работ в области разработки распределённых систем, моделирование, проектирование и разработка распределённой системы, управляемой событиями, с использованием Apache Kafka, gRPC и Spring.

Результатами являются — разработанная распределённая система, управляемая событиями, для анализа активности разработчиков.

Область применения — разработка высоконагруженных распределённых систем для широкого класса задач.

РЭФЕРАТ

Дыпломная работа, 62 ст., 29 мал., 9 дадаткаў.

Ключавыя словы: РАЗМЕРКАВАНЫЯ СІСТЭМЫ, СТРУМЕНЕВАЯ АПРАЦОЎКА ДАДЗЕНЫХ, МІКРАСЭРВІСНАЯ АРХІТЭКТУРА, APACHE KAFKA, JAVA, PROTOBUF, ZOOKEEPER, gRPC.

Аб'ект даследавання — аб’ектам даследавання з’яўляюцца высоканагружаныя размеркаваныя сістэмы і тэхналогіі, якія выкарыстоўваюцца для іх распрацоўкі і падтрымкі. У якасці прадмета даследавання выбіраем праектаванне, распрацоўку і даследаванне характарыстык размеркаванай сістэмы для аналізу актыўнасці распрацоўшчыкаў.

Мэты працы — разгледзець асноўныя паняцці, ключавыя прынцыпы, алгарытмы і кампрамісы, якія ўзнікаюць пры распрацоўцы размеркаваных сістэм, тэхналогіі, якія выкарыстоўваюцца для пабудовы дадзеных сістэм, разгледзець асноўныя канцэпцыі, якія ляжаць у аснове Apache Kafka, а таксама спраектаваць размеркаваную сістэму, кіруемую падзеямі, прызначаную для аналізу актыўнасці распрацоўшчыкаў.

Метады даследавання — а) тэарэтычныя: вывучэнне літаратуры, прысвечанай пабудове высоканагружаных размеркаваных сістэм; б) практычныя: абагульненне работ у галіне распрацоўкі размеркаваных сістэм, мадэляванне, праектаванне і распрацоўка размеркаванай сістэмы, кіруемай падзеямі, з выкарыстаннем Apache Kafka, gRPC і Spring.

Вынікамі з'яўляюцца — распрацаваная размеркаваная сістэма, кіруемая падзеямі, для аналізу актыўнасці распрацоўшчыкаў.

Вобласць ужывання — распрацоўка высоканагружаных размеркаваных сістэм для шырокага класа задач.

ESSAY

Graduate Work, 62 p., 29 illustrations, 9 appendixes.

Keywords: DISTRIBUTED SYSTEMS, STREAMING DATA PROCESSING, MICROSERVICE ARCHITECTURE, APACHE KAFKA, JAVA, PROTOBUF, ZOOKEEPER, gRPC.

Object of research is data-intensive distributed systems and technologies used to develop and maintain these systems. As the subject of study, we choose the design, development and analysis of characteristics of the distributed system for analyzing developer’s activity.

Purpose is to consider general distributed systems concepts, key concepts and algorithms, tools for building distributed systems, to study Apache Kafka concepts and to design a distributed system that analyzes developer’s activity.

Methods of research are a) theoretical: a study of the literature on building data-intensive distributed systems; b) practical: summarizing the work in the field of building distributed systems, modeling, designing and developing distributed system using Apache Kafka, gRPC and Spring.

The result is designed and implemented distributed event-driven system which is used for analyzing developer’s activity.

Scope is the developing data-intensive distributed systems for a wide class of tasks.

ВВЕДЕНИЕ

Со стремительным развитием информационных технологий возрастают ожидания пользователей от продукта. Возросшие ожидания пользователей порождают дальнейшее развитие информационных технологий. Как результат, продукты неизбежно усложняются, стремясь удовлетворять появляющиеся желания и требования пользователей. А значит повышаются и усложняются требования к соответствующим программным системам. Разработчики встают перед необходимостью поиска и применения решений, которые будут надёжно работать в условиях высокой нагрузки, больших потоков данных, при этом позволяя эффективно вести разработку нового функционала и поддерживать уже существующий.

Одним из ответов на вопрос устройства таких систем стала микросервисная архитектура, когда единое монолитное приложение разбивается на большое количество меньших частей, которые могут разрабатываться независимыми и изолированными командами, а также самостоятельно развёртываться и, следовательно, масштабироваться. Однако ещё одним вопросом, на который приходится отвечать, является вопрос о том, кто владеет данными и как ими управлять. Разъединение связанных сервисов с использованием потока событий оказывается довольно эффективным решением, хорошо показывающим себя на практике в условиях высоконагруженных систем. И такое решение является вполне естественным. Пользователи взаимодействуют некоторым образом с системой, нажимая на кнопки, или, скажем, переходя по ссылкам. Всё это является примером событий, происходящих в реальной жизни, и оказывается, что эти и подобные этим события хорошо переносятся на концепцию событий в контексте программных систем.

Проектируемая распределённая система, управляемая событиями, служит предметом исследования и примером того, как можно, нужно или не стоит разрабатывать распределённые системы. Система предназначена для анализа

активности разработчиков по их действиям на платформах хостинга кода, к примеру GitHub. Данные платформы предоставляют API для получения информации о действиях, совершенных в репозиториях, которые могут использоваться для сбора статистика и анализа продуктивности разработчиков. Данная система может найти применение в компаниях, занимающихся разработкой ПО и желающих измерять и улучшать эффективность разработчиков.

Объектом исследования являются высоконагруженные распределённые системы и технологии, использующиеся для их разработки и поддержки. В качестве предмета исследования выбираем проектирование, разработку и исследование характеристик распределённой системы для анализа активности разработчиков.

В качестве целей работы выделены: а) рассмотрение основных понятий, ключевых принципов, алгоритмов и компромиссов, возникающих при разработке распределённых систем, технологий, использующихся для построения данных систем, б) изучение основных концепций, лежащих в основе Apache Kafka, а также проектирование распределённой системы, управляемой событиями, предназначенной для анализа активности разработчиков на основе их действий на платформах хостинга кода.

Методы исследования включают изучение литературы, посвященной построению высоконагруженных распределённых систем; обобщение работ в области разработки распределённых систем, моделирование, проектирование и разработка распределённой системы, управляемой событиями, с использованием Apache Kafka, gRPC и Spring.

Областью применения является разработка высоконагруженных распределённых систем для широкого класса задач.

В главе 1 рассмотрены микросервисная архитектура, построение межсервисной коммуникации на основе удалённого вызова процедур с

использованием технологии gRPC, проблематика и области применения систем, управляемых событиями.

В главе 2 рассмотрены основные понятия, концепции и устройство Apache Kafka. Рассмотрена история создания и возможные области применения Apache Kafka. Проведён сравнительный анализ Apache Kafka с REST API, базами данных, шинами и классическими очередями. Выявлены преимущества и недостатки Apache Kafka и возможности платформы для масштабирования системы. Рассмотрены основные принципы обработки потоков событий и возможности библиотеки Kafka Streams.

В главе 3 описано предназначение системы, выявлен набор функциональных и нефункциональных требований, описана архитектура спроектированной распределённой системы, управляемой событиями, для анализа активности разработчиков.

В главе 4 описано решение задачи распределения конфигурации по активным загрузчикам, решение задачи синхронизации записи в базу данных MySQL и платформу Apache Kafka. Приводится описание реализации обработки потоков событий с использованием библиотеки Kafka Streams с дедуплицированием событий и возможностью осуществления запросов к текущему состоянию приложения. Продемонстрирована работа полного цикла распределённой системы с примерами взаимодействия различных компонент.

Глава 1 ПРИНЦИПЫ И ПОДХОДЫ К ПРОЕКТИРОВАНИЮ РАСПРЕДЕЛЁННЫХ СИСТЕМ

1.1 Микросервисная архитектура

Микросервисы — это архитектурный подход к разработке приложений. Микросервисная архитектура представляет собой набор небольших автономных сервисов. Каждый сервис должен реализовывать только одну бизнес-задачу. Микросервисы распределены и слабо связаны, таким образом, что изменения в одной части приложения не затрагивают другие части и, соответственно, имеют меньший шанс вызвать появление ошибок во всём приложении.

1.1.1 Устройство микросервисной архитектуры

Отличие микросервисной архитектуры от более традиционных, монолитных подходов состоит в том, как именно приложение разбивается на части, учитывая основные его функции. При микросервисном подходе каждая функция называется сервисом, и может быть разработана и развёрнута независимо. Это означает, что каждый сервис может работать (и ломаться) без острого негативного эффекта на остальные части приложения. Это облегчает технологическую часть DevOps и позволяет сделать постоянную итерацию разработки продукта и интеграции проще и достижимой более простым способом.

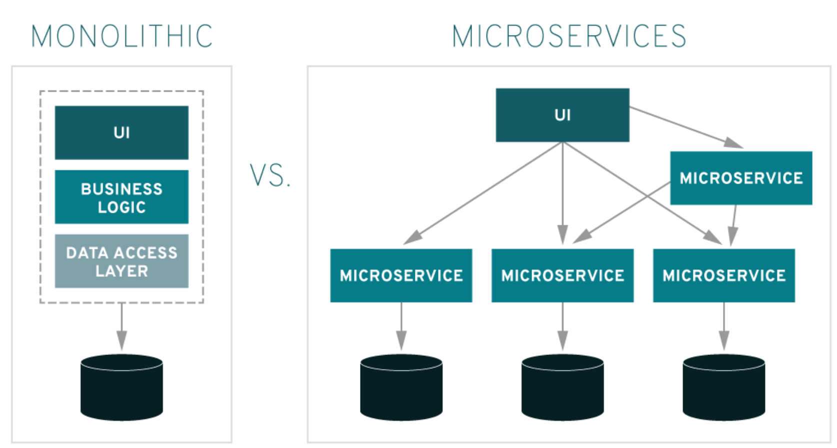

Рисунок 1.1 — Отличие монолитной и микросервисной архитектуры. [6]

Рассмотрим микросервисную архитектуру на примере веб-сайта розничной торговли. Наверняка такой сайт будет содержать строку поиска товаров. В случае микросервисной архитектуры функция поиска должна была бы быть реализована отдельным сервисом. Также данный сайт может содержать секцию с рекомендациями похожих товаров. Данные рекомендации также предоставляются отдельным сервисом, отвечающим исключительно за эту функцию.

1.1.2 Преимущества микросервисной архитектуры

Среди преимуществ микросервисной архитектуры выделяются следующие:

1) Ускорение разработки.

   Циклы разработки становятся короче, и микросервисная архитектура буквально создана для гибкой разработки и обновлений.

2) Масштабируемость.

   Как только потребность в каком-то сервисе увеличивается, можем создать несколько экземпляров данного сервиса, что было бы невозможным или затратным при монолитном подходе.

3) Гетерогенность системы.

   Так как все сервисы приложения являются самостоятельными и общаются с другими сервисами через предоставляемые API, разработчики имеют возможность разрабатывать данные сервисы с использованием различных технологий и языков программирования.

4) Поддержка приложения.

   Поскольку традиционное большое монолитное приложение разбивается на более мелкие части при микросервисном подходе, то и разобраться в этих частях, и, следовательно, во всем приложении становится проще.

Несмотря на все преимущества, которые имеет микросервисная архитектура, следует учитывать, что данные преимущества не появляются из пустоты. Разработка приложений с использованием микросервисной архитектуры является более сложной и требует более серьезных навыков и опыта.

1.2 Технологии удалённого вызова процедур

Удалённый вызов процедур — это мощная техника для создания распределённых приложений, базирующихся на клиент-серверном взаимодействии. В отличие от обычного вызова процедур вызываемая процедура не обязана находится в одном адресном пространстве с вызывающей её процедурой и, даже более того, не обязана находится на одном и том же компьютере. Два взаимодействующих процесса могут находится на разных компьютерах и связываются посредством сети.

1.2.1 Общая информация о технологии gRPC

gRPC — это современный высокопроизводительный RPC фреймворк с открытым исходным кодом, который поддерживает большое количество платформ [3]. gRPC эффективно соединяет сервисы внутри и через дата центры с подключаемой поддержкой балансирования нагрузки, трассировки, аутентификации и т. д.

Основные сценарии использования:

1) Высокопроизводительное соединение сервисов-полиглотов[1] в микросервисной архитектуре.
2) Соединение мобильных и браузерных клиентов с сервером.
3) Генерация эффективных клиентских библиотек.

gRPC также обладает рядом функций, которые выделяют данный фреймворк перед его аналогами:

1) Генерация идиоматичных клиентских библиотек на 10 языках.

1 Сервисы, написанные на различных языках программирования.

2) Высокоэффективный на уровне транспорта при этом обладающий простым фреймворком для объявления сервисов.
3) Двунаправленный стриминг с HTTP/2 в качестве транспорта.
4) Встраиваемая поддержка аутентификации, балансирования нагрузки и проверки «здоровья» сервисов.

1.2.2 Основные концепции удалённого вызова процедур на примере gRPC

В gRPC клиентское приложение может напрямую вызывать методы серверного приложения таким же образом, как если бы это был локальный объект, упрощая тем самым разработку распределённых приложений и сервисов. Как и многие системы удалённого вызова процедур, gRPC основан на идее объявления сервисов, содержащих методы, которые могут быть вызваны удалённо с их параметрами и возвращаемыми значениями. На стороне клиента клиент имеет заглушку или просто клиент, который предоставляет те же методы, что и сервер (Рисунок 1.2).

