

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS

Разработка распределённого встроенного хранилища "ключ-значение" для платформы JVM

Обучающийся / Student Федоров Сергей Денисович

Факультет/институт/клластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group Р34113

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Руководитель ВКР/ Thesis supervisor Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и

компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Консультант не из ИТМО / Third-party consultant Щербаков Алексей Михайлович, ООО "Тинькофф Центр Разработки", Ведущий разработчик, не имеет, не имеет

Обучающийся/Student

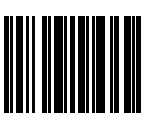
Документ подписан	
Федоров Сергей Денисович	
23.05.2022	

(эл. подпись/ signature)

Федоров Сергей
Денисович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
23.05.2022	

(эл. подпись/ signature)

Пенской
Александр
Владимирович

(Фамилия И.О./ name
and surname)

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Федоров Сергей Денисович
Факультет/институт/клластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group Р34113

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Разработка распределённого встроенного хранилища "ключ-значение" для платформы JVM

Руководитель ВКР/ Thesis supervisor Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Консультант не из ИТМО / Third-party consultant Щербаков Алексей Михайлович, ООО "Тинькофф Центр Разработки", Ведущий разработчик, не имеет, не имеет

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Цель работы: Разработка высокопроизводительного распределенного встроенного хранилища типа "ключ-значение".

Задачи:

- 1) изучить существующие реализации распределенных баз данных;
- 2) изучить существующие локальные системы хранения данных (дисковые);
- 3) разработать распределенную систему хранения данных на базе выбранного локального хранилища;
- 4) протестировать и сравнить разработанную систему с существующими аналогами.

Перечень подлежащих разработке вопросов:

- 1) программный интерфейс работы с распределённым хранилищем данных;
- 2) механизмы и программные интерфейсы взаимодействия локального и распределённого хранилища данных;
- 3) механизмы выбора лидирующего, алгоритма консенсуса и репликации журналов.

Форма представления материалов ВКР / Format(s) of thesis materials:

Форма представления основных или дополнительных результатов ВКР: текст.

Форма представления приложений: программный код, презентация.

Дата выдачи задания / Assignment issued on: 01.03.2022

Срок представления готовой ВКР / Deadline for final edition of the thesis 30.05.2022

Характеристика темы ВКР / Description of thesis subject (topic)

Название организации-партнера / Name of partner organization: -

Тема в области фундаментальных исследований / Subject of fundamental research: нет /
not

Тема в области прикладных исследований / Subject of applied research: да / yes

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
15.05.2022	

Пенской
Александр
Владимирович

(эл. подпись)

Документ подписан	
Федоров Сергей Денисович	
15.05.2022	

Федоров Сергей
Денисович

Задание принял к

исполнению/ Objectives

assumed BY

Руководитель ОП/ Head
of educational program

Документ подписан	
Дергачев Андрей Михайлович	
15.05.2022	

Дергачев
Андрей
Михайлович

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS

Обучающийся / Student Федоров Сергей Денисович

Факультет/институт/клластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Группа/Group Р34113

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Образовательная программа / Educational program Системное и прикладное программное обеспечение 2018

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Тема ВКР/ Thesis topic Разработка распределённого встроенного хранилища "ключ-значение" для платформы JVM

Руководитель ВКР/ Thesis supervisor Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и

компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Консультант не из ИТМО / Third-party consultant Щербаков Алексей Михайлович, ООО "Тинькофф Центр Разработки", Ведущий разработчик, не имеет, не имеет

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS

Цель исследования / Research goal

Повышение производительности распределенных систем с высокочастотной записью и чтением значений из базы данных, путем разработки распределенного встроенного хранилища.

Задачи, решаемые в ВКР / Research tasks

1) Изучение существующих реализаций распределенных баз данных; 2) Изучение существующих локальных систем хранения данных на диске; 3) Разработка распределенной системы хранения данных на базе выбранного локального хранилища; 4) Тестирование и сравнение разработанной системы с существующими аналогами.

Краткая характеристика полученных результатов / Short summary of results/findings

Было разработано распределённое встроенное хранилище данных. Реализованы: механизм взаимодействия локального и распределённого хранилища данных, механизм выбора лидирующего, алгоритм консенсуса и репликации журналов. Опытным путём получены результаты сравнения производительности разработанного решения и аналогов.

Полученные результаты подтвердили рост производительности для выбранных сценариев использования.

Обучающийся/Student

Документ подписан	
Федоров Сергей Денисович	
23.05.2022	

(эл. подпись/ signature)

Федоров Сергей
Денисович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
23.05.2022	

(эл. подпись/ signature)

Пенской
Александр
Владимирович

(Фамилия И.О./ name
and surname)

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ	7
ВВЕДЕНИЕ	9
СОДЕРЖАНИЕ	11
Актуальность разработки хранилища	11
Эволюция баз данных	11
Существующие NoSQL хранилища	15
Ограничения существующих систем	21
Анализ возможной реализации	26
Подсистема хранения данных	26
Алгоритм консенсуса	30
Протокол передачи данных	37
Используемые технологии	41
JVM – платформа исполнения	41
Scala – язык программирования	42
Cats-effect – контроль эффектов	44
Fs2 – обработка потоков данных	45
Protobuf – описание протокола	46
Netty – клиент-серверный фреймворк	47
RocksDB JNI API – интерфейс подсистемы хранения	47
Разработка хранилища	48
Разработка API для RocksDB	48
Реализация алгоритма RAFT	50
Модификация алгоритма консенсуса	55
Разработка Scala API	59
Тестирование производительности	61
ЗАКЛЮЧЕНИЕ	72
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ	73
ПРИЛОЖЕНИЕ А (Модуль работы с RocksDB)	76
ПРИЛОЖЕНИЕ Б (Модуль алгоритма консенсуса RAFT)	80
ПРИЛОЖЕНИЕ В (Модификация Мульти-RAFT)	120
ПРИЛОЖЕНИЕ Г (Scala API для разработанного хранилища)	128

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

- ПО – программное обеспечение
- БД – база данных
- СУБД – система управления базами данных
- ОС – операционная система
- SQL (Structured Query Language) – декларативный язык программирования создания, управления, модификации и запроса данных в реляционных базах данных
- ACID (Atomicity Consistency Isolation Durability) – акроним описывающий требования к транзакционной системе: атомарность, согласованность, изоляция, прочность
- BASE (Basically Available, Soft-state, Eventually consistent) – акроним описывающий требования к транзакционной системе: базовая доступность, неустойчивое состояние, согласованность в конечном счёте, обычно противопоставляется гарантиям ACID
- NoSQL (Not only SQL) – класс систем не поддерживающих гарантии реляционных баз данных, для возможности решения проблем масштабируемости и доступности
- NewSQL – класс систем пытающихся предоставить масштабируемость NoSQL систем, при этом сохраняя гарантии ACID
- OLAP (OnLine Analytical Processing) – тип обработки данных, заключающийся в подготовке агрегированной информации, используя большое количество исходных данных
- OLAP-куб – многомерный массив данных, как правило, разреженный и долговременно хранимый, используемый в OLAP системах
- API (Application Programming Interface) – описание различных способов для взаимодействия одной программы с другой

- WAL (Write-Ahead Log) – журнал записей команд, записываемые до изменения состояния базы данных
- CAP (Consistency, Availability, Partition tolerance) – теорема утверждающая невозможность обладания распределенной системой данных трех характеристик: согласованность данных, доступность, устойчивость к распаду сети
- JVM (Java Virtual Machine) – виртуальная машина Java, является основной частью исполняющей системы Java
- JNI (Java Native Interface) – интерфейс взаимодействия программ, исполняемых в виртуальной машине Java, с нативными программами
- HTTP (HyperText Transfer Protocol) – сетевой протокол передачи гипертекста
- IP (Internet Protocol) – протокол сетевого уровня стека TCP/IP
- TCP (Transmission Control Protocol) – протокол транспортного уровня стека TCP/IP, гарантирующий целостность доставленных данных и извещение отправителя о результатах передачи данных
- UDP (User Datagram Protocol) – протокол транспортного уровня стека TCP/IP, без гарантий надежности, упорядочивания или целостности данных
- TCP-over-UDP – подход реализации гарантий TCP поверх протокола UDP
- RPC (Remote Procedure Call) – тип интерфейса, позволяющий программам вызывать функции или процедуры на удаленных узлах

ВВЕДЕНИЕ

В настоящее время база данных была и является одним из основных компонентов практически любой программной системы. Базы данных используются в качестве хранилищ информации о реальном мире, такой как информации о пользователях системы или расписании тех, или иных событий, а также метаданных – информации необходимой программным продуктам для их функционирования, такие как слепки текущего состояния системы, время последнего обращения пользователя и другие. Различные типы и форматы данных диктуют различные сценарии их использования, а соответственно и различные подходы к проектированию разных систем баз данных.

В разные моменты времени, понятие базы данных изменилось достаточно сильно, пройдя путь от использования файлов на постоянном диске как основного хранилища до ныне разрабатываемых адаптивных облачных баз данных, которые автоматически подстраиваются под нагрузку и требования клиента без участия человека. Переосмысление архитектуры современных программных продуктов и их эволюция от запускаемых процедур до отдельных сервисов, также изменило и требования к используемым базам данных – разработанный в середине 70х годов стандарт SQL и требования ACID ушли на второй план[1]. На их место пришло понятие NoSQL, позволяющее строить различные типы систем, предоставляющих лишь те гарантии, которые нужны разработчикам и подходящие под конкретный сценарий использования.

Множество NoSQL баз данных относится к типу “ключ-значение”. В таких базах могут храниться флаги, конфигурационные значения, временно хранящиеся данные и другая, не требующая сложной обработки, информация. Также такие базы данных зачастую предлагают увеличенную,

по сравнению с реляционными базами данных, производительность и доступность. Различный характер требований к таким базам данных породил большое количество готовых решений: Redis, Aerospike, RocksDB, etcd, TiKV и другие. Однако среди существующих решений не предоставлено того, которое обладает всеми нижеперечисленными характеристиками одновременно: возможность хранения данных на постоянных дисках; высокая производительность для обработки высокочастотных запросов как на запись, так и на чтение; горизонтальная масштабируемость и согласованность, хотя бы в конечном счете.

Целью данной работы является повышение производительности распределенных систем с высокочастотной записью и чтением значений из базы данных, путем разработки высокопроизводительного встроенного распределенного хранилища. Высокая производительность, при хранении данных на диске, будет достигаться путем использования готовой локальной системы хранения данных и уменьшением сетевого взаимодействия между узлами. Согласованность и горизонтальная масштабируемость будут достигаться за счет реализованного алгоритма консенсуса. Данная разработка потребует решения следующих задач:

1. Изучить существующие реализации распределенных баз данных
2. Изучить существующие локальные системы хранения данных (дисковые).
3. Разработать распределенную систему хранения данных на базе выбранного локального хранилища.
4. Протестировать и сравнить разработанную систему с существующими аналогами.

СОДЕРЖАНИЕ

1. Актуальность разработки хранилища

1.1. Эволюция баз данных

Рассмотрение характера изменения архитектур и подходов к проектированию и использованию баз данных позволяет понять с какими конкретно проблемами инженерные решения прошлого боролись и какие последствия на итоговую программную систему будут, если у решения убрать те или иные свойства и характеристики.

Изначальная форма представления баз данных являлась обыкновенным файловым хранилищем. Вычислительные системы использовались для расчетов, то есть целью был некий конечный результат, а не бесперебойное обслуживание поступающих запросов. Еще одной особенностью являлось то, что данный подход не сильно отличается от концепции совмещения программного кода и данных, а значит что, даже несмотря на то, что данные ушли в файлы, методы работы с данными изменились лишь незначительно. Однако появились новые методы доступа, ввиду ограничений энергонезависимой памяти: последовательный; индексный; случайный. Данные особенности работы с памятью приходится учитывать при разработке структуры хранения данных и алгоритмов записи/чтения.

Использование файловых баз данных было сопряжено со следующими проблемами: проблема одновременного многопользовательского доступа, отсутствие транзакций, отсутствие стандартного описания формата записи данных на диске. Эти проблемы будут частично решены разработкой Иерархических баз данных. Основным их нововведением было появления связи “один-ко-многим”, что в свою очередь произвело разделение схемы

базы данных и самих данных. Графический пример устройства такой БД представлен на Рисунке 1.

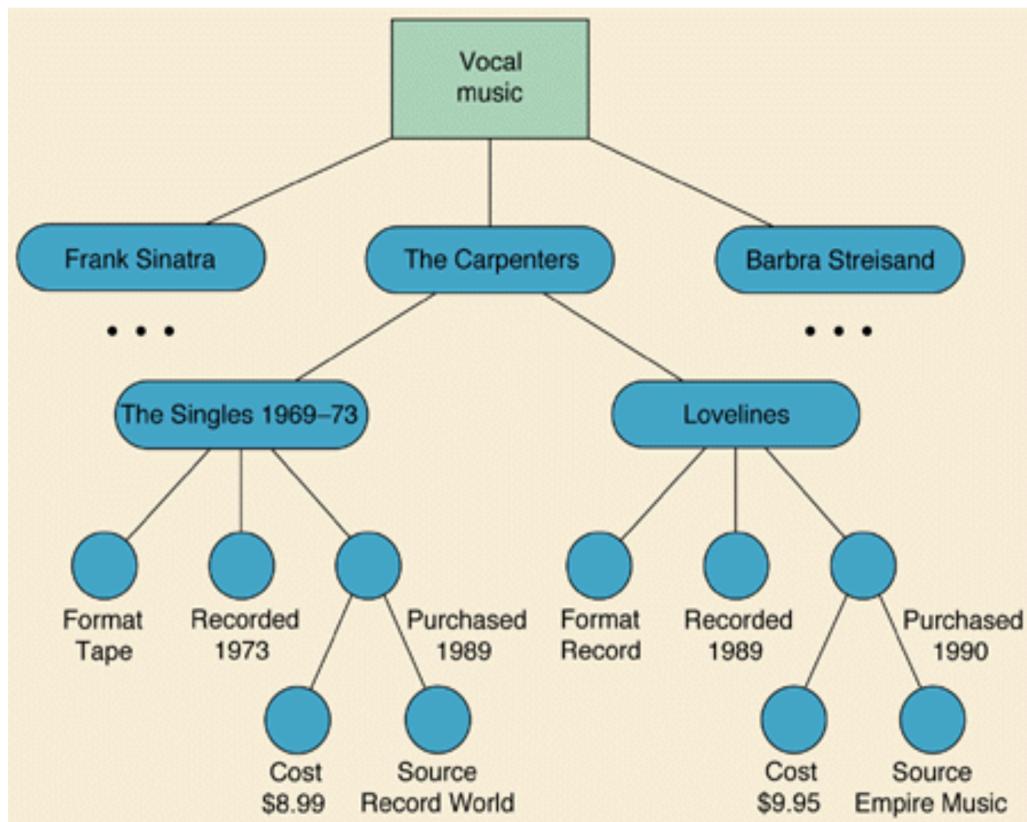


Рисунок 1 - Устройство иерархической базы данных[1]

Небольшой модификацией иерархических баз данных была разработка Сетевых баз данных, в формате которых данные связаны не деревом, а сетевой структурой – произвольным графом.

Дальнейшим развитием стало появление Реляционной модели данных, диктующей операции с логическими понятиями, а не с физическими структурами. Вводится понятие реляционной алгебры, а также взаимодействие с базами данных происходит путем использования декларативных запросов, которые описывают не способ получения данных, а их конкретный, ожидаемый формат. Появляются критерии надежности в виде 9 нормальных форм, который описывают свойства построения доменной

модели, а также появляются транзакции, изолирующие любые изменения вносимые пользователем в систему, до момента подтверждения данных изменений. Пример сущностей и их связей в реляционной БД представлен на Рисунке 2.



Рисунок 2 - Пример реляционной модели

За счет предоставления многих гарантий и взятия на себя задачи по обеспечению согласованности данных, реляционные базы данных имеют соответствующие издержки:

- Представляя все отношения в виде таблиц, база данных вводит ограничение на способы представления данных в системе. Не всегда предметная область может быть эффективно представлена в виде таблиц и их связей.
- Мало типов данных. Слабая поддержка расширяемости типов.
- В результате логического разделения таблиц при проектировании, позднее увеличивается сложность в понимании доменной модели.
- Внутреннее устройство хранения данных на дисках и декларативная природа запросов приводят к низкой скорости доступа к данным.

- Из-за обеспечения различных предоставляемых гарантий, такие системы не имеют возможности легко горизонтально масштабироваться, тем самым имея гораздо большее ограничение на максимальную производительность и объем хранимых данных.

Часть данных недостатков призваны решить разработанные Объектно-Реляционные базы данных, однако далеко не все вышеперечисленные проблемы могут быть решены таким подходом.

Помимо ранее разобранных проблем реляционных баз данных имеются и другие предпосылки к появлению и популяризации NoSQL хранилищ. Во-первых, появление и адаптация гибких методологий с итеративными циклами разработки требуют, чтобы каждый выпуск новой версии программного обеспечения не требовал сложной и долгой подготовки базы данных, для обеспечения скорости предоставления нового продукта конечному пользователю и уменьшения временных затрат разработчиков. И если механизм миграции схемы базы данных позволяет уменьшить сложность задачи изменения модели данных, то это никак не решает проблему больших затрат по времени на обновление базы. Во-вторых, все больше приходится обрабатывать запросов на получение информации из базы данных, без дополнительной агрегации или обработки, соответственно язык SQL и его обработчик становятся неэффективными. В-третьих, происходит смещение понятия программной системы, и все чаще используется сервисная архитектура, которая в свою очередь сильно меняет подход к использованию баз данных:

- Сервисы более не нуждаются в монолитной структуре. Каждый отдельный сервис имеет возможность использовать отдельную базу данных. При таком подходе часто достаточно баз данных с простой структурой, таких как “ключ-значение”

- Сервисы сами реализуют агрегацию данных, в том виде в котором будет оптимально для каждой конкретной задачи. Например, реализация OLAP-куба на базе колоночной базы данных.
- Сервисы сами реализуют контроль целостности данных, для тонкой настройки производительности и обеспечения сложной логики контроля целостности. Зачастую это означает что не все предоставляемые реляционной базой гарантии ACID нужны и отдается предпочтение базам типа BASE.
- Необходимость в обработке большого количества одновременных запросов, которые могут производить одновременные изменения в базе данных.
- По мере роста количества сервисов и количества обрабатываемых запросов необходимость в возможности горизонтального масштабирования становится критически важной для базы данных. База данных, должна быть распределенной.

Вышеизложенные пункты стали причинами для разработки нового типа баз данных – NoSQL базы данных, основной характеристикой которых было отсутствие поддержки стандарта языка SQL и их заточенность под конкретный сценарий использования. Данный подход будет дальше использоваться как ключевой, при построении собственного хранилища.

1.2. Существующие NoSQL хранилища

Количество различных типов и подходов при проектировании NoSQL хранилищ огромно, следовательно, будут рассматриваться только те решения, которые имеют схожую к разрабатываемой системе архитектуру и подход к

хранению данных, а именно будут рассмотрены несколько баз данных типа “ключ-значение”.