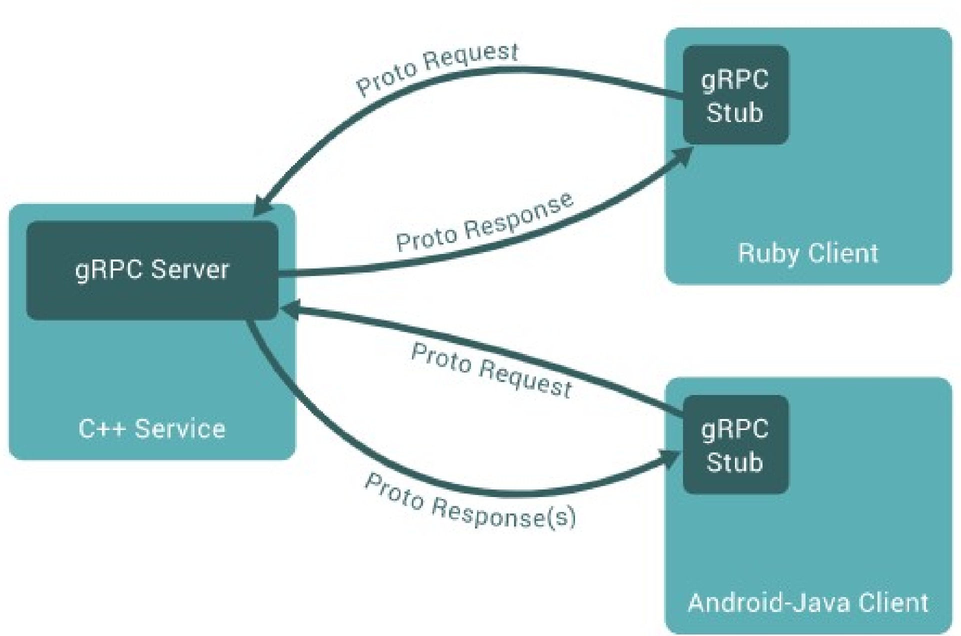

Рисунок 1.2 — Схема взаимодействия сервисов с использованием gRPC [3].

gRPC клиенты и серверы могут исполняться и коммуницировать друг с другом в большом разнообразии окружений — от серверов внутри Google до

обыкновенного компьютера — и могут быть написаны на любом из поддерживаемых gRPC языков. Например, мы можем создать сервер, написанный на Java, к которому будут обращаться клиенты, написанные на Go, Python или Ruby. В добавок ко всему, множество Google API предоставляют gRPC версии своих интерфейсов, позволяя легко интегрировать функциональность, предоставляемую Google, в свои приложения.

1.2.3 Использование gRPC

По умолчанию, gRPC использует Protocol Buffers, механизм, разработанный Google, для сериализации структурированных данных. Первый шаг в работе с gRPC — это объявление структуры данных, которые мы хотим сериализовать в файлах объявлений, специальных файлах с расширением .proto. Такие структуры данных называются сообщениями, и каждое сообщение является небольшой логической записью информации, состоящей из нескольких пар имён и значений, называющихся полями (Рисунок 1.3).

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

Рисунок 1.3 — Объявление сообщения.

Затем, объявив все необходимые структуры данных, мы можем использовать компилятор для генерации соответствующих классов данных на предпочитаемом языке по объявлениям в файле объявлений. Такие сгенерированные классы имеют простые методы доступа и модификации, а также методы для сериализации и десериализации структуры данных.

gRPC сервисы объявляются в тех же файлах, что и сообщения, с RPC методами с указанными параметрами и возвращаемыми типами (Рисунок 1.4).

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Рисунок 1.4 — Объявление сервиса.

1.2.4 Балансирование нагрузки в gRPC

Для построения приложений, которые смогут выдерживать большие нагрузки и обрабатывать большое количество одновременно поступающих запросов, зачастую применяется горизонтальное масштабирование. Это означает, что если какой-либо сервис должен справляться с большими нагрузками, то мы создаем несколько одновременно работающих экземпляров этого сервиса, которые как раз и будут обрабатывать поступающие запросы.

gRPC поддерживает разные варианты балансирования нагрузки — прокси, клиентская, отдельный балансирующий сервер, сервисная сетка[2].

gRPC поддерживает балансирование на стороне клиента. Однако стандартные клиентские алгоритмы балансировки примитивны и, при необходимости, может возникнуть потребность в собственной реализации.

Так как gRPC использует в качестве транспорта HTTP/2, который мультиплексирует запросы, следует быть внимательным с тем, на каком уровне OSI осуществляется балансирование, в случае использования прокси. Если балансирование работает на транспортном уровне, то получим не

2 Service mesh (англ.). Набор прокси-серверов, расположенных рядом с процессом приложения.

балансирование каждого запроса клиента к серверу, а просто изначальный выбор клиентом какого-то сервера, к которому он будет впоследствии обращаться при каждом вызове по установленному TCP соединению.

1.3 Системы, управляемые событиями

Микросервисный подход к архитектуре снискал большую популярность, позволив эффективно масштабировать высоконагруженные приложения. Однако, если в теории разделение большого приложения на большое количество меньших сервисов, должно было бы повысить надежность всего приложения целиком, а в случае если один сервис выходит из строя, то система продолжает работать. На практике же такая сеть высокосвязанных сервисов приводит к ситуациям массового сбоя системы, когда один сервис начинает медленно отвечать или выходит из строя, и все сервисы, зависимые от затормозившего, также начинают выходить из строя по каскадной цепочке.

1.3.1 Модель «издатель-подписчик»

Модель «издатель-подписчик» представляет из себя модель взаимодействия, шаблон, в котором издатель (производитель) направляет сообщение (событие) не напрямую получателю. Вместо этого, издатель каким-то образом классифицирует сообщение, а получатель подписывается на этот определённый класс сообщений. Обычно системы, работающие по модели «издатель-подписчик», имеют брокера или несколько брокеров, которые являются центральной точкой, где собираются все сообщения.

1.3.2 Проблема клиент-серверного взаимодействия

Рассмотрим проблемы, возникающие при клиент-серверном взаимодействии большого количества сервисов, на примере приложения, которое отправляет свою мониторинговую информацию специально выделенному под это сервису. В самом начале разработки, мы решаем, что наши сервисы будут выгружать мониторинговую информацию, напрямую используя API сервиса метрик (Рисунок 1.5).

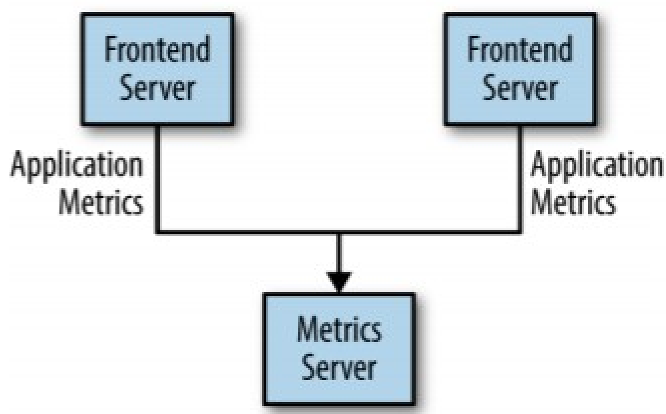

Рисунок 1.5 — Простейший способ организации взаимодействия сервисов [2].

Данное решение является решением простой проблемы. Но представим, что теперь хотим не просто собирать метрики и отображать их, но и анализировать. Для этого мы создаём новый сервис, который может получать метрики, хранить их и анализировать. При этом модифицируем наши существующие сервисы таким образом, чтобы они отправляли свои метрики сразу в две системы сбора метрик. Еще через некоторое время решаем, что еще больше приложений хотят иметь доступ к метрикам и использовать их для разных целей. Все это превращается в систему, в которой становится тяжело уследить за сетью взаимосвязей, не говоря о надежности всей системы целиком, возможностях разработки нового функционала и поддержки уже существующего.

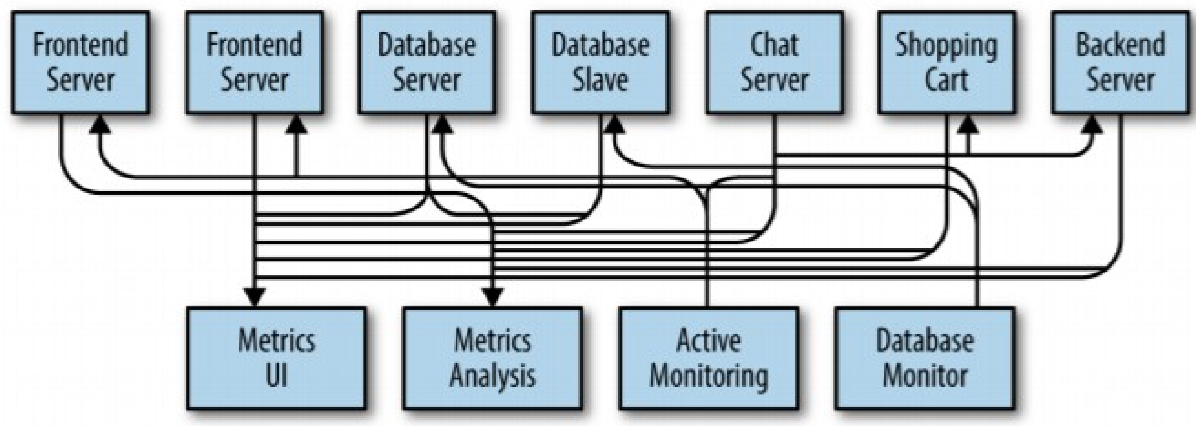

Рисунок 1.6 — Добавление новых приложений в систему [2].

Технический долг и проблемы, возникающие при такой схеме взаимодействия, очевидны. Для того чтобы избавиться от данных проблем, мы можем разъединить приложения, которые предоставляют свои метрики, и приложения, которые эти метрики используют, убрать явную связь между ними. Это достигается за счёт введения нового компонента в систему, который будет получать метрики от всех сервисов, которые их предоставляют. А приложения, которые обрабатывают метрики в разных целях, будут обращаться к этому специально выделенному компоненту.

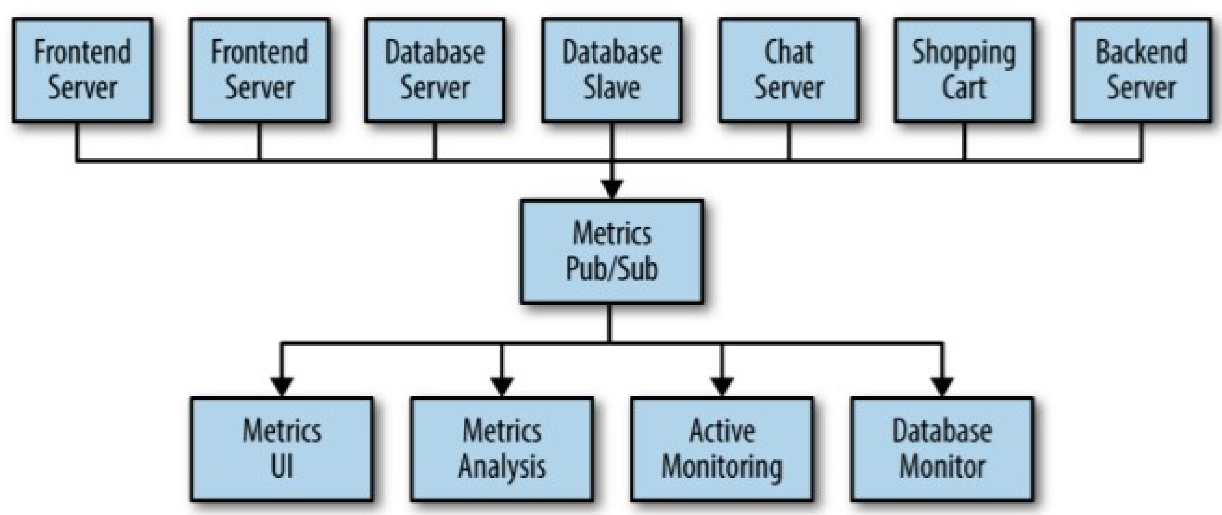

Рисунок 1.7 — Взаимодействие согласно шаблону “издатель-подписчик” [2].

ВЫВОДЫ

1) Рассмотрены понятия микросервисной архитектуры, её устройство, преимущества и области применения. Определено, что технологии удалённого вызова процедур хорошо подходят для коммуникации сервисов в микросервисной архитектуре.
2) Проведён обзор основных концепций технологий удалённого вызова процедур на основе gRPC. Проанализированы преимущества и возможные случаи использования технологий удалённого вызова процедур.

3) Рассмотрены проблемы, возникающие при построении клиент-серверного взаимодействия в микросервисной архитектуре. Предложено решение данных проблем с использованием модели «издатель-подписчик» и организации взаимодействия сервисов в системе путём использования событий.

Глава 2 ОСНОВНЫЕ ПОНЯТИЯ, КОНЦЕПЦИИ И ВНУТРЕННЕЕ УСТРОЙСТВО APACHE KAFKA

2.1 История появления Apache Kafka и области применения

Apache Kafka — это система передачи сообщений, действующая по модели «издатель-подписчик» и призванная решить проблемы, возникающие при клиент-серверном взаимодействий в архитектурах с большим количеством связей и зависимостей. Внутри Kafka устроена таким образом, что позволяет хранить сообщения сколько-угодно времени, что даёт возможность перевоспроизведения сообщений, о случаях использования которой будет рассказано в дальнейшем.

Изначально Kafka начала разрабатываться компанией LinkedIn, когда они непосредственно столкнулись с проблемами, описанным в первой главе. В начале пути в LinkedIn было монолитное приложение, написанное на Java, которое впоследствии было разделено на порядка тысячи отдельных сервисов, что привело к сложным зависимостям, которые приводили к нестабильностям, проблемам с версиями использующихся сервисов и библиотек.

Для решения данных проблем было решено внедрить асинхронный подход при взаимодействии сервисов, что позволило передавать триллионы сообщений в день внутри сервисов компании. Для реализации данного подхода и была разработана система, получившая название Apache Kafka.

2.2. Отличия Apache Kafka от существующих механизмов организации взаимодействия

Данный раздел сосредоточится на выявлении того, чем Apache Kafka не является, какие имеет отличия от других существующих механизмов организации межсервисного взаимодействия, и в каких случаях данные особенности и отличия оказываются выгодными при разработке распределённых систем, управляемых событиями.

2.2.1 Сравнение Apache Kafka с REST API

Kafka предоставляет асинхронный протокол взаимодействия приложений, и имеет существенные отличия от TCP, HTTP или RPC протоколов. Ключевое отличие заключается в наличии брокера. Брокер — это отдельная часть инфраструктуры которая распространяет сообщения всем сторонам, которые заинтересованы в этом, а также способна хранить сообщения сколь-угодно большое количество времени. Это делает Apache Kafka подходящим выбором для потоковой обработки и построения распределённых систем, управляемых событиями.

REST представляет из себя клиент-серверный протокол, также клиент-серверное взаимодействие можно построить при помощи Kafka. Однако данные подход не является естественным, поскольку брокер в таком случае не добавляет никакой ценности, как только от него перестает требоваться распространение событий. Подходящей областью использования Kafka является обработка на основе событий, где события могут быть определены как некоторые бизнес факты, которые могут быть полезны нескольким сервисам и для которых есть ценность в их хранении.