Самой популярной базой данных типа “ключ-значение” на данный момент является Redis[2]. Данная база данных имеет ряд преимуществ перед своими конкурентами:

- Redis обладает отличной производительностью, как на запись, так и на чтение. Достигается это оптимизированным программным обеспечением, простотой внутренних структур данных и, самое главное, работой в оперативной памяти – данные не записываются в постоянную память.
- Redis имеет простой и универсальный протокол общения RESP, описывающий взаимодействия со структурами данных, представленными в Redis. Многие другие базы данных типа “ключ-значение” обычно реализуют собственный API и прокси интерфейс, который обрабатывает команды Redis протокола.
- Redis, несмотря на то, что имеет единственный режим работы только в оперативной памяти, поддерживает запись WAL журнала на диски постоянной памяти, тем самым позволяя сохранять состояние, и при необходимости производить восстановление системы в случае сбоя.
- Redis поддерживает репликацию WAL журнала команд в формате “Ведущий-Ведомый”. Пример такой репликации схематично продемонстрирован на Рисунке 3.

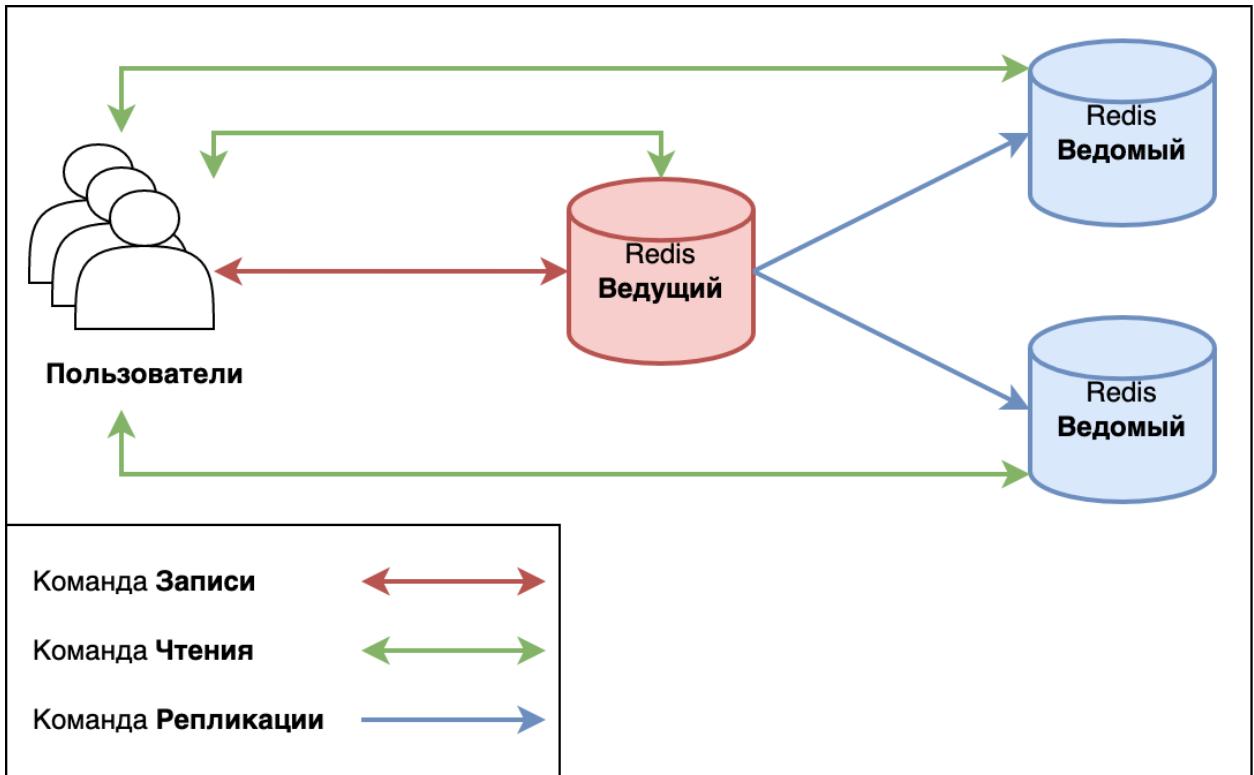


Рисунок 3 - Конфигурация REDIS с моделью “Ведущий – Ведомый”

Одной из самых популярных баз данных типа “ключ-значение”, гарантирующей горизонтальную масштабируемость, устойчивость к распаду сети и строгую согласованность является etcd[3]. Данная база данных, чаще всего, используется для хранения различных конфигурационных значений или в качестве внешней системы консенсуса для некоторого значения. Из особенностей можно выделить:

- Удобный интерфейс работы с базой. Запросы на запись и чтение могут быть выполнены обычным HTTP запросом.
- Иерархическая структура хранения ключей, схожая с обычной файловой системой.
- Механизм подписки на обновление конкретного значения

- Является строго согласованной и устойчивой к распаду сети базой данных, в терминах CAP теоремы, является CP системой. Данные свойства достигаются путем реализации алгоритма консенсуса RAFT[4]. Схематическое представление устройства системы изображено на Рисунке 4.
- Реализация алгоритма консенсуса RAFT в etcd достаточно часто считается эталонной и многими сторонними системами переиспользуется.

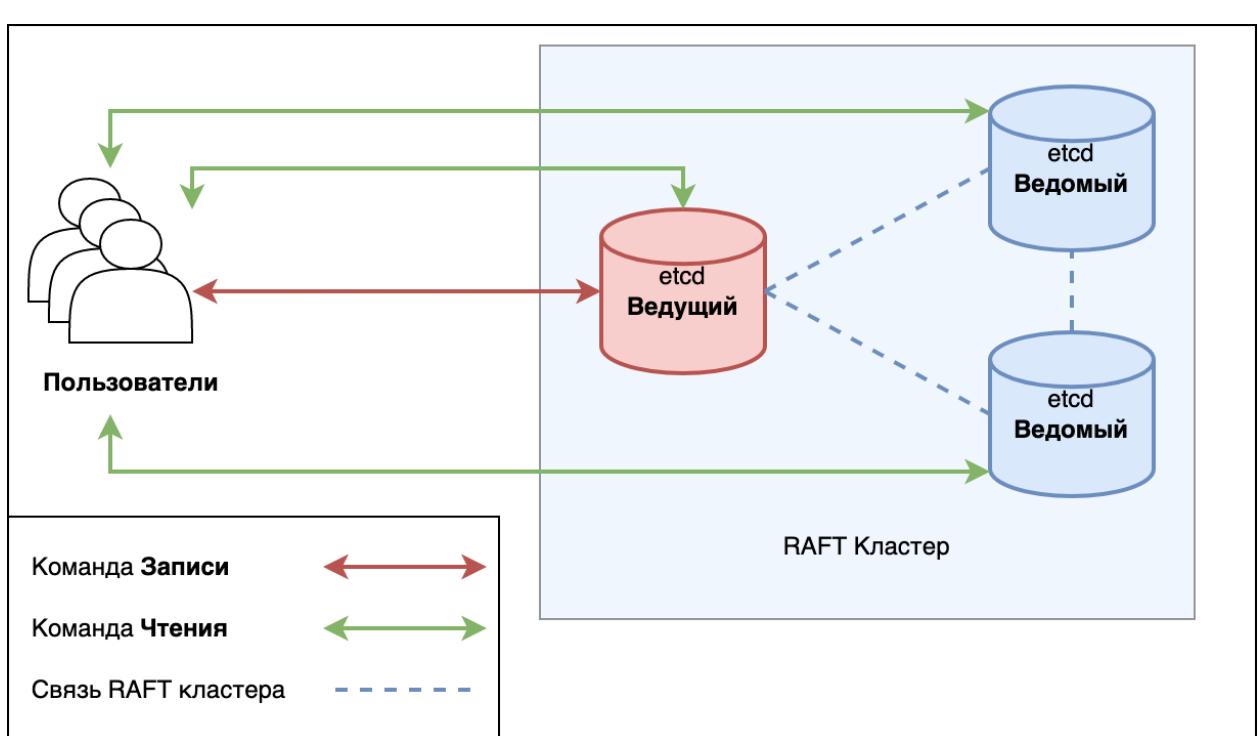


Рисунок 4 - Внутреннее устройство кластера etcd

Популярным решением высокопроизводительного распределенное хранилища типа “ключ-значение” является система Aerospike[5]. Это платное программное обеспечение, предоставляющее строго согласованный кластер, с оптимизированной работой с твердотельными накопителями и индексами в оперативной памяти. Ключевые особенности системы:

- Система поддерживает два режима работы. Режим работы только с оперативной памятью и режим работы с записью данных на диск. Первый режим работы рассчитан на работу с системой в качестве буфера или хранения временной информации. Второй режим используется при необходимости сохранения информации в постоянное хранилище.
- Aerospike поддерживает ACID транзакции в работе с единственным ключом. Например: прочитать значение; удвоить значение; записать обновленное значение обратно.
- Aerospike может быть сконфигурирован и как AP-система – система с высокой доступностью и устойчивостью к распаду сети, так и как CP-система – система со строгой согласованностью и устойчивостью к распаду сети. Данное свойство системы, делает Aerospike практически универсальным хранилищем для большинства задач.

Отличным от вышеописанных хранилищ “ключ-значение” является тип подсистем хранения, которые обычно используются для разработки других баз данных и отвечают в основном за выполнение низкоуровневых задач работы с диском, таких как сохранение новых значений в файлы базы данных или сжатие уже существующих значений с удалением ненужной информации. Существование данных систем обусловлено сложностью работы с данными хранящимися на дисках, ввиду их увеличенной задержки, по сравнению с оперативной памятью, и разницы в производительности в зависимости от режима доступа к данным. Одной из самых популярных и широко используемой подсистемой хранения данных является RocksDB[6], из ее особенностей можно выделить следующее:

- RocksDB базируется на другой подсистеме хранения – LevelDB и является ее улучшенным вариантом. Главные отличия это оптимизированная работа с твердотельными накопителями и дополненная функциональность.
- RocksDB имеет механизм транзакций.
- RocksDB имеет механизм создания слепков базы данных, которые могут быть использованы для восстановления или передачи текущего состояния базы данных.
- Передача данных и вызов конкретных процедур в RocksDB происходит не через сетевой обмен сообщениями по такому стеку как TCP/IP, а через нативные вызовы функций, используя либо скомпилированные под конкретную систему бинарные файлы, либо через JNI для платформы JVM.
- Данные в RocksDB не имеют типов, поэтому все вхождения в базу данных хранятся в бинарном виде.
- Также RocksDB обладает следующими свойствами и характеристиками: является системой типа “семейство столбцов”; поддерживает запись временных ключей; имеет реализацию bloom фильтров[7] по ключам, имеется атомарный оператор слияния значений.
- RocksDB предоставляет возможность управлять низкоуровневыми процессами, для оптимизации производительности и поддержания согласованности данных. Например, можно приостановить все фоновые процессы базы данных, для выполнения некоторой логики, затрагивающей базу данных как таковую, а не хранимые данные.

Схематичное устройство взаимодействия с RocksDB предоставлено на Рисунке 5.

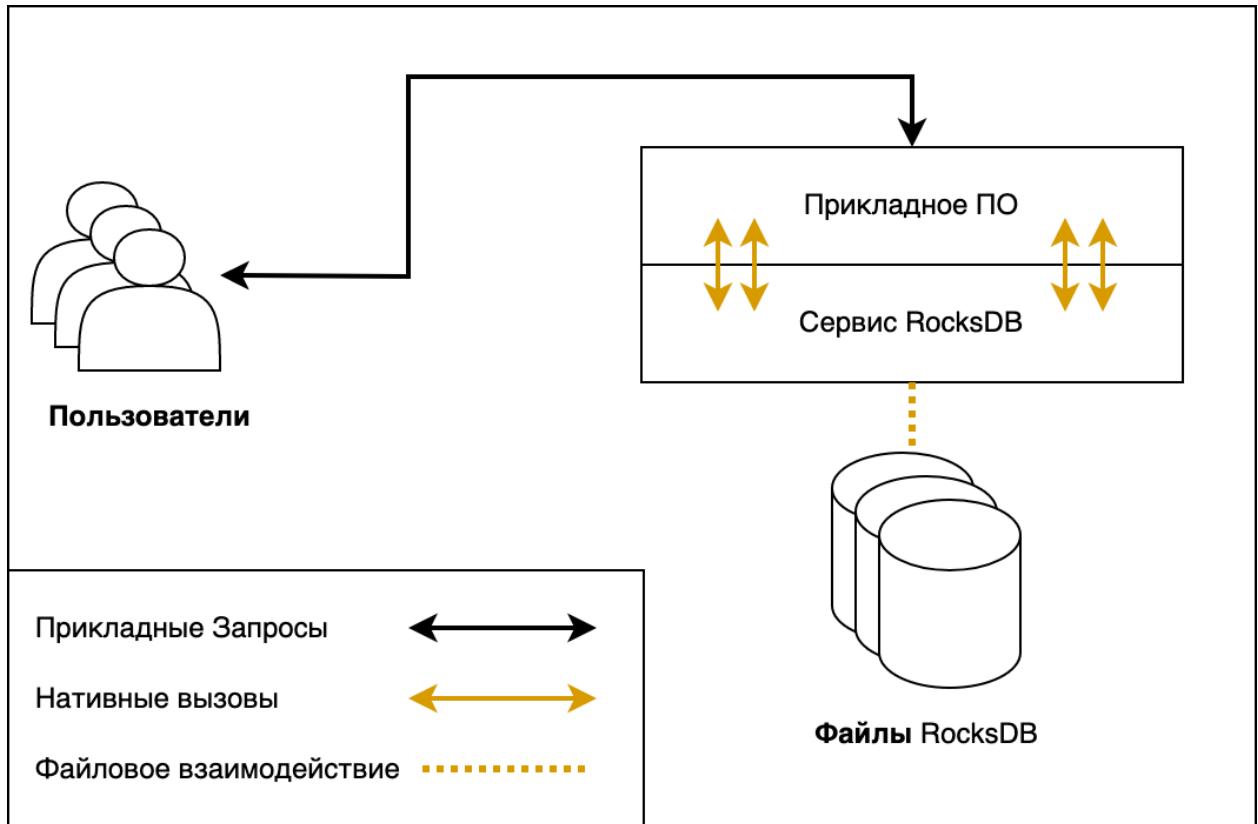


Рисунок 5 - Устройство взаимодействия RocksDB

1.3. Ограничения существующих систем

Все из вышеперечисленных реализаций NoSQL баз данных являются активно поддерживаемыми и развивающимися проектами, а также протестированы на практике различными командами и в различных условиях. Однако ввиду либо исторических, либо архитектурных причин у каждой из данных систем есть свои недостатки, некоторые из которых будут приведены далее.

Несмотря на то, что Redis на данный момент практически является стандартом хранилища типа “ключ-значение” у него есть ряд серьезных

недостатков, из-за которых все больше проектов чаще используют другие базы данных:

- Несмотря на то, что Redis способен обслуживать команды различных пользователей одновременно, обработка самих команд ведется в однопоточном режиме, что снижает эффективность работы Redis на многоядерных системах, так как база данных не утилизирует все ресурсы центрального процессора. Такой подход применяется для добавления командам свойств атомарности без затрат на дополнительную синхронизацию.
- Redis имеет возможность репликации данных в топологии “Ведущий-Ведомый”, однако такой подход имеет следующие недостатки: вся нагрузка по обработке операций записи ложится на один узел системы, с дополнительной нагрузкой в виде пересылки данных ведомым; в случае аварийной остановки ведущего узла, система не может поддерживать свою работоспособность.
- Redis имеет возможность репликации данных в режиме кластера, однако такой подход имеет следующие ограничения: конфигурация кластера не расширяема во время его работы, каждый узел должен, при подключении к кластеру, иметь представление о топологии всего кластера; функциональность кластерного режима Redis является ограниченной по сравнению с обычным режимом или режимом репликации в режиме “Ведомый-Ведущий”.

Хранилище etcd является отличным выбором при необходимости хранить большое количество конфигурационной, мета или другой иной информации с гарантией надежности, целостности и устойчивости, однако следуя постулату теоремы CAP, главный недостаток данной системы –

невозможность гарантировать постоянную доступность и производительность сопоставимую с АР системами.

Система Aerospike является универсальным решением для большинства задач ввиду ее гибкости конфигурации и различных режимов работы, однако при этом присутствует ряд недостатков, которое стоит учесть:

- Система Aerospike записывает все добавления и обновления значений в виде неизменяемых блоков, которые позднее будут сжаты на этапе compaction (этапе компоновки изменений в системе). Ввиду структуры устройства сохраняемого значения – список “ключ-значение”, которые в официальной документации называются bin, и необходимости эффективно проводить этап compaction, Aerospike вынужден новые операции добавления/обновления сохранять в памяти в виде блоков размером от 128 КБ до 1 МБ. Проблема заключается в том, что при частой записи значений в систему и ограниченным объемом оперативной памяти может потребоваться больше памяти чем система может предоставить процессу, до того произойдет вызов этапа compaction. Данная проблема в Aerospike не решается в автономном режиме и все сводится к правильной настройке параметров системы, которая состоит в нахождении баланса между размером записываемых блоков, частотой вызова процедуры compaction и снижением максимального размера ключа.
- Имея много различных улучшений и реализаций оптимизированных алгоритмов, Aerospike имеет различные версии поставки продукта: бесплатная и платная. Большое количество функциональности системы содержится только в платной версии поставки продукта, среди прочего: режим строгой согласованности и географической распределенности; режим работы только на дисках постоянной памяти и дисках Intel

Optane[8]; уведомления об изменении значений ключей; механизм сжатия данных на дисках; готовые конфигурации подключения Aerospike к различным системам обработки данных. Стоит также заметить, что стоимость платной версии настолько велика, что многие коммерческие команды разработки не могут себе позволить ее приобретение, а соответственно данное решение доступно только большим и хорошо финансируемым компаниям.

Система хранения данных RocksDB изначально создавалась в качестве подсистемы хранения данных, соответственно не должна использоваться как отдельная, готовая к применению на прикладном уровне база данных. Данная система предоставляет отличный набор инструментов для оперирования данными для низкоуровневых оптимизаций и проверок, однако реализация остальных механизмов по сохранению целостности, репликации, сериализации данных и другие – делегируются системам которые будут использовать RocksDB.

Все вышеописанные системы имеют свои преимущества и недостатки, однако существуют сценарии использования баз данных, при которых ни одна из вышеописанных систем не будет подходить под выполнения задачи без модификации самих систем.

Сценарий использования, под который нужна новая реализация базы данных или модификация существующей, включает в себя следующие пункты:

- Высокая нагрузка в виде операций записи и чтения значений по ключу.
- Репликация журналов операций и состояния между узлами сети.

- Возможность расширения и уменьшения кластера в динамическом режиме – без необходимости в приостановке работы.
- Низкая задержка в ответе, как на операцию чтения, так и на операцию записи.
- Линеаризуемость операций в рамках одного узла и наличие согласованности всего кластера в конечном счете.
- Опциональная возможность посылки команды записи/чтения, требующей строгой согласованности.