2.2.2 Сравнение Apache Kafka с Service Bus

Рассматривая Kafka с одной стороны, принимая во внимание наличие коннекторов, которые предоставляют возможность забирать и отдавать данные между большим количеством интерфейсов и хранилищ данных, а также API для потоковой обработки, это действительно выглядит как Enterprise Service Bus. Однако отличие, и оно существенно, состоит в том, что ESB предназначена для интеграции наследственных систем и коробочных решений и имеет довольную низкую пропускную способность, а также поощряет использование клиент-серверного взаимодействия, что отсылает к предыдущему пункту.

Kafka же является стриминговой платформой и естественно делает акцент на высокую пропускную способность и потоковую обработку. Кластер в Kafka это по определению распределённая система, предоставляющая хранилище, линейное масштабирование и высокую доступность. Наличие кластера как раз и отличает Kafka от многих традиционный систем передачи сообщений, которые обыкновенно ограничены одним сервером, а если в них и присутствует возможность масштабирования, то такие возможности сильно ограничены в сравнении с Kafka. Инструменты по типу Kafka Streams и KSQL позволяют писать простые программы, обрабатывающие события.

Таким образом, хоть Kafka и может выглядеть похожей га ESB, на самом деле её устройство значительно отличается. Kafka предоставляет гораздо большую пропускную способность, доступность и хранилище.

2.2.3 Сравнение Apache Kafka с базами данных

Также можно провести сравнение Kafka с базой данных. И Kafka, и базы данных предоставляют хранилище. Kafka способна хранить сотни терабайтов, а также предоставляет SQL подобный язык определения запросов к данным и поддерживает транзакции, что делает её очень схожей с базами данных.

В Kafka присутствует много элементов, которые присущи базам данных, но, несмотря на это, Kafka можно описать как базу данных изнутри наружу. Разработчик может положить в Kafka набор данных и затем выполнить запросы к нему при помощи KSQL. Стоит отметить, что с точки зрения производительности Kafka больше оптимизирована для непрерывной потоковой обработки событий, нежели пакетной обработки данных. Kafka спроектирована с учетом потоковой обработки на первом месте и хранения данных на втором.

2.2.4 Apache Kafka как стриминговая платформа

Правильнее всего и содержательнее определять Apache Kafka как стриминговую платформу. По сути Kafka представляет собой кластер, с

которым можно коммуницировать самыми разными способами и через разные интерфейсы. Разработчик может выбрать подходящий для конкретной ситуации способ среди широкого разнообразия клиентов на большом количестве языком программирования, воспользоваться REST API. Также Kafka предоставляет специальный API для потоковой обработки: Stream API и KSQL. Данные инструменты дают возможность обрабатывать данные на ходу, позволяя разработчикам фильтровать потоки, объединять их, агрегировать, хранить промежуточные состояния.

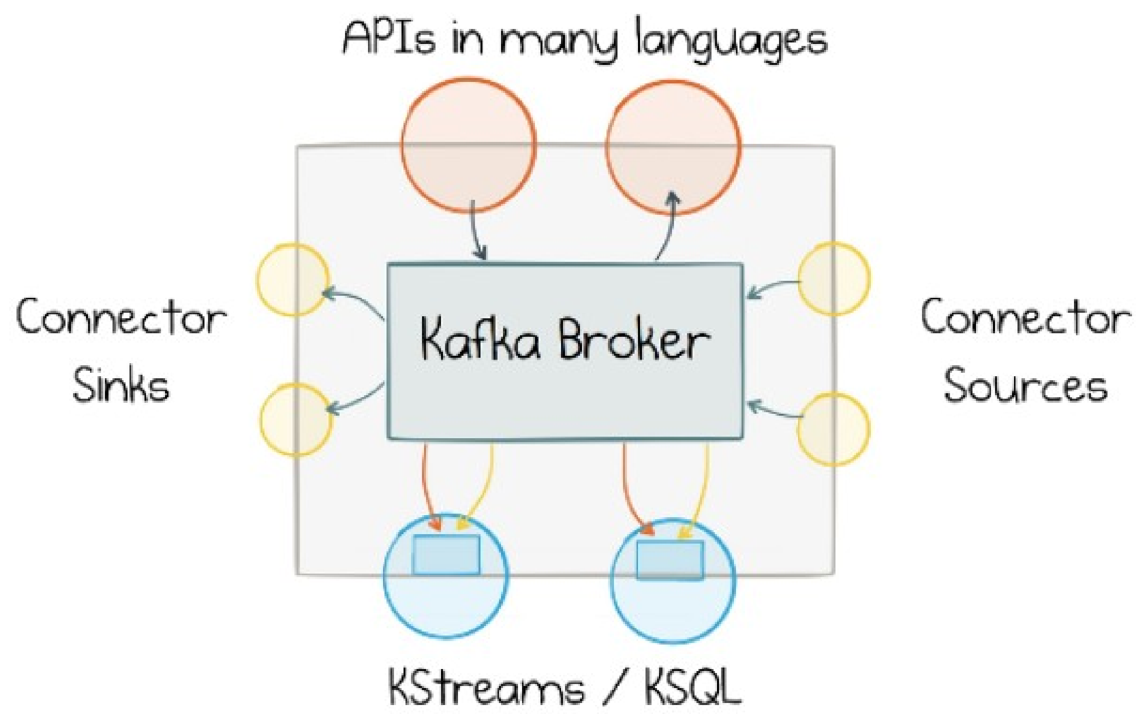

Рисунок 2.1 — Ключевые компоненты стриминговой платформы [1].

Третий способ взаимодействия с Kafka — это коннекторы. Коннекторы являются целой экосистемой интерфейсов взаимодействия с самыми разными источниками данных.

Apache Kafka, как стриминговая платформа, соединяет все эти инструменты вместе с целью предоставления разнообразия доступных

инструментов для поддержки всех возможных способов использования данных в компании.

2.3 Внутреннее устройство Apache Kafka

2.3.1 Хранение данных в Apache Kafka

По существу, кластер в Kafka — это набор файлов, заполненных сообщениями, распределенный по нескольким вычислительным узлам. Kafka разрабатывалась с учётом того, чтобы система была способна обрабатывать огромное количество данных, распределённых по сети.

Как и многие результаты в информатике, отличная способность Kafka к масштабированию обусловлена довольно простой концепцией. Основополагающей абстракцией в Kafka является журнал с разделами (partitioned log). Данный журнал представляет из себя набор файлов, в которые можем только добавлять данные в конец, и эти файлы распределены по нескольким серверам в кластере. Данное устройство делает операции последовательного доступа естественно вытекающими из устройства физического обеспечения.

Кластер в Kafka — это распределённая система, в которой данные хранятся на нескольких серверах, что позволяет обеспечить отказоустойчивость и предусмотреть возможность линейного масштабирования.

2.3.2 Реализация модели «издатель-подписчик» в Apache Kafka

Сообщения в Kafka хранятся в темах (topics). Издатели записывают сообщения в соответствующую тему, а подписчики осуществляют подписку на интересующие их темы, и начинают получать новые сообщения из этих тем. Грубыми аналогами для темы являются таблица базы данных или папки в файловой системе. Каждая тема состоит из журналов сообщений. Сообщения могут записываться только в конец журнала. Обыкновенно одна тема состоит из нескольких журналов. И именно журналы являются средством масштабирования.

2.3.3 Устройство журнала сообщений

Фундаментальной структурой данных в Kafka является журнал сообщений с разделами, который может быть повторен с самого начала или с определённого момента. Журнал представляет из себя структуру данных, которая содержит элементы, называемые сообщения; сообщения могут записываться только в конец журнала. Когда клиент хочет прочитать сообщения из Kafka, он смотрит, на какой позиции было последнее сообщение, которое он прочитал, затем последовательно читает новые сообщения в том порядке, в котором они расположены в журнале.

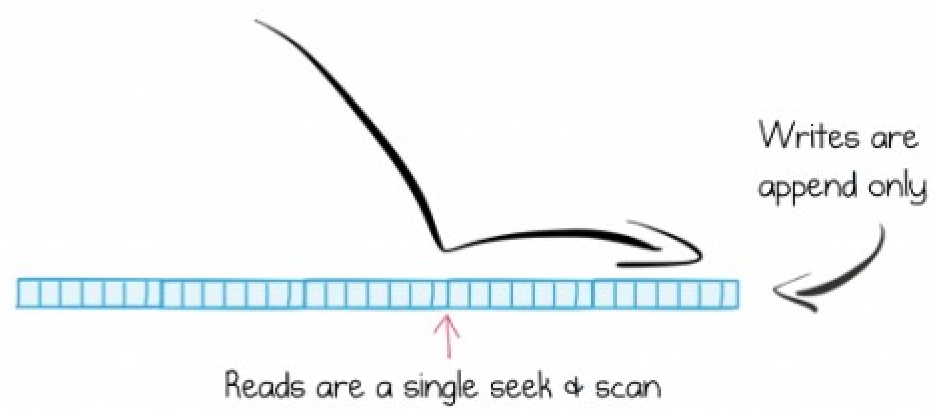

Рисунок 2.2 — Ключевые компоненты стриминговой платформы [1].

Подобный подход с хранением данных в виде журналов имеет интересные последствия. И операции чтения, и операции записи являются последовательными операциями. Это делает предварительную выборку, кеширование проще, а пакетирование сообщений происходит и вовсе естественным образом. Всё это на выходе повышает общую производительность системы. Когда сообщение читается из Kafka, серверу даже нет необходимости импортировать данные в JVM, так как по вышеперечисленным особенностям данные могут быть напрямую скопированы с диска в сетевой буфер.

Также пакетирование и последовательность операций дают возможность хранить данные долгосрочно. Многие традиционные брокеры сообщений построены при помощи хеш-таблиц или B-деревьев, для которых хранение данных на долгие сроки не является самой сильной стороной, по той причине, что данные структуры данных индексируют данные, а поддержание индексов для большого количества сообщений является трудоёмким.

Среди других последствий подхода с журналами является то, что накопление новых сообщений без из регулярного чтения не оказывает существенного влияния на производительность системы в отличие от традиционных брокеров сообщений и очередей. И в конце концов, все эти факторы делают Kafka, подходящей для хранилища событий.

2.3.4 Линейное масштабирование

Kafka в качестве основной структуры данных использует журнал с разделами, однако на самом деле это не один журнал, а их множество, распределённое по различными машинам.

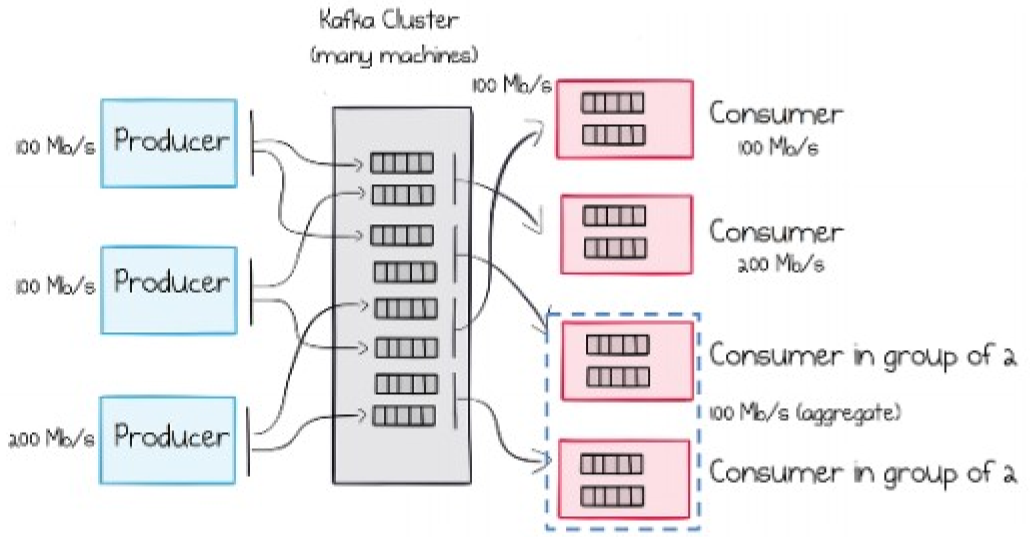

Рисунок 2.3 — Взаимодействие издателей и подписчиков с кластером [1].

Система объединяет их всех вместе, обеспечивая маршрутизацию сообщений, репликацию данных и благоприятную обработку ошибок. Хотя запуск Kafka на одной машине возможен, в подавляющем количестве случаев используются кластеры, которые могут обыкновенно включать в себя от трёх до сотни машин. При совершении операций с данными данные будут писаться или читаться с распределением по всем машинам кластера.

При таком устройстве хранилища масштабирование становится простым — достаточно добавить новые машины и осуществить перебалансировку подписчиков.

2.3.5 Репликация данных

Репликация данных является важнейшей частью устройства Apache Kafka. Именно репликация гарантирует доступность и долговечность данных, когда отдельные части системы приходят в негодность, что является неизбежным.

Как уже упоминалось, сообщения хранятся в темах, темы состоят их нескольких журналов, а каждый журнал в свою очередь имеет несколько реплик. Реплика хранятся на брокерах. Обычно каждый брокер хранит сотни или тысячи реплик, принадлежащих различным темам и журналам.

Различают два вида реплик:

1) Реплика-лидер.

Каждый раздел имеет единственную реплику-лидера. Все запросы на запись и чтения сообщения приходят на реплику-лидера для соблюдения консистентности в порядке сообщений.

2) Реплика-последователь.

Все реплики, не являющиеся лидерами, называются репликами-последователями. Последователи не обрабатывают клиентских запросов. Задача реплик-последователей реплицировать данные с лидера. Если реплика-лидер для раздела пропадает, то происходит избрание новой реплики-лидера среди последователей.

2.4 Издатели и подписчики в Apache Kafka

Каким бы не был вариант использования Apache Kafka, компонентами, которые будут присутствовать всегда в системе, являются издатели и подписчики. Издатели и подписчики могут находится как в разных приложениях, так и в одном. Apache Kafka предоставляет специальное клиентское API для разработки приложений, взаимодействующих с Kafka.

2.4.1 Издатели в Kafka

Существует множество различных вариантов, когда приложению может понадобится записать сообщение в Kafka: запись пользовательской активности, хранение событий, запись показателей метрик, асинхронная коммуникация с другими приложениями.

Разнообразие вариантов использования подразумевает разнообразие требований. В зависимости от поставленной задачи при разработке издателя следует учитывать различные аспекты отправки сообщений. В некоторых ситуациях каждое сообщение является важным. И нельзя позволить их потерять. В некоторых ситуациях приходится дополнительно обрабатывать случаи дублирования сообщений.