Далее приведем практический пример, когда необходимо наличие всех описанных свойств одновременно. Современные OLAP системы строятся поверх систем баз данных, поддерживающих запись огромного количества данных, с последующим не частым агрегированием значений, используя сложные и долго выполняемые запросы. Для некоторых практических применений полезно иметь возможность отдавать составные результаты информации о событиях с сопоставимой скоростью по сравнению с записью. Примером может служить механизм онлайн голосования, где участники опроса могут моментально наблюдать за изменением результатов опроса в режиме постоянных обновления значений. Такие задачи могут быть решены, за счет изменения подхода к агрегации информации – когда известно конечное применение системы, а размеры составных результатов увеличиваются меньше чем линейно от количества событий или, в лучшем случае, остаются неизменными, в таком случае можно использовать алгоритм агрегации в момент поступления события, а затем, при необходимости, моментально получать предагрегированное значение искомой метрики. Такая система как раз подпадает под сценарий использования описанный выше.

2. Анализ возможной реализации

2.1. Подсистема хранения данных

По причинам оговоренным ранее, разработка хранилища, сохраняющего свое состояние на диск постоянной памяти, является сложной инженерной задачей, включающей в себя следующие подзадачи для достижения оптимальной производительности:

- Программирование на языке, чьи абстракции будут близки к аппаратным, для достижения увеличенных характеристик производительности.
- Реализация структуры хранения данных с нуля. Структура должна учитывать природу различных типов дисков постоянной памяти, возможность конкурентного использования, обновления данных, удаления данных, вставки новых данных и наличия циклов обслуживания структуры при необходимости, таких как, например балансировка В-дерева.
- При реализации различной функциональности необходимо учитывать особенности реализации целевой операционной системы, среди которых: особенности ее системных вызовов и флагов; особенности работы системы с блочными устройствами; особенности реализации виртуальной памяти, такие как синхронизации страниц с постоянной памятью.
- Настройка и оптимизация производительности системы, путем тщательного подбора подходящих алгоритмов и параметров.

Данный внушительный список задач, чье решение необходимо для написания собственного хранилища с нуля, вынуждает разработчиков

большинства баз данных использовать готовые решения, так называемые подсистемы хранения данных. Данные системы обычно не предоставляют таких же гарантий и возможностей как другие типы баз данных: часто не будет никакого механизма сериализации и работать необходимо с массивами байт; отсутствуют решения которые предоставляли бы все гарантии ACID; чаще всего такие системы являются встроеннымми, то есть не запускаются как отдельная система и не обмениваются сообщениями по сетевому протоколу. Учитывая что существуют различные типы подсистем хранения данных, стоит заметить что самый часто используемый тип – хранилище типа “ключ-значение”.

Можно выделить несколько популярных и производительных подсистем хранения данных:

- LMDB – open-source проект, в основе лежит структура B+ дерева
- LevelDB – open-source проект, в основном разрабатываемый компанией Google, с их наработками от системы BigTable. Система использует структуру LSM-дерева.
- HyperLevelDB – проект на базе LevelDB. Основными направлениями оптимизаций было: распределение различных задач в фоновые процессы; отмена механизма обратного давления при записи на диск; оптимизация сжатия файлов.
- RocksDB – проект на базе LevelDB. Основными направлениями оптимизаций было: добавление функциональности, необходимой для облегченного пользования системой, таких как транзакции на ключе, итераторы по значениям, семейства столбцов и другие; оптимизированная работа с flash-накопителями.

Ввиду того что функциональность вышеописанных подсистем сопоставима друг с другом, определяющим фактором выбора конкретной системы будет производительность каждого из решений, под различной нагрузкой.

Согласно отчету о производительности, опубликованного компанией InfluxData[9], проводилось тестирование всех вышеописанных систем при следующем сценарии нагрузки:

- 1) Были записаны 100 миллионов значений с размером ключа равным 24 байт. Запись значений производится в пакетном режиме, по 1000 значений за раз.
- 2) Запрос 100 миллионов отсортированных значений, с проверкой на остановку при определенном значении.
- 3) Удаление 50 миллионов значений.
- 4) Форсирование процесса компоновки и ожидание его завершения.
- 5) Запрос 50 миллионов отсортированных значений.
- 6) Запись 50 миллионов значений.

Результаты теста представлены в Таблице 1.

Таблица 1 - Тестирование подсистем хранения

Шаг теста/ Контрольное значение	LevelDB	RocksDB	HyperLevelDB	LMDB
Запись 100 миллионов значений	36 мин 8 сек	21 мин 18 сек	10 мин 45 сек	1 час 13 мин 21 сек
Размер хранилища	2.7 ГБ	3.2 ГБ	3.2 ГБ	7.6 ГБ
Запрос 100 миллионов значений	2 мин 55 сек	2 мин 44 сек	13 мин 49 сек	5 мин 24 сек
Удаление 50 миллионов значений	3 мин 47 сек	1 мин 53 сек	6 мин 1 сек	6 мин 15 сек
Этап компоновки	3 мин 59 сек	3 мин 20 сек	6 мин 33 сек	0 сек
Размер хранилища	1.4 ГБ	1.6 ГБ	1.6 ГБ	7.6 ГБ
Запрос 50 миллионов значений	12 сек	13 сек	23 сек	8 сек
Запись 50 миллионов значений	3 мин 5 сек	1 мин 26 сек	1 мин 54 сек	3 мин 25 сек
Размер хранилища после компрессии	673 МБ	993 МБ	928 МБ	2.5 ГБ

Из результатов тестирования становится понятно что различные подсистемы хранения оптимизированы под различные сценарии нагрузки, однако так как предполагаемый сценарий нагрузки на подсистему хранения данных является сбалансированным – примерно одинаковое соотношение на запись и на чтение, лучшим решением для выполнения данной работы будет RocksDB.

2.2. Алгоритм консенсуса

Алгоритм консенсуса – алгоритм получения согласованного состояния в распределенной сети. Состояние может согласовываться путем обмена командами на изменение состояния локального узла и согласование поступления и применения данной команды между участниками распределенной сети.

Обоснование использования алгоритма консенсуса вместо модели передачи состояния типа “Ведущий-Ведомый”:

- Необходимо ручное управление “Ведущим” и в случае его отказа система становится неработоспособной.
- Неравномерно разделенная нагрузка на различные узлы сети
- При наличии разделения в сети, система также становится неработоспособной.

По итогу, использование алгоритма передачи состояния типа “Ведущий-Ведомый” стоит использовать при абсолютной необходимости в повышенной производительности, за счет уменьшения устойчивости системы к ошибкам.

На данный момент времени, существует два хорошо описанных, стандартизованных алгоритма консенсуса[10]: алгоритм консенсуса Паксос[11] и алгоритм консенсуса RAFT[4]. На базе данных алгоритмов были предложены несколько улучшений и типов их реализации для достижения улучшенной производительности. Список таких алгоритмов:

- Алгоритмы семейства Паксос:
 - Стандартный Паксос
 - Мульти-Паксос[12]
 - Быстрый Паксос[13]
 - Эгалитарный Паксос[14]
 - Гибкий Паксос[15]
 - Обобщенный Паксос[16]
- Алгоритмы семейства RAFT
 - Стандартный RAFT
 - Мульти-RAFT[17]

Первым разработанным алгоритмом из списка выше был Алгоритм Паксос в его стандартном виде. В этой версии алгоритма у узла распределенной системы есть три возможные роли во время работы алгоритма:

- Заявитель – узел предлагающий изменение в состояние системы
- Акцептор – узлы голосующие за изменение
- Ученик – узлы хранящие результаты принятых решений

Каждый раз, при наличии запроса на изменение состояния, должно происходить два этапа: этап голосования на выдачу прав заявителю; этап принятия решения по запросу на изменение.

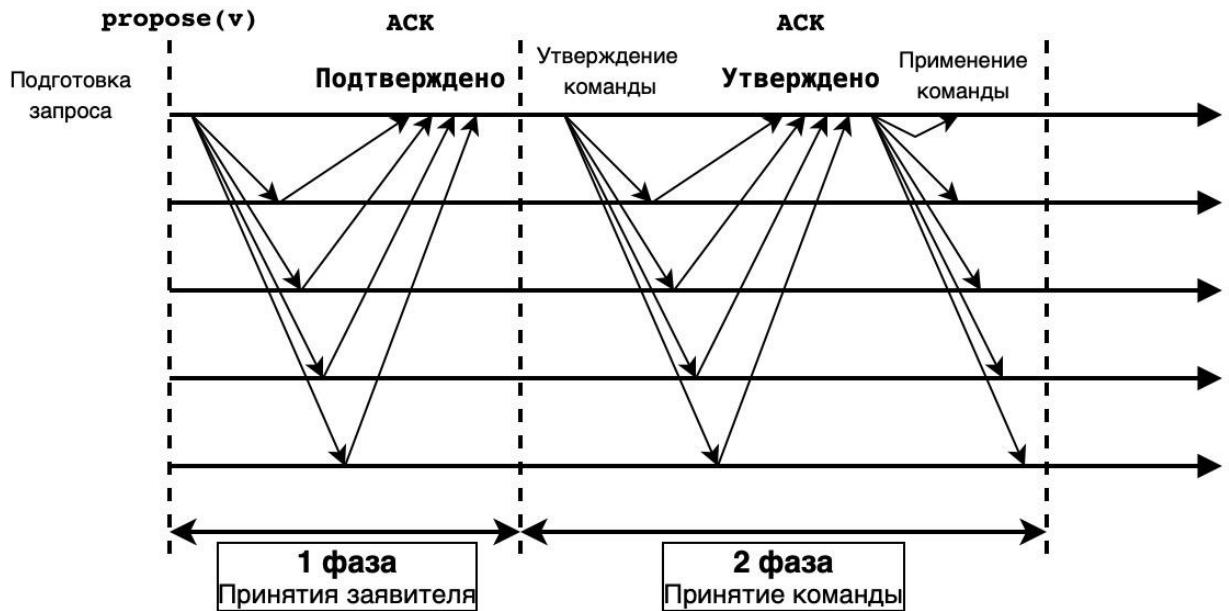


Рисунок 6 - Стандартные этапы алгоритма Паксос

Основными недостатками этого алгоритма являются: постоянная необходимость в выборе подтвержденного заявителя; наличие двух фаз принятия каждой команды; сложность алгоритма. Дальнейшие модификации данного алгоритма адресовали эти и другие проблемы.