Для отправки сообщений в Kafka используется объект класс ProducerRecord, который содержит значение сообщения и тему, в которую следует записать сообщения. Также опционально есть возможность указать ключ и раздел. При отправке сообщения первым делом издатель сериализует ключ и значение в массивы байтов для последующей их передачи по сети. Затем определяется, в какой раздел записать сообщение, если раздел не был указан явно. После того, как определено, куда сообщение следует записать, оно добавляется к набору сообщений, которые будут отправлены в ту же тему и раздел. Отдельный поток ответственен за отправку наборов таких сообщений.

Когда брокер получает сообщения, он возвращает ответ с информацией о том, куда было записано сообщение или ошибку, получив которую издатель может повторить попытку отправки сообщения.

2.4.2 Подписчики в Kafka

Для получения сообщений подписчики осуществляют подписку на темы с соответствующими сообщениями. При работе с подписчиками в Kafka важно понимать понятие групп подписчиков.

Каждый подписчик в Kafka принадлежит некоторой группе подписчиков, принадлежность группе подписчиков конфигурируется при создании подписчика. Когда несколько подписчиков подписаны на одинаковую тему и при этом являются частью одной группы подписчиков, каждый подписчик получает собственное подмножество разделов, из которых ему будут приходить сообщения. Таким образом, каждый подписчик из одной группы подписчиков будет получать собственные сообщения, которые не будут получать другие подписчики из этой группы. Благодаря такому устройству появляется возможность масштабировать систему, добавляя новых подписчиков в группу. Также стоит учитывать, что добавляя в одну группу подписчиков больше, чем количество разделов в теме, лишние подписчики будут простаивать.

2.5 Клиентская библиотека Kafka Streams

Kafka Streams — это клиентская библиотека для обработки и анализа данных, хранящихся в Apache Kafka. Данная библиотека построена на основных принципах и концепциях обработки потоков событий: различие между временем события и временем обработки, поддержка оконных операций, возможность осуществления запросов к приложению с целью узнать его состояние в режиме реального времени и др.

2.5.1 Понятие времени в системах обработки потоков событий

Одним из самых важных аспектов при обработке потоков событий является понятие времени. Ряд операций по обработке данных зависит от понятия времени, например, оконные операции, операции агрегирования данных по предварительно сгруппированным окнам с событиями, обработка событий, пришедших с опозданием.

Различают следующие определения времени:

1) Время события.

Точка во времени, в которую рассматриваемое событие произошло на источнике произошедшего события. Примером может послужить событие изменения местоположения, сообщаемое GPS датчиком в автомобиле, в данном случае временем события будет время, в которое датчик GPS захватил изменение местоположения.

2) Время обработки.

Точка во времени, в которую рассматриваемое событие оказывается обработанным приложением, то есть в момент потребления события. Время обработки может быть позже времени события с разницей как в миллисекунды, так и в часы, дни или больше.

3) Время приёма.

Точка во времени, в которую рассматриваемое событие было помещено в соответствующий раздел на брокере Kafka. Отличие времени приёма от времени события заключается в том, что время события устанавливается на источнике события, в то время как время приёма устанавливается в момент записи события на диск. Отличие времени приёма от времени обработки заключается в том, что время обработки — это время обработки события самим приложением, то есть, если запись не обработана, то для неё не определено время обработки, но определено время приёма.

Для задания конкретной семантики времени для каждого события в Kafka Streams используется интерфейс TimestampExtractor. Извлеченные с помощью данного интерфейса отметки времени описывают прогресс обработки потока событий по отношению ко времени, а также используются операциями, опирающимися на понятие времени, например оконными операциями.

2.5.2 Двойственность потоков данных и таблиц

Реализуя обработку потоков данных на практике, зачастую приходится сталкиваться с тем, что возникает необходимость как в самих потоках данных, так и в базах данных. К примеру, мы разрабатываем приложение электронной торговли, которое обогащает входящий поток событий покупательских транзакций последними данными о покупателях из таблицы в базе данных. Таким образом, нам необходима поддержка для работы с потоками данных и таблицами.

Между таблицами и потоками данных существует близкая взаимосвязь, которую можно называть двойственностью таблиц и потоков данных. Kafka использует эту двойственность, чтобы достичь эластичности приложений, отказоустойчивости и возможности осуществления запросов к приложению в реальном времени.

Двойственность потоков данных и таблиц описывает близкую связь между потоками данных и таблицами:

1) Поток данных как таблица.

Поток данных может быть рассмотрен как история изменений таблицы, где каждая запись в потоке запечатлевает изменение состояния таблицы. Таким образом, поток данных может быть запросто «превращён» в таблицу путём воспроизведения потока с самого начала и до конца.

2) Таблица как поток данных.

Таблица может быть рассмотрена как снимок состояния в определённый момент времени, где каждая строка — это последнее значение для соответствующего ключа в потоке данных. Таким образом, поток данных получается из таблицы путём прохождения по всем парам ключ-значение в таблице.

2.5.3 Операции агрегации событий

Операции агрегации принимают в качестве аргумента один входящий поток событий или таблицу, результатом операции является новая таблица, в которой несколько входящих записей объединены в одну выходящую запись. Простыми примерами таких операций являются операции подсчёта записей определённого типа, суммирование. В Kafka Streams входящим параметром может быть KStream или KTable, а выходящим — KTable. Это позволяет обновлять агрегированное значение с приходом внеочередных записей. Так как выходящее значение является таблицей KTable, новое агрегированное значение переписывает старое значение для того же ключа.

2.5.4 Оконные операции

Оконные операции позволяют контролировать группировку записей, которые имеют одинаковые ключи, для операций с состоянием, таких как операции агрегации или объединения. Окна отслеживаются с использованием ключей записей.

Работая с оконными операциями в Kafka Streams, разработчик может указать период, в течение которого приложение будет ждать записи, пришедшие вне очереди, для конкретного окна. Так же, в зависимости от определения семантики времени для определённого типа записей, понятия внеочередной записи может не существовать по определению, например, в случае выбора времени обработки в качестве семантики времени.

ВЫВОДЫ

1) Рассмотрена история создания и возможные области применения Apache Kafka.
2) Проведён сравнительный анализ Apache Kafka с REST API, базами данных, шинами и классическими очередями.
3) Рассмотрены основные концепции и понятия Apache Kafka, возможности масштабирования с использованием данной платформы.
4) Выявлены преимущества и недостатки Apache Kafka.
5) Рассмотрены основные концепции потоковой обработки событий и клиентская библиотеки Kafka Streams.

Глава 3 ПРОЕКТИРОВАНИЕ РАСПРЕДЕЛЁННОЙ СИСТЕМЫ, УПРАВЛЯЕМОЙ СОБЫТИЯМИ

3.1 Назначение системы

Проектируемая распределённая система предназначена для анализа активности разработчиков с целью слежения за продуктивностью, выявления факторов, влияющих на продуктивность разработчиков. На основании данных, предоставляемых API сервиса GitHub, система ведёт статистику активности разработчиков. Данная система может найти применение в компаниях, занимающихся разработкой программного обеспечения, среднего и крупного размера, которые заинтересованы в слежении и исследовании продуктивности разработчиков и определении влияния активности разработчиков на показатели компании.

3.1.1 Функциональные требования

Разрабатываемая система должна отвечать следующим функциональным требованиям:

1. Регистрация пользователей в системе.
2. Авторизация пользователей в системе.
3. Пользователи должны иметь возможность искать и добавлять репозитории, хранящиеся на платформе хостинга кода GitHub.
4. Для добавленных репозиториев пользователи должны иметь возможность просматривать статистику, собранную системой.
5. Для добавленных репозиториев пользователи должны иметь возможность просматривать статистику, предоставляемую API GitHub.

3.1.2 Нефункциональные требования

Разрабатываемая система должна отвечать следующим нефункциональным требованиям:

1. Наглядный интерфейс.

2. Независимое масштабирование каждого компонента системы.
3. Сохранение потока загруженных записей в хранилище для возможности их повторной обработки.
4. Построение взаимодействия компонент системы, ответственных за обработку запросов пользователя и ответственных за загрузку и обработку потоков данных, через асинхронную межсервисную коммуникацию.

3.2 Общая схема системы

На Рисунке 3.1 представлена общая схема системы. Логически всю систему можно разделить на три части: пользовательское веб-приложение, загрузка данных по заданным источникам (репозиториям), используя API GitHub, и обработка и анализ поступающих событий в реальном времени. Пользователь взаимодействует с системой посредством веб-интерфейса.

Рисунок 3.1 — Общая схема системы.

Например, через веб-интерфейс пользователь может добавлять репозитории и просматривать статистику по добавленным репозиториям. Серверная часть веб-приложения обновляет набор репозиториев, из которых необходимо осуществлять загрузку, а также формирует запросы к компоненту

системы, отвечающему за обработку событий, с целью формирования конечной статистики, отображаемой в веб-интерфейсе. По актуальному набору репозиториев, из который необходимо обрабатывать данные, специальный компонент осуществляют загрузку последней актуальной информации. Загруженные записи обрабатываются отдельно выделенным компонентом, отвечающим за обработку и подсчёт статистики по поступающим записям, которая потом запрашивается серверной частью веб-приложения.

3.3 Высокоуровневая архитектура системы

На рисунке 3.2 представлена высокоуровневая архитектура системы.

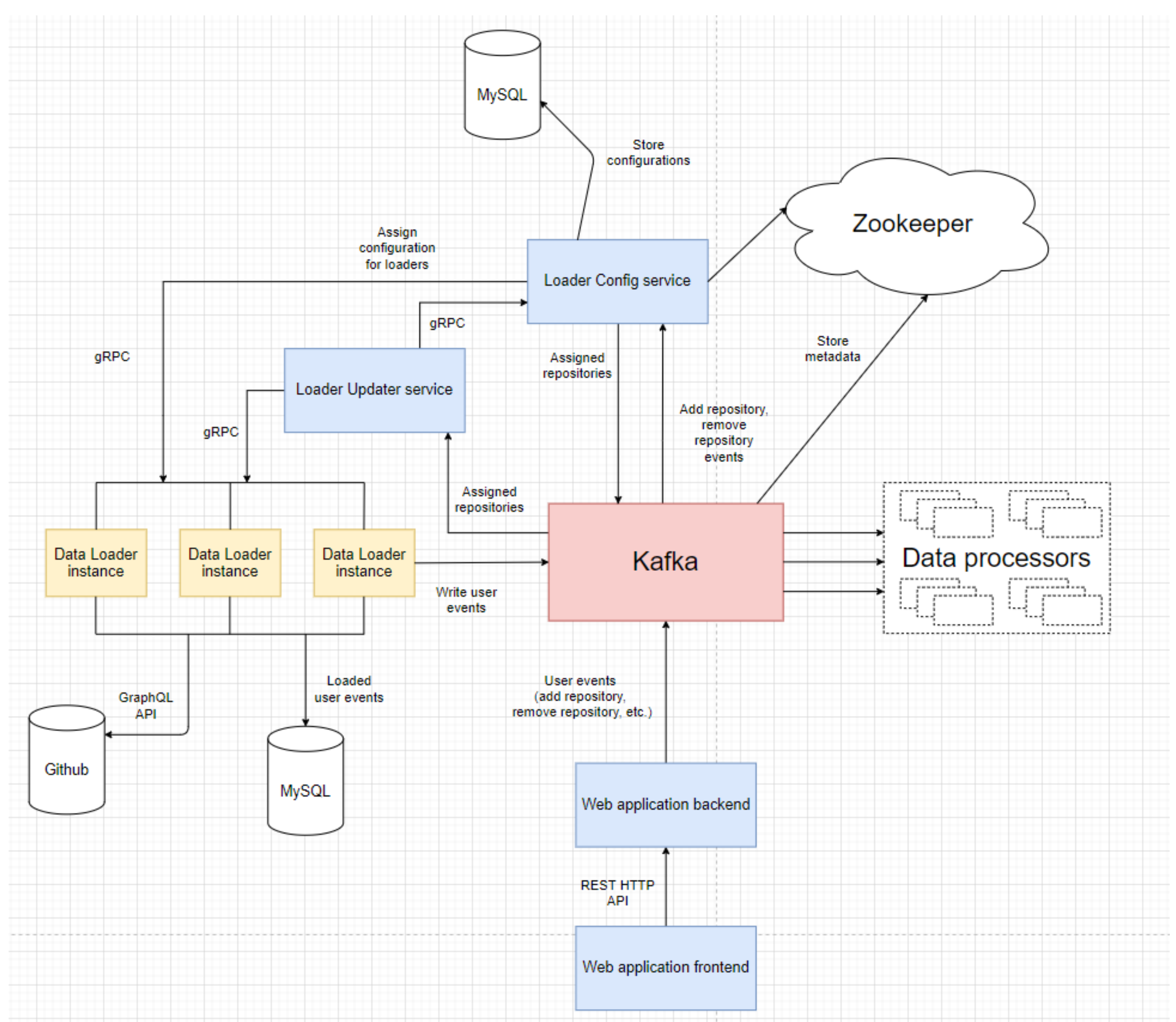

Рисунок 3.2 — Высокоуровневая архитектура системы.

Общая схема взаимодействия определяется потоками данных. Данные, поступающие в Kafka, соответствуют различным событиям, происходящим на платформе хостинга кода GitHub, — фиксациям изменений, запросам на принятие изменений, комментариям, подтверждениям. В качестве издателей выступают сервисы-загрузчики, которые используют GraphQL API (Приложение А, Б), предоставляемое GitHub, выгружают данные на периодической основе.

Загрузчики осуществляют загрузку данных на основе конфигурации, которая определяется динамически в процессе работы системы и может изменяться. Конфигурация представляет из себя набор источников, например, репозиториев, из которых данный загрузчик должен загружать произошедшие события, и обновляется серверной частью веб-приложения.

Распределение конфигурации осуществляется посредством специального сервиса для распределения конфигураций и хранилища ZooKeeper. Загруженные события загрузчики записывают в Apache Kafka. Например, для событий, соответствующим записям изменений в репозиторий, создана тема kamus.commits. Для сериализации сообщений выбрана технология сериализации Protobuf (Приложение Б), которая имеет официальную поддержку для интеграции с Apache Kafka.

Загруженные данные обрабатываются приложениями, использующими библиотеку Kafka Streams. Поступающие события дедуплицируются, затем по потокам дедудуплицированных событий подсчитывается различная статистика, которая доступна другим сервисам через gRPC API.

Пользователь взаимодействует с приложением посредством веб-интерфейса. Серверная часть веб-приложения обрабатывает запросы веб-клиента, хранит информацию о пользователях и репозиториях, за которыми они следят, а также формирует структурированную информацию о репозиториях,

осуществляя RPC запросы к сервисам, подсчитывающим статистику репозиториев. Также серверная часть веб-приложения обновляет конфигурацию активных репозиториев, для которых необходимо загружать информацию. Если пользователь добавляет какой-либо репозиторий или удаляет его, соответствующее сообщение отправляется в тему kamus.track.repository.

3.4 Распределение конфигурации по загрузчикам

Используя веб-интерфейс, пользователи имеют возможность добавлять источники, откуда они хотели бы собирать события. Это могут быть репозитории, профили пользователей, организации. Для разделения возможных источников данных между загрузчиками необходимо предусмотреть механизм распределения конфигурации. При этом конфигурация должна быть равномерно распределена между активными загрузчиками, реагировать на выход из строя некоторых загрузчиков, а также на добавление новых загрузчиков в систему.

3.4.1 ZooKeeper как хранилище состояния распределённых систем

ZooKeeper является централизованным сервисом для хранения конфигурационной информации, разрешения имен, предоставления распределённой синхронизации и других механизмов, необходимых для поддержания распределённых систем. Используя готовое решение в виде ZooKeeper, избавляемся от необходимости разрабатывать собственное решение, необходимости исправления ошибок и необходимостью разрешать проблемы, связанные с состоянием гонки, присущие распределённым системам.

3.4.2 Организация распределения конфигурации

Процесс распределения конфигурации по загрузчикам построен следующим образом (Рисунок 3.3). При запуске каждый загрузчик добавляет себя в ZooKeeper путём создания специального эфемерного узла, который будет существовать только на время соединения клиента с Zookeeper. То есть в

момент отключения определённого загрузчика узел, созданный им, будет уничтожен. Это позволяет следить за активными загрузчиками и распределять источники данных между ними. Для это спроектирован специальный сервис, который следит за эфемерными узлами загрузчиков и равномерно распределяет конфигурацию.

Каждый раз, когда появляется новый загрузчик или удаляется существующий, этот специальный сервис производит балансировку нагрузки между загрузчиками и обновляет их конфигурации. Коммуникация между сервисом конфигурации и загрузчиками происходит с использованием механизма удалённого вызова процедур gRPC (Приложение Д).

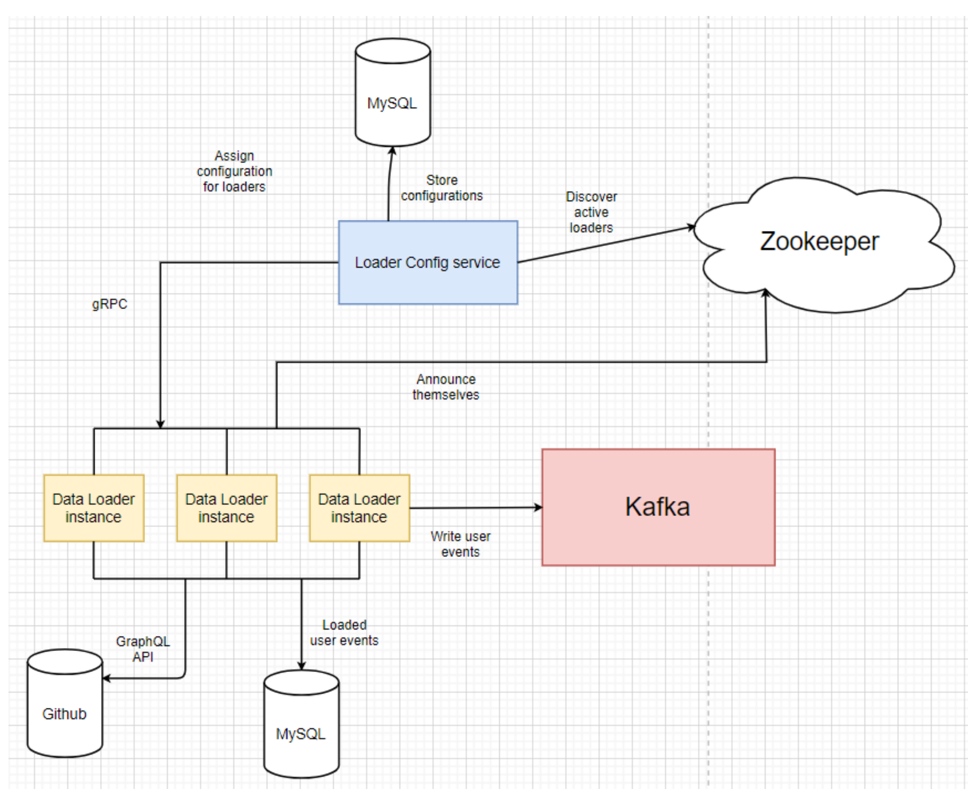

Рисунок 3.3 — Распределение конфигурации по активным загрузчикам.

3.4.3 Структура конфигурации

На самом деле между загрузчиками распределяются не источники напрямую, а корзины, а источники в свою очередь распределяются по корзинам (Рисунок 3.4, Рисунок 3.5). Распределение источников по корзинам происходит, например, в момент чтения события добавления репозитория сервисом

конфигурации, если данный репозиторий не был распределён до этого. В момент распределения очередной пачки источников определяется корзина, имеющая наименьшее число источников.

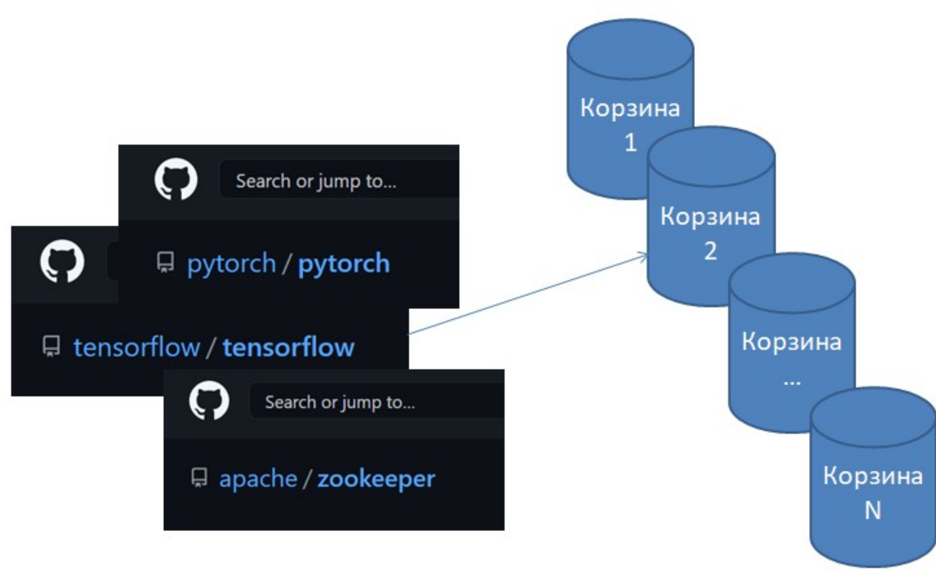

Рисунок 3.4 — Распределение пачек репозиториев по корзинам.

Распределение источников по корзинам происходит в пачках, что позволяет экономить на операции определения наименее загруженной корзины и распределять источники более эффективно.

Частой операцией к данной таблице является чтение всех источников, например, репозиториев для определённой корзины, то мы устанавливаем индекс по соответствующей колонке.

```
mysql> describe tracked_repository;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| name      | varchar(255) | NO   | PRI | NULL    |       |
| owner     | varchar(255) | NO   | PRI | NULL    |       |
| bucket_id | int          | NO   | MUL | NULL    |       |
| metric    | int          | NO   |     | NULL    |       |
| tracked   | bit(1)       | NO   |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

Рисунок 3.5 — Схема таблицы, хранящей распределение репозиториев по корзинам.

3.4.4 Обновление конфигурации загрузчиков

В момент распределения корзин по загрузчиками сервис конфигурации задает текущую конфигурации загрузки для каждого загрузчика. Однако в процессе функционирования системы могут добавляться новые источники, и нам нужен способ постепенно обновлять загрузчики источниками, относящимся к соответствующим корзинам.

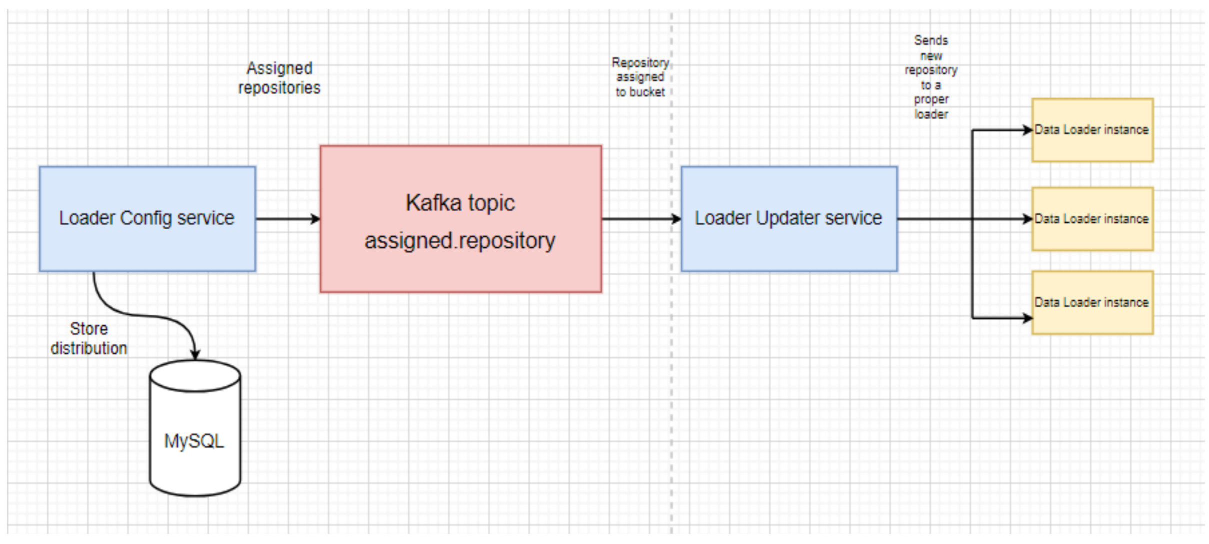

Рисунок 3.6 — Обновление конфигурации загрузчиков.

Ответственность за такое обновление загрузчиков лежит на отдельном сервисе (Loader Updater service), который по сути представляет из себя Kafka подписчика (Рисунок 3.6). Данный сервис получает события о том, что источник распределен в какую-либо корзину. Затем он обновляет соответствующего загрузчика, в конфигурации которого входит данная корзина.

Такое разбиение на отдельные сервисы позволяет независимо масштабироваться сервису конфигурации и сервису обновления загрузчиков, а также получить другие преимущества разъединения системы на отдельные и независимые компоненты, связанные посредством Kafka.

3.5 Обработка потоков событий и подсчёт статистики

3.5.1 Сервис обработки и анализа потоков событий репозиториев

События, соответствующие некоторым активностям разработчиков в репозиториях, помещаются в соответствующие темы Kafka. Непосредственно обработкой потоков событий и подсчётом статистики занимается сервис анализа репозиториев, который подписывается на необходимые темы, например темы фиксаций изменений в репозиториях или запросов на слияние. Apache Kafka позволяет сделать параллельной обработку событий по данным, используя механизм разделов. Каждому экземпляру приложения будет назначен свой набор разделов, записи из которого данный экземпляр будет обрабатывать.

Для обеспечения корректности подсчёта статистики необходимо осуществить дедупликацию поступающих событий с целью устранения дубликатов, которые могут появляться, когда загрузчик теряет соединение с частью системы и не имеет актуальную конфигурацию, используя которую ему необходимо осуществлять загрузку. В таком случае складывается ситуация, при которой два загрузчика имеют в конфигурациях одни и те же источники и загружают одинаковые события в Kafka.

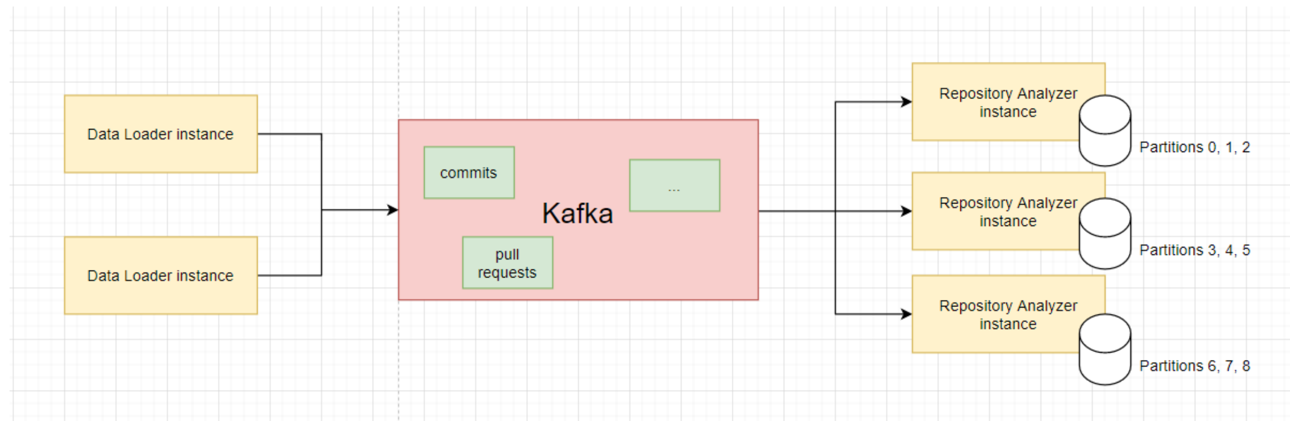

Рисунок 3.7 — Загрузка событий репозиториев и их обработка.

3.5.2 Получение информации по определённому репозиторию

Для отображения информации по репозиториям конечным пользователям в веб-интерфейсе серверная часть веб-приложения делает запросы к сервису обработки и анализа потоков событий репозиториев. Kafka устроена таким образом, что данные распределяются по разделам, при этом мы можем для каждого типа события задать ключ, по которому будет считаться хеш и определяться конкретный раздел, куда будет распределена очередная запись. В свою очередь разделы распределяются между всеми активными подписчиками. Таким образом, каждый экземпляр сервиса, подсчитывающего статистику, в какой-то момент времени владеет статистикой для определённого набора репозиториев в зависимости от того, какие разделы ему назначены. Для того чтобы получить статистику по интересующему репозиторию, необходимо определять, какой экземпляр сервиса хранит информацию о данном репозитории, и делать запрос непосредственно к нему.

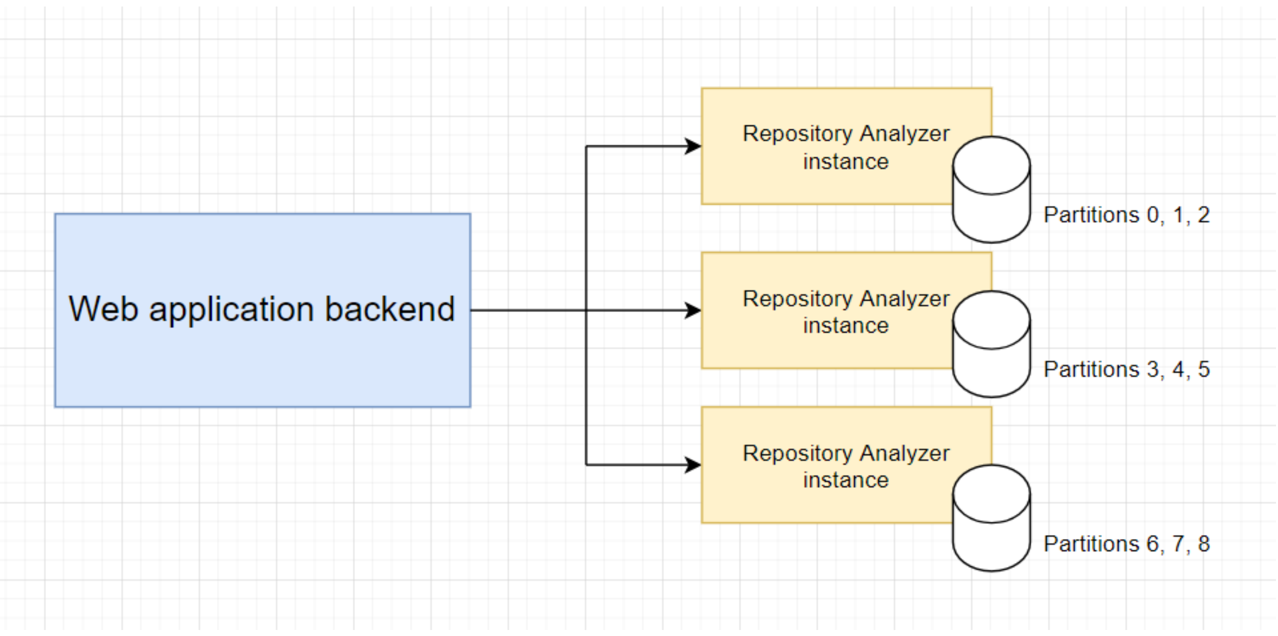

Рисунок 3.8 — Осуществление запросов к сервису подсчёта статистики.

ВЫВОДЫ

1) Определено предназначение разрабатываемой системы.
2) Выявлен набор функциональных и нефункциональных требований к разрабатываемой системе.
3) Спроектирован механизм распределения конфигурации между загрузчиками.
4) Спроектированы компоненты системы, отвечающие за обработку потоков событий.
5) Спроектирована распределённая система, управляемая событиями, соответствующая назначению и отвечающая поставленным функциональным и нефункциональным требованиям.

Глава 4 РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОЙ СИСТЕМЫ, УПРАВЛЯЕМОЙ СОБЫТИЯМИ

4.1 Решение проблемы синхронизации записи в базу данных и платформу Apache Kafka

4.1.1 Необходимость использования базы данных

Загрузчики загружают данные из GitHub путём периодического опрашивания. Таким образом имеется необходимость хранить информацию о том, какие данные были выгружены последними для каждого источника данных, например, последняя фиксация изменения для репозитория. При этом данная информация должна быть общей для нескольких загрузчиков на тот случай, когда сначала один загрузчик загружает данные из репозитория, а затем конфигурации изменяются таким образом, что за данный репозиторий становится ответственным другой загрузчик. Также для реализации данной функциональности необходима поддержка ACID транзакций.

4.1.2 Проблема синхронизации записи в базу данных и Apache Kafka

При использовании базы данных и Apache Kafka одновременно появляется проблема синхронизации записи. Представим ситуацию, в которой загрузчик загрузил последние фиксации изменений для некоторого репозитория. Далее ему предстоит совершить два действия: 1) записать последнюю фиксацию изменения в базу данных, чтобы понимать с какого момента продолжать загрузку; 2) непосредственно отправить новые события в Apache Kafka. Если произойдет только первое действие, а новые события по какой-либо причине не запишутся в Apache Kafka, то мы потеряем эти события. Если произойдёт только второе событие, а первое закончится не успешно, это приведёт к дублированию событий. Для того чтобы избежать данных ситуаций, необходимо предусмотреть механизм синхронизации записи.