Мульти-Паксос вводит понятие лидера и предлагает не переизбирать заявителя каждый раз. Такое решение помогает уменьшить количество этапов для принятия каждой команды до одного, а также позволяет использовать механизм выбора лидирующего.

Быстрый Паксос модифицирует раунд принятия решения по команде изменения состояния. Вводится понятия быстрого и классического раунда. Классический раунд – порядок событий, как и у обычного раунда Паксос. Быстрый раунд позволяет заявителям отправлять команды непосредственно акцепторам в обход лидера, а лидер лишь занимается подтверждением этих изменений.

Эгалитарный Паксос. Вместо использования лидера и номеров команд для их линеаризации, алгоритм предлагает использовать лидера для фиксации каждой конкретной команды и устанавливать порядок команд путем просмотра и установки зависимостей.

Гибкий Паксос позволяет на втором этапе алгоритма, а именно на этапе подтверждения значения набирать не большинство голосов, а лишь непустое множество удовлетворяющее установленному кворуму. Например, имея 5 акцепторов в системе, лидер может себе позволить дожидаться положительного ответа лишь от 2-х акцепторов, а не от 4-х, как того требовал бы стандартный алгоритм консенсуса Паксос. Формула для расчета конкретного количества необходимых подтверждений на каждом из этапов для установления кворума в данном алгоритме представлена ниже, где Q_1 – количество подтверждений на этапе выбора заявителя, Q_2 – количество подтверждений этапе подтверждения команды и N – общее количество акцепторов в системе:

$$Q_1 + Q_2 > N$$

Обобщенный алгоритм консенсуса Паксос является расширенной версией Эгалитарной версией алгоритма, а точнее является его улучшенной реализацией. Вводится понятие регистров, клиентов и серверов, впоследствии регистры используются для разрешения конфликтов на чтение и запись между клиентами и серверами и при отсутствии конфликтов позволяют проводить несколько операций параллельно.

Основными недостатками алгоритма Паксос и его улучшенных модификаций являются сложность в понимании и изучении, а также отсутствие строго задокументированной реализации алгоритма. Чтобы

адресовать эти проблемы, был разработан алгоритм консенсуса RAFT, основной задачей которого было создание консенсуса в распределенной сети с сопоставимой производительностью алгоритму Паксос, но упрощенной структурой, а соответственно и меньшим порогом вхождения во время изучения данного алгоритма.

Алгоритм RAFT делится на два этапа: этап выбора лидирующего и этап репликации данных. Этап выбора лидирующего состоит в том, чтобы по истечению некоторого, заранее установленного, срока узел при отсутствии сообщений от лидирующего за этот срок становится кандидатом в лидирующие узлы, после чего инициирует начало выборов лидирующего и если набирает большинство голосов, становится лидером, на этом этапе выбора лидирующего завершается. Задачей лидирующего узла, во время этапа репликации данных, распространять информацию об изменениях в системе, получать информацию о состоянии системы и отдавать пользователю и периодически поддерживать связь с другими участниками системы чтобы те не становились кандидатами. Если в какой-то момент времени, лидирующий узел становится недоступен для остальных участников сети или сообщения о проверке связи не дошли до адресатов, то этап выбора лидирующего инициируется снова. Краткая графическая презентация алгоритма представлена на Рисунке 7.

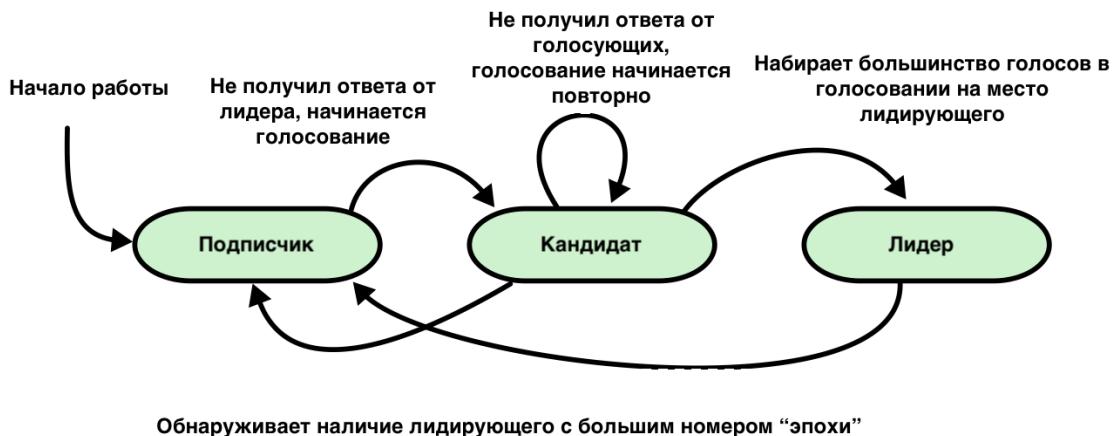


Рисунок 7 - Жизненный цикл узла в алгоритме RAFT

Алгоритм RAFT при всех его достоинствах имеет также и недостатки, которые мешают его использованию в чистом виде. Главным недостатком является наличие единственного лидирующего узла, через который должны проходить как все команды на изменение, так и на получение состояния, при необходимости в линеаризации данных. Подходом призванным решить данную проблему является Мульти-RAFT, который разделяет состояние сети на области ответственности между всеми участниками сети и на каждую из таких областей создается отдельный RAFT кластер, или по другому, RAFT группа, со своим лидером и подписчиками. В случае с хранилищем типа “ключ-значение” можно разделить пространство ключей на области используя различные правила, например по диапазону значений или по значению хеш-ключа. Пример такого разделения продемонстрирован на Рисунке 8. Однозначно правильной или лучшей реализации данного подхода не существует и каждый раз используется такая модификация алгоритма, которая подойдет в наибольшей степени к конкретной задаче. Примерами систем использующие Мульти-RAFT являются: TiKV, CockroachDB и другие.

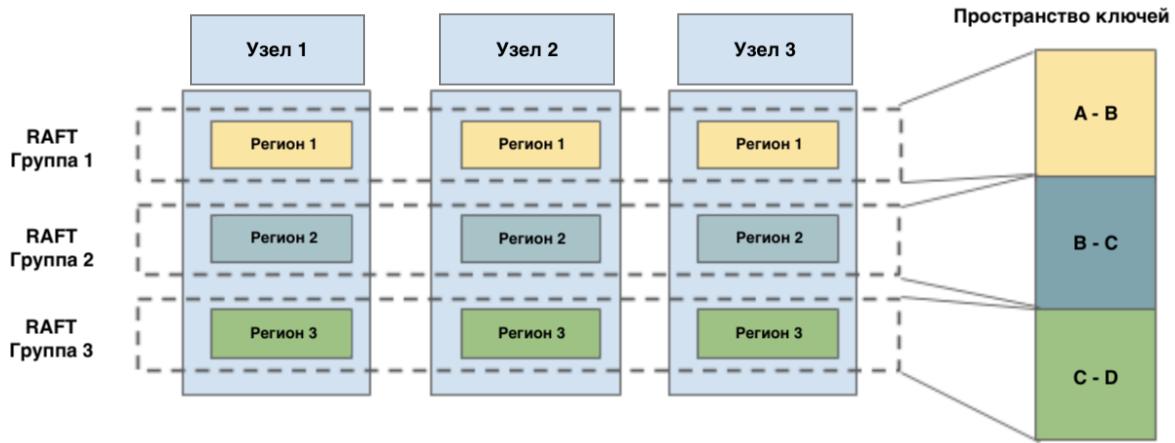


Рисунок 8 - Пример разделения пространства ключей на набор групп RAFT

Также стоит упомянуть о некоторых системах, в которых используются данные алгоритмы, это даст дополнительное понимание о тренде использования предоставленных технологий:

- Технологии на базе алгоритма Паксос[18]:
 - BigTable, Chubby – алгоритм распределенной блокировки (2006г)
 - Google Spanner – геораспределенная СУБД (2012г)
 - Neo4j – графовая NoSQL база данных (2013г)
 - Apache Cassandra – распределенная NoSQL система на базе семейства столбцов (2013г)
- Технологии на базе алгоритма RAFT[19]:
 - LogCabin – распределенная система для хранения метаданных (2015г)
 - etcd – распределенная база данных со строгой согласованностью (2015г)
 - TiKV – распределенная транзакционная база данных “ключ-значение” (2017г)
 - CockroachDB – распределенная NewSQL база данных (2015г)

Приведенные выше описания различных алгоритмов консенсуса, а также примеры их реализации и использования на практике дают

возможность проанализировать и выбрать лучший алгоритм под данную задачу. Основными проблемами алгоритма Паксос, как и было оговорено ранее, являются его сложность для понимания и отсутствие конкретных указаний к реализации, в то время как авторы алгоритма RAFT ставят свое решение в противопоставление алгоритму Паксос и предлагают хорошо описанный алгоритм с исправленными недостатками. Наличие конкретики в описании работы алгоритма RAFT, позволяет легче реализовать его на практике. Также по информации из предоставленного списка проектов видно, что за последнее время все больше проектов отдают предпочтение алгоритму RAFT. Потому при разработке хранилища будет использован алгоритм RAFT с собственной модификацией Мульти-RAFT с разделением пространства ключей по хеш-значению и дополнительным администрирующим кластером. Подробное описание реализации алгоритма консенсуса будет представлено в разделах 4.2 и 4.3.