```
mysql> describe latest_commit;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| name        | varchar(255) | NO   | PRI | NULL    |       |
| owner       | varchar(255) | NO   | PRI | NULL    |       |
| commit_date | datetime(6)  | YES  |     | NULL    |       |
| sha         | varchar(255) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

Рисунок 4.1 — Схема таблицы, хранящей последние фиксации изменения для каждого репозитория.

4.1.3 Решение проблемы синхронизации записи в базу данных и Apache Kafka

Для решения данной проблемы используется следующая техника. СУБД MySQL предоставляет функциональность, называемую бинарным журналом. Мы можем сконфигурировать сервер базы данных таким образом, чтобы любое изменение, происходящее с таблицами и их содержимым, записывалось в данный журнал. Например, классическим использованием данного журнала является передача данных от ведущих серверов MySQL к ведомым. Этот же журнал может использоваться для синхронизации записи в базу данных и Apache Kafka. Для этого в рамках одной транзакции записываем в базу данных как последнее событие (например, последнюю фиксацию изменения в репозитории), начиная с которого будем продолжать дальнейшую загрузку данных, так и все события, загруженные за данный опрос. Затем мы используем бинарный журнал для получения событий, загруженных за данный опрос, для отправки их в Apache Kafka. Таким образом, мы уверены в атомарности записи последнего события и соответствующих новых загруженных событий.

4.2 Обработка и анализ потоков событий деятельности разработчиков с использованием библиотеки Kafka Streams

Перед непосредственно обработкой поступающих событий по соответствующей приложению бизнес-логике необходимо произвести устранение дубликатов событий, например, устранение нескольких событий соответствующих одной и той же фиксации изменений в репозитории. Причины возникновения таких дубликатов описаны в предыдущей главе.

Для устранения дубликатов нам необходимо уметь идентифицировать события, например, для фиксации изменений это может быть хеш-код, который присваивается каждому изменению. Далее нам необходимо хранить идентификаторы пришедших событий, при поступлении нового события, мы проверяем по идентификатору приходило ли такое событие ранее. Если данное событие приходило, то мы исключаем его из потока событий, в противном случае добавляем его идентификатор в хранилище.

Для гарантированного устранения дубликатов мы должны хранить идентификаторы в постоянном хранилище. Таким образом, даже после осуществления ребалансировки подписчиков различные экземпляры приложения смогут корректно обрабатывать дубликаты событий.

Также исходя из семантики события, нам нужно настроить время, по истечению которого необходимо удалять идентификаторы из хранилища, с целью экономия места на диске. В нашем случае дубликаты появляются по причине рассогласования загрузчиков, а значит, что для определения времени хранения идентификаторов, нам стоит оценить продолжительность возможных неполадок в системе. Также имеет смысл заложить определённое количество времени на сохранение возможности истории сообщений для целей анализа происшествий и отладки. Разумным периодом хранения идентификаторов событий будет 1-3 дня.

Клиентская библиотека Kafka Streams позволяет решить эту задачу при помощи оконных хранилищ (WindowStore) и трансформаций (ValueTransformerWithKey). Мы создаём оконное хранилище, в котором идентификаторы событий будут хранится 1-3 дня, продолжительность окна равна продолжительности хранения идентификаторов. Затем мы используем данное хранилище в специальной трансформации, которая проверяет приходило ли событие с данным идентификатором ранее, и либо игнорирует его, либо добавляет его идентификатор в хранилище. Реализация дедуплицирующей трансформации приведена в Приложении И.

4.3 Реализация возможности осуществления запросов к приложению Kafka Streams

4.3.1 Распределённое хранение состояния между экземплярами приложения Kafka Streams

Полное состояние приложения распределено между всеми экземплярами приложения и между несколькими хранилищами состояний (State Stores), которые управляются некоторыми экземплярами приложения (Рисунок 4.2). Записи в потоках событий и хранилищах распределяются по некоторому ключу, в данном случае по идентификатору репозитория. Это позволяет масштабировать приложение и обрабатывать большое количество данных. Такая поддержка обеспечивается Apache Kafka и библиотекой Kafka Streams.

Экземпляры приложения, задающие топологию обработки событий, используя библиотеку Kafka Streams, могут делать запросы к хранилищам с целью получения текущего состояния, при этом доступ к хранилищам возможен только для чтения. Экземпляры приложения могут осуществлять запросы только к тем хранилищам, которые им назначены и управляются ими. Для того чтобы получить часть состояния, находящуюся в хранилище, которое управляется другим экземпляром приложения, необходимо иметь возможность решать две задачи:

1) По заданному ключу определять, какой экземпляр приложения управляет хранилищем, в котором находится нужные данные, и может их предоставить.
2) Иметь механизм коммуникации между экземплярами приложения непосредственно для передачи необходимых данных.

Первая задача решается библиотекой Kafka Streams — каждый экземпляр приложения имеет информацию обо всех остальных экземплярах и о том, какие разделы потоков данных и хранилища им назначены. Вторая задача должна решаться разработчиком, обычно добавлением RPC слоя в приложение.

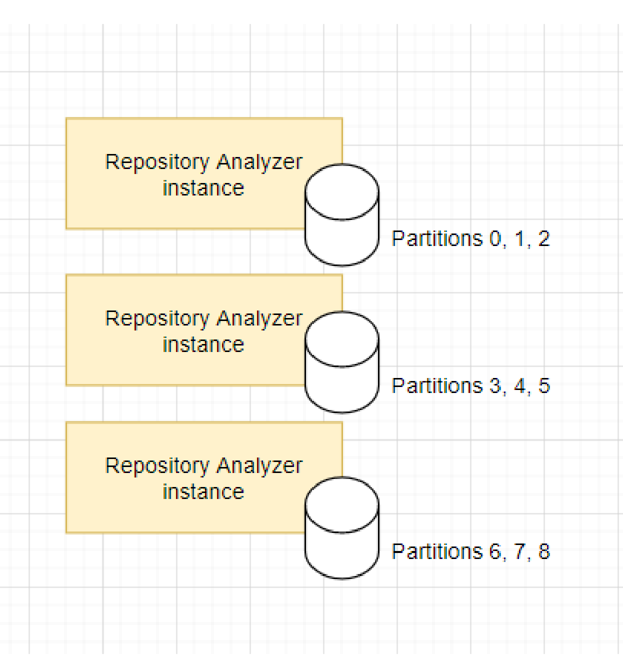

Рисунок 4.2 — Распределение состояния между экземплярами приложения.

4.3.2 Реализация механизма осуществления запросов

Для полноценной поддержки возможности осуществления запросов к сервису обработки событий и подсчёта статистики необходимо реализовать механизм коммуникации между экземплярами приложения Kafka Streams. Для того чтобы добиться этого, необходимо добавить слой RPC коммуникации между экземплярами приложения. Каждый экземпляр Kafka Streams приложения должен предоставлять gRPC API, при этом экземпляры приложения используют gRPC клиенты для взаимодействия между собой.

Основной задачей, которую необходимо решить при построении такой коммуникации, является задача интеграции механизмов разрешения имён и балансирования нагрузки gRPC с жизненным циклом приложения Kafka Streams и использованием метаинформации Kafka Streams приложения для определения активных экземпляров и определения конкретного экземпляра, управляющего хранилищем, содержащим необходимые данные. Для реализации вышеперечисленных задач была разработана вспомогательная библиотека, реализующая основные интерфейсы, предоставляемые библиотекой gRPC, которые позволяют расширять её возможности.

4.4 Реализованное веб-приложение с использованием фреймворка React

С целью демонстрации реализованной распределённой системы было разработано веб-приложение с использованием фреймворка React.

Используя веб-интерфейс пользователь имеет возможность добавлять репозитории и просматривать статистику для добавленных им репозиториев.

На странице добавления репозиториев пользователь может воспользоваться поиском по репозиториям GitHub и добавить набор репозиториев, за которыми он хочет следить (Рисунок 4.3, Рисунок 4.4).

На главной странице пользователь может просматривать список добавленных репозиториев с их краткими описаниями (Рисунок 4.5).

С главной страницы пользователь имеет возможность перейти на страницу конкретного репозитория и просмотреть его статистику, включающую наиболее активных разработчиков, статистику фиксации изменений и запросов на слияние (Рисунок 4.6).

Рисунок 4.3 — Поиск репозиториев.

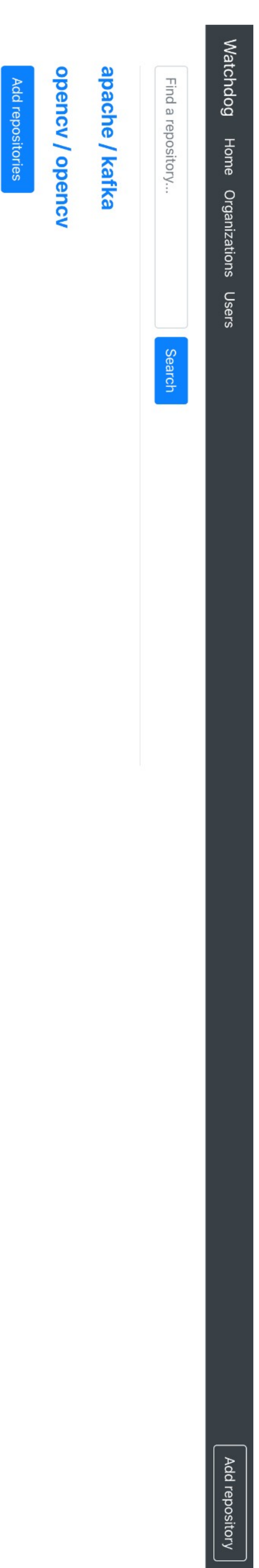

Рисунок 4.4 — Добавление репозиториев.

Watchdog Home Organizations Users

Add repository

Find a repository...

microsoft / calculator

Windows Calculator: A simple yet powerful calculator that ships with Windows

C++ Updated 09 May 2021 ☆ 21964

grpc / grpc-java

The Java gRPC implementation. HTTP/2 based RPC

Java Updated 09 May 2021 ☆ 8649

apache / kafka

Mirror of Apache Kafka

Java Updated 09 May 2021 ☆ 18801

KirillZhelt / Kamus

Java Updated 04 April 2021 ☆ 0

KirillZhelt / MayMayMay

University course work

Kotlin Updated 29 November 2020 ☆ 0

opencv / opencv

Open Source Computer Vision Library

Рисунок 4.5 — Добавленные репозитория пользователя.

Watchdog Home Organizations Users

Add repository

microsoft / calculator

View on Github

Windows Calculator: A simple yet powerful calculator that ships with Windows

C++ Updated 09 May 2021 ☆ 21964

Commit Stats

Total number: 652

Last 30 days: 3

Last week: 1

Today: 0

Pull Request Stats

Total number: 762

Last 30 days: 7

Last week: 2

Today: 0

Top contributors

rudyhuyn #1
112 commits 17826 ++ 16959 --

mcooley #2
99 commits 99894 ++ 45352 --

joseartrivera #3
91 commits 19041 ++ 20104 --

sanderl #4
77 commits 26266 ++ 22323 --

janisozaur #5
31 commits 1028 ++ 865 --

Рисунок 4.6 — Страница репозитория.

4.5 Демонстрация работы системы

Данный раздел демонстрирует полный цикл работы системы с того момента, как пользователь добавляет некоторый репозиторий в веб-интерфейсе и до момента, когда пользователь может просматривать статистику, собранную по добавленному репозиторию.

4.5.1 Добавление нового репозитория

Пользователь добавляет репозиторий в веб-интерфейсе. После нажатия на кнопку добавления веб-клиент посылает HTTP запрос серверной части веб-приложения. Серверная часть веб-приложения, получив запрос на добавление репозитория и выполняет две операции:

1) Посылает событие, соответствующее добавлению нового репозитория в Apache Kafka в тему kamus.track.repository.
2) Сохраняет репозиторий в базу данных для пользователя, авторизованного в системе.

После успешного выполнения этих двух действий сервер возвращает успешный HTTP ответ с кодом 201.

Сервис распределения конфигурации (Loader Config сервис), отвечающий за распределение конфигурации между загрузчиками, подписан на события из темы kamus.track.repository. Получив сообщение о добавление репозитория, сервис проверяет, не был ли добавлен данный репозиторий ранее. Если репозиторий оказывается новым, то сервис находит корзину с наименьшим количеством репоизториев и отправляет туда новый репозиторий. Затем сервис отправляет новое событие в тему kamus.assigned.repository (Рисунок 4.7).

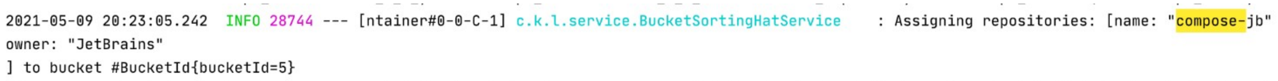

Рисунок 4.7 — Распределение репозитория в корзину.

Сервис обновления конфигурации загрузчиков (Loader Updater сервис) читает события из темы kamus.assigned.repository и обновляет конфигурацию

загрузчика, который загружает данные для корзины, к которой принадлежит новый репозиторий, осуществляя gRPC вызов (Рисунок 4.8).

```
2021-05-09 20:23:05.529  INFO 28727 --- [ntainer#0-0-C-1] c.k.l.u.k.AssignedRepositoriesConsumer   : Processing repository {
  name: "compose-jb"
  owner: "JetBrains"
}
bucketId: 5
```

Рисунок 4.8 — Обновление конфигурации загрузчика.

4.5.2 Загрузка и обработка потока событий для добавленного репозитория

Обновив свою конфигурацию загрузчик начинает загружать данные для нового репозитория. Загруженные данные загрузчик помещает в несколько тем в Apache Kafka таких как kamus.commits, kamus.pull.requests (Рисунок 4.9).

```
2021-05-09 20:23:08.539  INFO 28726 --- [.github.com/...] c.k.d.service.GithubDataLoaderService    : Loaded 100 pull requests for repository name: "compose-jb"
owner: "JetBrains"
. Total 102
2021-05-09 20:23:09.049  INFO 28726 --- [.github.com/...] c.k.d.service.GithubDataLoaderService    : Loaded 69 pull requests for repository name: "compose-jb"
owner: "JetBrains"
. Total 171
```

Рисунок 4.9 — Загрузка данных для добавленного репозитория.

Сервис подсчёта статистики подписан на темы kamus.commits, kamus.pull.requests и обрабатывает потоки событий, подсчитывая статистику для репозиториев (Рисунок 4.10). При этом текущие значения подсчитанных величин можно получить путём осуществления запроса к gRPC API сервиса.

```
2021-05-09 20:23:09.136  INFO 28684 --- [-StreamThread-1] c.k.c.t.s.PullRequestsLoggingStream      : Consumed pullRequest {
  repository {
    name: "compose-jb"
    owner: "JetBrains"
  }
  title: "Desktop application plugin"
  number: 33
  authorName: "AlexeyTsvetkov"
  createdAt: "2020-10-27T13:28:16Z"
  state: MERGED
}
```

Рисунок 4.10 — Обработка потоков событий добавленного репозитория.

4.5.3 Просмотр результатов обработки данных

Когда пользователь заходит на страницу добавленного им репозитория с целью просмотра статистики по этому репозиторию, серверная часть веб-

приложения делает gRPC запрос к сервису подсчёта статистики с целью получения текущих значений подсчитанных величин и подготовки статистики к отображению в веб-интерфейсе (Рисунок 4.11).

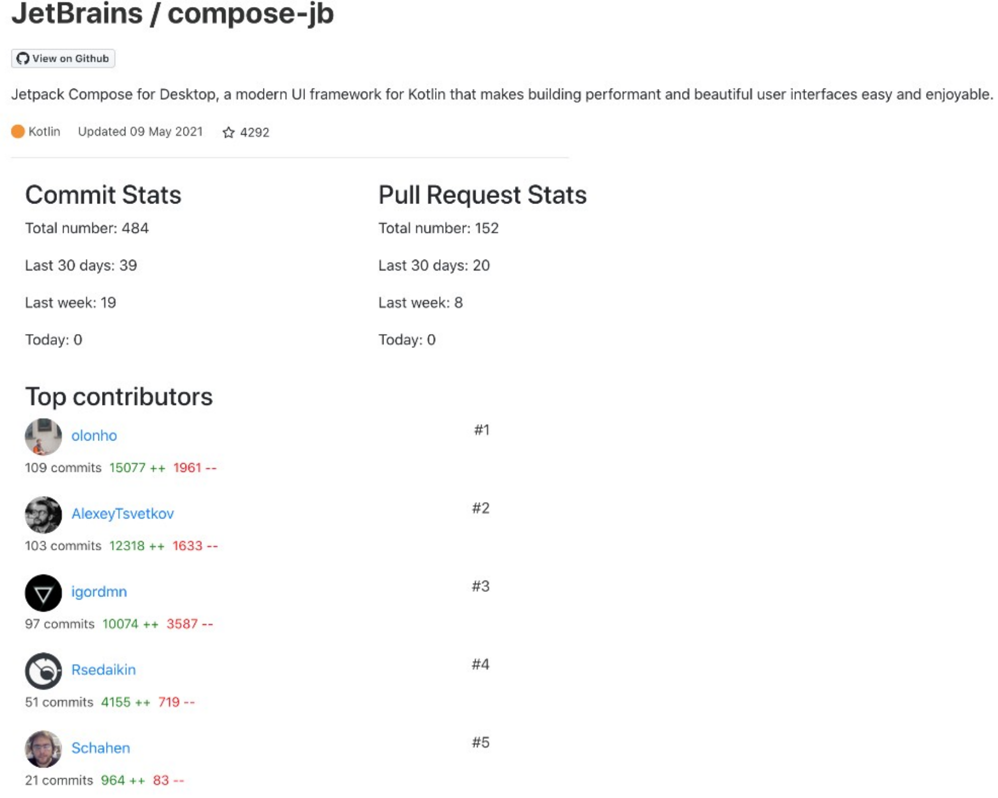

Рисунок 4.11 — Подсчитанная статистика для добавленного репозитория.

ВЫВОДЫ

1) Решена задача распределения конфигурации по загрузчикам.
2) Решена задача синхронизации записи в базу данных MySQL и платформу Apache Kafka с использованием функций предоставляемых MySQL — транзакций и бинарного журнала.
3) Реализована обработка потоков событий с использованием библиотеки Kafka Streams с дедуплицированием событий и возможности осуществления запросов к текущему состоянию приложения.
4) Реализована распределённая система, управляемая событиями, соответствующая назначению и отвечающая поставленным функциональным и нефункциональным требованиям.

5) Продемонстрирована работа полного цикла распределённой системы с примерами взаимодействия различных компонент.

ЗАКЛЮЧЕНИЕ

К современным сервисам и приложениям предъявляются высокие требования в отношении их производительности, способности обрабатывать большое количество запросов одновременно. Для того чтобы удовлетворить данным требованиям, мы должны подходить к вопросу архитектуры соответствующим образом. Микросервисная архитектура позволяет разрабатывать приложения и сервисы, которые легко развёртывать и масштабировать. Для коммуникации сервисов в микросервисной архитектуре хорошо подходят технологии удалённого вызова процедур. Дальнейшее совершенствование микросервисной архитектуры приводит к системам, управляемым событиями.

В работе рассмотрены микросервисная архитектура, технология удалённого вызова процедур gRPC в контексте построения распределённых систем, обработка потоков событий с использованием библиотеки Kafka Streams, разработка распределённых систем, управляемых событиями, с использованием платформы Apache Kafka.

Спроектированная и реализованная в ходе работы распределённая система, управляемая событиями, предоставляет возможности для анализа активности разработчиков.

В ходе выполнения работы были рассмотрены и решены следующие задачи:

1) Рассмотрены понятия микросервисной архитектуры и её устройство.
2) Проведён обзор основных концепций технологий удалённого вызова процедур на основе gRPC.
3) Рассмотрены основные концепции и понятия Apache Kafka,
   возможности масштабирования с использованием данной платформы,
   используя модель

4) Рассмотрены основные концепции потоковой обработки событий и клиентская библиотеки Kafka Streams.
5) Спроектирована распределённая система для анализа активности разработчиков.
6) Решена задача синхронизированной записи в базу данных и платформу Apache Kafka.
7) Реализована обработка потоков событий с использованием библиотеки Kafka Streams с дедуплицированием событий и возможности осуществления запросов к текущему состоянию приложения.
8) Разработана распределённая система для анализа активности разработчиков.

Для реализации приложения использовалась среда разработки IntelliJ IDEA и редактор кода Microsoft Visual Studio Code.

Результаты исследования представлены на 78-ой Научной конференции студентов и аспирантов БГУ.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Narkhede, N. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale / N. Narkhede, Gwen Shapira, Todd Palino. — Sebastopol: O'Reilly Media, 2017. — 493 p.

2. Stopford, B. Designing Event-Driven Systems / B. Stopford. — Sebastopol: O'Reilly Media, 2018. — 166 p.

3. Documentation - gRPC [Electronic resource] / gRPC — Mode of access: https://grpc.io/docs/. — Date of access: 18.11.2020.

4. What are microservices? [Electronic resource] / Microservice Architecture — Mode of access: https://microservices.io/. — Date of access: 20.11.2020.

5. Apache Kafka documentation [Electronic resource] / Apache Kafka — Mode of access: https://kafka.apache.org/documentation/. — Date of access: 08.12.2020.

6. ReactiveX documentation [Electronic resource] / ReactiveX — Mode of access: http://reactivex.io/intro.html/. — Date of access: 08.12.2020.

7. Streaming 101: The world beyong batch [Electronic resource] / O'Reilly — Mode of access: https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/. — Data of access: 09.05.2021.

ПРИЛОЖЕНИЯ

Приложение А

```
query GetCommitsPaginatedWithUntil($owner: String!, $name: String!, $after:
GitObjectID!, $afterCursor: String!,

   $commitsPerPage: Int!, $latestCommitDate: GitTimestamp!) {

   repository(owner: $owner, name: $name) {

       object(oid: $after) {

           ... on Commit {

               history(first: $commitsPerPage, after: $afterCursor, since:
$latestCommitDate) {

                   edges {

                       node {

                           ... on Commit {

                               author {

                                   email

                                   name

                               }

                               additions

                               deletions

                               changedFiles

                               committedDate

                               oid

                           }

                       }

                   }

                   pageInfo {

                       endCursor

                       hasNextPage
```

```
                        }
                    }
                }
            }
        }
    }
```

```
private Observable<List<Commit>> getCommitsPaginatedWithUntil(Repository
repository, String rootCommitOid, String afterCursor, int commitsPerPage,
LatestCommit untilCommit) {

   return githubTemplate.queryCall(new
GetCommitsPaginatedWithUntilQuery(repository.getOwner(), repository.getName(),

           rootCommitOid, afterCursor, commitsPerPage,
untilCommit.getCommitDate()))

                  .flatMap(data -> {

                      GetCommitsPaginatedWithUntilQuery.AsCommit rootCommit =
(GetCommitsPaginatedWithUntilQuery.AsCommit) data.repository().object();



                      List<Commit> commits =
rootCommit.history().edges().stream()

                                                          .map(edge ->
GraphQL2ProtobufConverter.fromNode(repository, edge.node()))

                                                          .collect(Collectors.toList(
));



                      if (rootCommit.history().pageInfo().hasNextPage()) {

                          return Observable.just(commits)

                                              .zipWith(getCommitsPaginatedWithUntil(r
epository, rootCommitOid, rootCommit.history().pageInfo().endCursor(),
commitsPerPage, untilCommit),

                                                      (l, r) ->
Stream.concat(l.stream(), r.stream()).collect(Collectors.toList()));

                      } else {

                          commits.removeIf(commit ->
commit.getSha().equals(untilCommit.getSha()));

                          return Observable.just(commits);

                      }

                  });

}
```

```
syntax = "proto3";

package com.kamus.common.grpcjava;

option java_multiple_files = true;
option java_package = "com.kamus.common.grpcjava";

message Repository {
 string name = 1;
 string owner = 2; // The login field of a user or organization.
}

message CommitStats {
 int32 additions = 1;
 int32 deletions = 2;
 int32 changedFiles = 3;
}

message Commit {
 Repository repository = 1;
 string authorName = 2;
 string authorEmail = 3;
 string commitDate = 4;
 string sha = 5;
 CommitStats stats = 6;
}
```