В конце стоит отметить что протокол RAFT, несмотря на обособленность и на общественное противопоставление алгоритму Паксос по сути является представителем семейства этого же алгоритма, с чертами схожими с Мульти-Паксос.

2.3. Протокол передачи данных

Выбор подходящего протокола передачи данных является важной задачей, потому как, именно на базе выбранного протокола будет происходить общение между узлами сети и именно он будет диктовать формат общения.

Прежде всего стоит выбрать сетевой и транспортный протокол передачи сообщений. Данный этап не представляет сложности, потому как

основной моделью передачи данных на данный момент является TCP/IP, где соответственно сетевым протоколом является IP, а транспортными протоколами являются TCP или UDP, из которых использоваться будет именно TCP так как приоритет уходит на конечную доставку верного сообщения нежели на доставку с потерями, тем более при использовании UDP в конечном счете придется прибегать к реализации TCP-over-UDP.

Выбор протокола прикладного уровня определяет не столько характеристики производительности и гарантии доставки, а сколько формат передачи сообщений и модель общения между узлами, среди которых два основных: двунаправленный обмен сообщениями или модель “Запрос-Ответ”. Рассматривались следующие форматы:

- TCP с собственным бинарным кодеком
- HTTP с собственным бинарным кодеком
- gRPC (поверх HTTP/2) с protobuf кодеком

Разработка собственного кодека для конвертации сообщений в бинарное представление и обратно не представляет собой сложную задачу и в любом случае является обязательной. Тут есть два подхода, полагаться на сериализацию объектов платформы исполнения или же использовать собственный кодек для описания сущностей. По итогу было принято решение для использования комбинированного подхода: использование кодека для сущностей имеющих описание в формате собственного кодека, а все остальное сериализуется стандартным механизмом платформы исполнения, например механизм объектной сериализации java. Такое решение принято для возможности закодировать любое сообщение требуемое для передачи, удобства использования, но при этом сохранения возможности оптимизации

размера и скорости кодирования/декодирования информации, тем самым, соответственно, повышая производительность системы.

Реализация обмена сообщениями с помощью лишь только TCP дает гибкость при построении протокола сообщений между узлами сети. Дуплексная природа обмена TCP сообщениями, позволяет обмениваться информацией без необходимости придерживаться формата “запрос-ответ”, что в свою очередь может быть полезно в некоторых ситуациях. Также стоит добавить что, из перечисленных выше протоколов, данный способ является самым легковесным в плане количества передаваемой информации. Однако данные плюсы также являются главным минусом данного подхода – ввиду асинхронности обмена сообщениями и отсутствия привязки к конкретным “сессиям”, придется с нуля реализовывать некоторый контроллер приходящих и исходящих сообщений для возможности обмена сообщениями связанными друг с другом причинно-следственной связью.

Избавиться от необходимости реализации такого контроллера, помогает использование готовых протоколов поверх TCP, одним из которых является HTTP. Изначально данный протокол использовался лишь для передачи гипертекста, однако затем стал популярным решением для любой, представленной в бинарном виде, информации. Стандарт HTTP предоставляет больше гарантий чем TCP, а также имеет большое количество готовых реализаций клиентов и сервером данного протокола, которые можно переиспользовать. Однако и тут несколько сложностей с которыми необходимо разобраться: сохраняется необходимость в описании всех сообщений и их метаданных в бинарном виде; далеко не все готовые реализации данного протокола поддерживают потоковую передачу сообщений, необходимой для передачи большого количества команд или

большого по объёму состояния узла другому узлу, для синхронизации состояния.

Протокол gRPC решает вышеописанные проблемы. Данный протокол базируется на базе HTTP/2, что сильно повышает производительность передачи данных в сравнении с более старыми версиями. Присутствует поддержка потоковой передачи данных. Сообщения описываются в терминах protobuf, формата конвертируемого в сущности целевой среды исполнения и эффективно представляемые в бинарном виде для передачи данных. Еще одним преимуществом данной технологии является то, что gRPC описывает взаимодействие узлов сети в терминах удаленного вызова процедур (RPC), а именно в этом формате описан протокол RAFT его авторами. Данное свойство упростит реализацию алгоритма. Пример запросов в сети gRPC предоставлен на Рисунке 9.

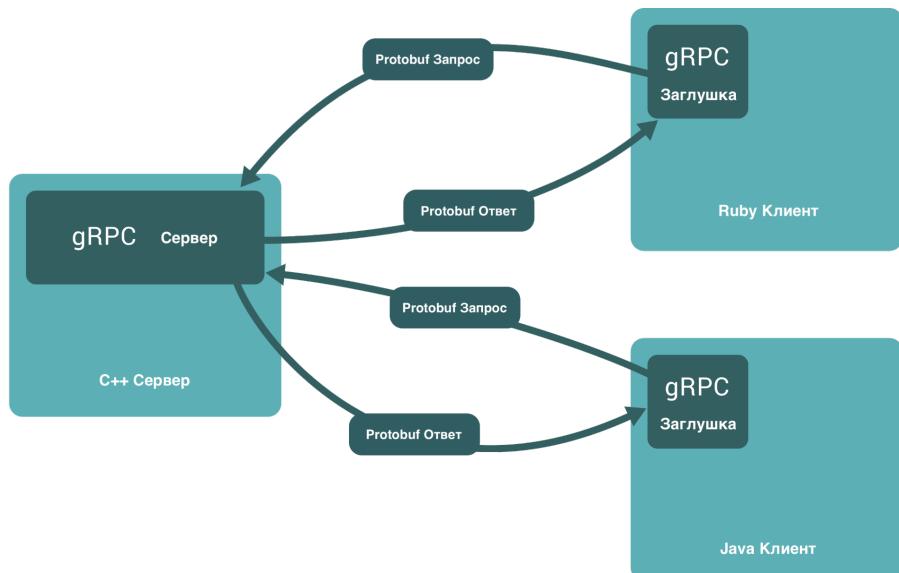


Рисунок 9 - Пример работы протокола gRPC

3. Используемые технологии

3.1. JVM – платформа исполнения

Выбор платформы исполнения влияет на производительность, на способ взаимодействия с подсистемой хранения данных, а также на доступность к использованию разработанного решения для прикладных приложений. Целевой платформой исполнения для решения поставленной задачи была выбрана Java Virtual Machine.

Сравнение производительности исполнения JVM программы и программы, скомпилированной в нативный для ОС байткод, является сложной задачей ввиду того что в различных конфигурациях и задачах результат такого сравнения не будет постоянен. Считается что программа написанная под JVM исполняется медленнее программы оптимизированной и скомпилированной под конкретную ОС, однако стоит заметить что это не всегда является правдой, ведь JVM проводит большое количество оптимизаций во время процесса исполнения, в том числе достаточно часто позитивное воздействие на производительность имеет сборщик мусора. К тому же стоит упомянуть что основные по затратам задачи разработанного решения являются операции с диском и сетью, которые выполняются либо путем вызова нативных функций подсистемы хранения данных через JNI, либо через системные вызовы передачи данных через сетевой контроллер. Из всего выше сказанного, можно предположить, что использование JVM как целевой платформы исполнения не должно повлиять или по крайней мере сильно ухудшить производительность разрабатываемого хранилища.

Важна также и легкость, доступность применения разрабатываемого хранилища разработчиком прикладного ПО. Одним из самых главных свойств JVM является наличие кроссплатформенности, что означает, что если присутствует поддержка JVM и подсистемы хранения данных на некоторой

ОС, там возможна разработка с использованием данного хранилища. Также на фактор доступности к применению влияет популярность целевой платформы исполнения и специфика языков программирования для нее. Согласно отчету компании Tiobe[20] 5 самых популярных JVM языков вместе составляют 12% от общей популярности языков, стоит также учесть что если рассматривать языки программирования, используемые для написания серверных приложений, то рейтинг JVM языков значительно вырастет.

Еще одной причиной для разработки на JVM. Среди проектов для нативной платформы исполнения существует проект Rocksplicator[21], который не выполняет задачи поставленные в данной работе, однако имеет определенную схожесть с разрабатываемым хранилищем. Возможно, но не факт, в случае разработки на нативном языке можно было бы перестроить архитектуру хранилища и уменьшить количество подлежащих разработке модулей, пусть и путем потери некоторых гарантий работы системы.

В довершении стоит сказать, что хоть данный аспект не рассматривается в данной работе, изначальным резоном для разработки данной системы послужила недостаточная производительность сервиса, обсчитывающего статистические данные, выполняющегося на платформе JVM и упирающегося в производительность хранилища OLAP-куба. В последствии, разработанное хранилище планируется интегрировать в вышеупомянутый сервис для увеличения его производительности.