Приложение Г

```
syntax = "proto3";

import "com/kamus/common/GitEntities.proto";

package com.kamus.loader.config.grpcjava;

option java_multiple_files = true;
option java_package = "com.kamus.loader.config.grpcjava";

message LoaderConfiguration {
 repeated com.kamus.common.grpcjava.Repository repository = 1;
}

message AddRepositoryRequest {
 com.kamus.common.grpcjava.Repository repository = 1;
}

message AddRepositoryResponse {
 bool added = 1; // true if added, false if already existed
}

message RemoveRepositoryRequest {
 com.kamus.common.grpcjava.Repository repository = 1;
}
```

```
message RemoveRepositoryResponse {
 bool removed = 1; // true if removed, false if doesn't existed
}

message GetRepositoriesRequest {

}

message GetRepositoriesResponse {
 repeated com.kamus.common.grpcjava.Repository repository = 1;
}

message GetLoaderConfigurationRequest {
 string configurationId = 1;
}

message GetLoaderConfigurationResponse {
 LoaderConfiguration loaderConfiguration = 1;
}

service LoaderConfigurationService {
 rpc addRepository(AddRepositoryRequest) returns (AddRepositoryResponse);
 rpc removeRepository(RemoveRepositoryRequest) returns
(RemoveRepositoryResponse);
 rpc getRepositories(GetRepositoriesRequest) returns (GetRepositoriesResponse);

 rpc getLoaderConfiguration(GetLoaderConfigurationRequest) returns
(GetLoaderConfigurationResponse);
}
```

Приложение Д

```
@Component
public class AssignedRepositoriesConsumer {

   private static final Logger logger =
LoggerFactory.getLogger(AssignedRepositoriesConsumer.class);

   private final LoaderConfigurationService loaderConfigurationService;
   private final LoadersUpdaterService loadersUpdaterService;

   public AssignedRepositoriesConsumer(LoaderConfigurationService
loaderConfigurationService, LoadersUpdaterService loadersUpdaterService) {
       this.loaderConfigurationService = loaderConfigurationService;
       this.loadersUpdaterService = loadersUpdaterService;
   }

   @KafkaListener(topics = "assigned.repositories", containerFactory =
"assignedRepositoriesListenerContainerFactory")
   public void processAssignedRepository(@Payload AssignedRepository
assignedRepository) {
       logger.info("Processing {}", assignedRepository);

       try {
           Loader loader =
                   loaderConfigurationService.getLoaderForBucket(assignedReposit
ory.getBucketId());
           updateLoader(loader, assignedRepository);
       } catch (NoActiveLoaderException e) {
           logger.info("No active loader for {}", assignedRepository);
       }
   }

   private void updateLoader(Loader loader, AssignedRepository
assignedRepository) {
       try {
           loadersUpdaterService.updateLoaderWithRepository(loader,
assignedRepository);
       } catch (Exception e) {
           logger.error("Wasn't able to update loader {} with the repository {}
because of {}", loader, assignedRepository, e);
       }
   }

}
```

```
@Component
public class GrpcServerRunner implements CommandLineRunner, DisposableBean {

   private static final Logger logger =
LoggerFactory.getLogger(GrpcServerRunner.class);

   private static final String AWAIT_THREAD_NAME = "grpc-server-await-thread";

   private static final long SHUTDOWN_GRACE = 5;
   private static final TimeUnit SHUTDOWN_GRACE_UNIT = TimeUnit.SECONDS;

   private final Server grpcServer;
   private final CountDownLatch latch;

   private final ApplicationEventPublisher eventPublisher;

   public GrpcServerRunner(@Nonnull Server grpcServer) {
       this(grpcServer, null);
   }

   public GrpcServerRunner(@Nonnull Server grpcServer, @Nullable
ApplicationEventPublisher eventPublisher) {
       Preconditions.checkNotNull(grpcServer);

       this.grpcServer = grpcServer;
       this.latch = new CountDownLatch(1);

       this.eventPublisher = eventPublisher;
   }

   @Override
   public void run(String... args) throws Exception {
       grpcServer.start();
       logger.info("gRPC server is started!");

       if (Objects.nonNull(eventPublisher)) {
           eventPublisher.publishEvent(new GrpcServerStartedEvent(this));
       }

       startAwaitThread();
   }

   @Override
   public void destroy() throws Exception {
       logger.info("Shutting down gRPC server...");

       try {
           grpcServer.shutdown();
           grpcServer.awaitTermination(SHUTDOWN_GRACE, SHUTDOWN_GRACE_UNIT);
       } finally {
           latch.countDown();
       }

       logger.info("gRPC server is stopped.");
   }

   private void startAwaitThread() {
```

```
        Thread awaitThread = new Thread(() -> {
            try {
                latch.await();

                logger.info(AWAIT_THREAD_NAME + " is terminating.");
            } catch (InterruptedException ex) {
                logger.info(AWAIT_THREAD_NAME + " was interrupted.");
                Thread.currentThread().interrupt();
            }
        }, AWAIT_THREAD_NAME);

        awaitThread.start();
    }

}
```

```
@Component
public class DataLoadersCoordinator implements LoadersChangeAware {

   private static final Logger logger =
LoggerFactory.getLogger(DataLoadersCoordinator.class);

   private final DataLoaderStubFactory stubFactory;
   private final BucketsDistributor bucketsDistributor;
   private final LoaderUpdater loaderUpdater;

   private final Set<Loader> activeLoaders = new HashSet<>();
   private final Map<LoaderId, DataLoaderServiceFutureStub> loaderStubs = new
HashMap<>();

   @Autowired
   public DataLoadersCoordinator(DataLoaderStubFactory stubFactory,
                                 BucketsDistributor bucketsDistributor,
                                 LoaderUpdater updater) {
       this.stubFactory = stubFactory;
       this.bucketsDistributor = bucketsDistributor;
       this.loaderUpdater = updater;
   }

   @Override
   public void onLoadersInit(Set<Loader> loaders) {
       if (!activeLoaders.isEmpty()) {
           throw new IllegalStateException("active data-loaders map is not
empty!");
       }

       activeLoaders.addAll(loaders);

       Map<LoaderId, DataLoaderServiceFutureStub> newLoaderStubs =
               loaders.stream().collect(Collectors.toMap(
                       Loader::getId,
                       this::createDataLoaderStub));

       loaderStubs.putAll(newLoaderStubs);

       updateLoaders(bucketsDistributor.distributeOnLoadersInit(newLoaderStubs.k
eySet()));
   }

   @Override
   public void onLoaderAdded(Loader loader) {
       if (!activeLoaders.add(loader)) {
           throw new IllegalStateException("data-loader already existed: " +
loader.toString());
       }

       loaderStubs.put(loader.getId(), createDataLoaderStub(loader));

       updateLoaders(bucketsDistributor.distributeOnLoaderAdded(loader.getId(),
loaderStubs.keySet()));
   }

   @Override
   public void onLoaderRemoved(LoaderId loaderId) {
```

```
        if (!activeLoaders.removeIf(loader -> loader.getId().equals(loaderId))) {
            throw new IllegalStateException("data-loader does not exist: " +
loaderId.toString());
        }

        loaderStubs.remove(loaderId);

        updateLoaders(bucketsDistributor.distributeOnLoaderRemoved(loaderId,
loaderStubs.keySet()));
    }

    public Optional<Loader> getLoaderById(LoaderId id) {
        return activeLoaders.stream().filter(loader ->
loader.getId().equals(id)).findFirst();
    }

    private void updateLoaders(Map<LoaderId, BucketsDistribution>
distributionMap) {
        distributionMap.forEach((id, distribution) -> {
            logger.info("Updating loader {} with assigned workload", id.getId());

            loaderUpdater.updateLoader(loaderStubs.get(id), distribution);
        });
    }

    private DataLoaderServiceFutureStub createDataLoaderStub(Loader loader) {
        return
stubFactory.newFutureStub(loader.getEndpoints().getEndpoints().get(DataLoaderSer
viceGrpc.SERVICE_NAME));
    }
```

Приложение 3

```java
@Component
public class ActiveLoadersWatcher {

   private static final Logger logger =
LoggerFactory.getLogger(ActiveLoadersWatcher.class);

   private static final String LOADERS_PATH = "/announcements/data-loader";

   private final CuratorFramework zkClient;
   private final PathChildrenCache loadersCache;

   private final Set<LoadersChangeAware> subscribers;

   private final ObjectMapper objectMapper;

   private boolean isInitialized = false;

   public ActiveLoadersWatcher(@Value("${zookeeper.url}") String zkUrl,
                               Set<LoadersChangeAware> subscribers,
                               ObjectMapper objectMapper) {
       this.zkClient = createZkClient(zkUrl, new RetryNTimes(5, 500));
       this.loadersCache = new PathChildrenCache(zkClient, LOADERS_PATH, true);

       this.subscribers = subscribers;

       this.objectMapper = objectMapper;

       setupCache();
   }

   @PostConstruct
   public void startWatching() throws Exception {
       loadersCache.start(PathChildrenCache.StartMode.POST_INITIALIZED_EVENT);
   }

   public void subscribe(LoadersChangeAware subscriber) {
       Preconditions.checkNotNull(subscriber);

       subscribers.add(subscriber);
   }

   public void unsubscribe(LoadersChangeAware subscriber) {
       Preconditions.checkNotNull(subscriber);

       subscribers.remove(subscriber);
   }

   @PreDestroy
   public void stopWatching() throws IOException {
       loadersCache.close();
       zkClient.close();
   }

   private void setupCache() {
       PathChildrenCacheListener listener = new PathChildrenCacheListener() {
           @Override
```

```
            public void childEvent(CuratorFramework client,
PathChildrenCacheEvent event) throws Exception {
                switch (event.getType()) {
                    case INITIALIZED: {
                        Set<Loader> loaders = event.getInitialData().stream()
                                                              .map(childData -> {
                                                                  try {
                                                                      return
loaderFromEvent(childData);
                                                                  } catch (Exception ex)
{
                                                                      logger.error("An
exception was thrown while deserializing Loader: " + ex.toString());
                                                                      return null;
                                                                  }
                                                              })
                                                              .filter(Objects::nonNull)
                                                              .collect(Collectors.toSet()
);
                        callSubscribersOnInit(loaders);

                        break;
                    }

                    case CHILD_ADDED: {
                        callSubscribersOnAdded(loaderFromEvent(event.getData()));
                        break;
                    }

                    case CHILD_REMOVED: {
                        callSubscribersRemoved(event.getData().getPath());
                        break;
                    }

                    case CHILD_UPDATED: {
                        logger.error("data-loader is updated: " +
event.getData().getPath());
                        throw new IllegalStateException("data-loader is updated:
" + event.getData().getPath());
                    }
                }
            }
        };

        loadersCache.getListenable().addListener(listener);
    }

    private void callSubscribersOnInit(Set<Loader> loaders) {
        if (!isInitialized) {
            logger.info("data-loaders are initialized: " + loaders);
            subscribers.forEach(subscriber -> subscriber.onLoadersInit(loaders));

            isInitialized = true;
        } else {
            throw new IllegalStateException("Loaders cache is already
initialized");
        }
    }

    private void callSubscribersOnAdded(Loader loader) {
        if (isInitialized) {
            logger.info("New data-loader is added: " + loader);
```

```
            subscribers.forEach(subscriber -> subscriber.onLoaderAdded(loader));
        }
    }

    private void callSubscribersRemoved(String path) {
        if (isInitialized) {
            logger.info("data-loader is removed: " + path);
            subscribers.forEach(subscriber -> subscriber.onLoaderRemoved(new
LoaderId(path)));
        } else {
            throw new IllegalStateException("callSubscribersRemoved cannot be
called before loaders cache is initialized");
        }


    }

    private CuratorFramework createZkClient(String zookeeperUrl, RetryPolicy
retryPolicy) {
        CuratorFramework client = CuratorFrameworkFactory.newClient(zookeeperUrl,
retryPolicy);
        client.start();
        return client;
    }

    private Loader loaderFromEvent(ChildData childData) throws Exception {
        String path = childData.getPath();
        Endpoints endpoints = objectMapper.readValue(childData.getData(),
Endpoints.class);

        return new Loader(path, endpoints);
    }

}
```

```
public class DeduplicationTransformer<K, V, E> implements
ValueTransformerWithKey<K, V, V> {

   private final long leftDurationMs;
   private final long rightDurationMs;
   private final String storeName;

   private final KeyValueMapper<K, V, E> idExtractor;

   private WindowStore<E, Long> eventIdStore;

   public DeduplicationTransformer(long maintainDurationPerEventInMs,
KeyValueMapper<K, V, E> idExtractor,
                                   String storeName) {
       Preconditions.checkArgument(maintainDurationPerEventInMs > 0, "maintain
duration per event must be >= 1");

       this.leftDurationMs = maintainDurationPerEventInMs / 2;
       this.rightDurationMs = maintainDurationPerEventInMs - leftDurationMs;
       this.storeName = storeName;

       this.idExtractor = idExtractor;
   }

   @Override
   @SuppressWarnings("unchecked")
   public void init(ProcessorContext context) {
       eventIdStore = (WindowStore<E, Long>) context.getStateStore(storeName);
   }

   @Override
   public V transform(K readOnlyKey, V value) {
       E eventId = idExtractor.apply(readOnlyKey, value);
       if (Objects.isNull(eventId)) {
           return value;
       } else {
           V output;
           if (isDuplicate(eventId)) {
               output = null;
               updateTimestampOfExistingEventToPreventExpiry(eventId,
System.currentTimeMillis());
           } else {
               output = value;
               rememberNewEvent(eventId, System.currentTimeMillis());
           }

           return output;
       }
   }

   private void updateTimestampOfExistingEventToPreventExpiry(E eventId, long
timestamp) {
       eventIdStore.put(eventId, timestamp, timestamp);
   }

   private void rememberNewEvent(E eventId, long timestamp) {
       eventIdStore.put(eventId, timestamp, timestamp);
   }
```

```java
    private boolean isDuplicate(E eventId) {
        long eventTime = System.currentTimeMillis();
        WindowStoreIterator<Long> timeIterator = eventIdStore.fetch(
                eventId,
                eventTime - leftDurationMs,
                eventTime + rightDurationMs
        );

        boolean isDuplicate = timeIterator.hasNext();
        timeIterator.close();
        return isDuplicate;
    }

    @Override
    public void close() {

    }
}
```