3.2. Scala – язык программирования

При ранее выбранной платформе разработки множество языков на базе которых можно разрабатывать дальнейшее решение ограничивается языками семейства JVM. Выбор будет делаться ограничиваясь 5 самыми популярными языками, среди которых: Java; Kotlin; Scala; Groovy; Clojure. Основными требованиями является:

- Возможность использования разработанного интерфейса в других JVM языках, без особой сложности. Отчасти данное требование добавляет требование о поддержке либо объектно-ориентированного подхода, либо мультипарадигменного с поддержкой объектно-ориентированного.
- Развитая система абстрагирования и наследования.
- Наличие удобных и производительных примитивов работы с конкурентными вычислениями.
- Высокая производительность байткода скомпилированным из конкретного языка.
- Опционально, поддержка функциональной парадигмы.

Языки Groovy и Clojure, отчасти ввиду их динамической типизации, не подходят потому как имеют слабую производительность по сравнению с другими языками, а так же их компиляторы анализируют значительно меньше ошибок на этапе компиляции.

Язык Java является лидеров по производительности среди других языков из списка, однако его возможность абстрагироваться от конкретных реализаций не идеальная, а также наличие, пусть и очень производительных, но слишком базовых примитивов конкурентных вычислений сильно увеличивает вероятность появления сложно исправляемых ошибок связанных с многопоточностью и усложняет настройку механизма многопоточного исполнения.

Из оставшихся языков выбор пал на Scala, ввиду наличия более развитой системы типов и их проверок, лучшей поддержки функциональной парадигмы, а также наличия библиотек контроля эффектов, которые позволяют разрабатывать многопоточные программы с относительной простотой.

3.3. Cats-effect – контроль эффектов

Контроль эффектов – тот или иной способ контроля исполнения побочных эффектов[22] в программе. Может быть, реализован в различных форматах[23]: специальный синтаксис; специальные типы; алгебраический эффект; использование монад.

Библиотека cats поставляется с набором различных инструментов для функционального программирования на Scala, а также определенным набором зависимых классов типов используемых для описания различных значений и операциями над ними.

Библиотека cats-effect является расширением cats и предоставляет иерархию классов типов для контроля эффектов. Определяет операции для работы с отложенными операциями, синхронным и асинхронным исполнением, параллельными вычислениями, безопасностью ресурсов и другим. Также библиотека поставляет собственную реализацию предоставленной иерархии абстракций называемой IO, которая по факту является одной из возможных сред исполнения программы. IO контролирует нити исполнения, пулы потоков, блокирующие и неблокирующие операции, операции со временем и синхронизацией состояний. Иерархия классов типов, поставляемых в последней версии библиотеки представлена на Рисунке 10.

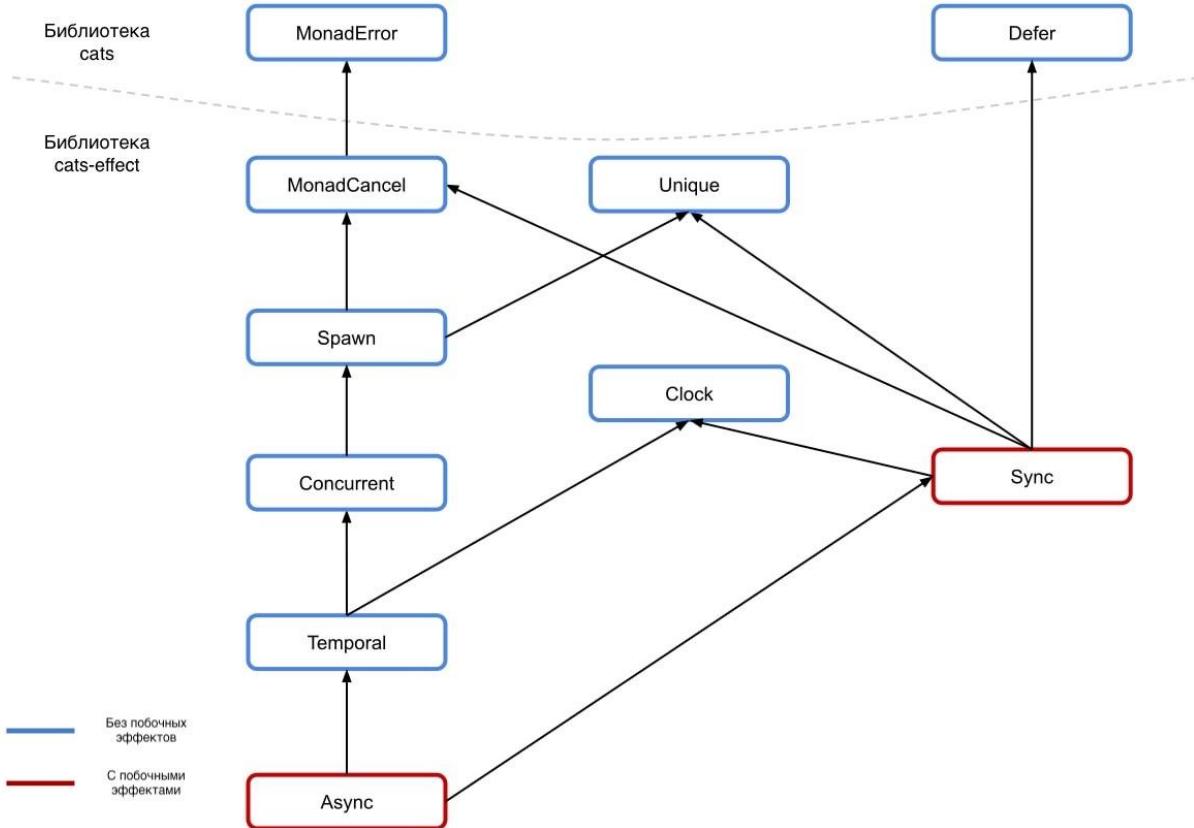


Рисунок 10 - Иерархия классов типов библиотеки cats-effect[24]

3.4. Fs2 – обработка потоков данных

Разрабатываемое хранилище также будет содержать большое количество потоковой обработки данных. По мере того как алгоритмы обработки данных становятся сложнее, появляются сложности в описании процесса потоковой обработки в виде обычного цикла и обычно прибегают к некоторому декларативному интерфейсу описания потока данных. Среди различных библиотек для JVM есть много тех которые предлагают такой интерфейс, среди прочих: reactive-streams, reactor, akka-streams, rxjava и прочие. Однако ввиду использования библиотеки cats-effect и других typelevel библиотек, выбор библиотеки управления потоков данных пал на fs2.

Библиотека fs2 предоставляет две основные возможности[25]:

- Конструировать различные алгоритмы обработки и генерации потока данных, с возможным использованием встроенных эффектов, таких как например асинхронное исполнение.
- Изменять один или несколько потоков данных используя заранее определенный набор операций над потоками.

Также fs2 предоставляет:

- Примитивы для конкурентных операций, такие как: рассылку, сигнал, очередь.
- Модуль fs2-io для работы с операциями ввода/вывода, такими как: передача данных по TCP/UDP, операции с консолью, операции с файлами.
- Модуль fs2-timeseries для работы с временными рядами.
- Взаимодействие с библиотекой бинарного кодирования scodec.

3.5. Protobuf – описание протокола

Protocol Buffer (сокращенно protobuf) – формат описания сообщений, предложенный компанией Google в качестве замены формату XML. Формат описывает сообщения как структуру “ключ-значение”, где ключ это заранее заданная целочисленная константа, а значение имеет один из определенных форматом типов, таких как: целочисленное значение, набор байтов, строка и другие. Описание сообщений происходит с помощью описания шаблона в файле с расширением proto, который затем один из выбранных компиляторов protobuf использует для генерации представления данного сообщения в целевой язык программирования. Например, компилятор protobuf для языка java генерирует java класс, а для языка haskell результатом будет haskell record.

Данный подход позволяет кроссплатформенно описывать формат передаваемых сообщений, а также экономить занимаемое место и скорость

обработки сообщения по сравнению с XML[26]. Также, именно данный формат используется в протоколе обмена сообщениями gRPC.

3.6. Netty – клиент-серверный фреймворк

Netty – клиент/серверный фреймворк для разработки сетевых приложений на языке Java. В фреймворке используется неблокирующие примитивы языка Java, а также добавлено множество инструментов для упрощения работы с сетевыми сокетами.

Помимо наличия возможности разработки обычных сетевых приложений прикладного уровня, в поставку также включена поддержка таких технологий как SSL/TLS, HTTP, HTTP/2, WebSockets, DNS, Google Protocol Buffers и других.

3.7. RocksDB JNI API – интерфейс подсистемы хранения

RocksJava – проект разработки высокопроизводительного Java драйвера для RocksDB. Интерфейс подсистемы хранения разбит на три уровня:

- Java классы реализующие RocksJava API. Только с данным слоем непосредственно взаимодействует программа написанная на языке Java.
- Код слоя JNI написанный на C++, который предоставляет связь между Java API и RocksDB.
- RocksDB сам по себе написанный на C++ и скомпилированный в нативную библиотеку, используемую в JNI.

4. Разработка хранилища

4.1. Разработка API для RocksDB

Разработка API для RocksDB обусловлена низкоуровневым характером поставляемого RocksJava интерфейса взаимодействия с базой данных.

Одним из основных моментов, которые стоит учесть это управление выделяемыми ресурсами. Все объекты RocksDB создаваемые при использовании системы, такие как объекты конфигурации операций, объекты баз данных, объекты слепков состояния и другие, должны быть, после завершения их использования, закрыты и ресурсы, ими занятые, должны быть завершены. Такое поведение обусловлено ручным характером управления памятью в C++ и отсутствия однозначно правильного способа использования механизма очистки нативного объекта в паре с алгоритмом сборки мусора используемого в JVM. На всякий случай RocksJava переопределяет метод деструкции у создаваемых объектов, на случай их очистки сборщиком мусора без предшествующей явной очистки пользователем, однако такой подход считается небезопасным[27] и предоставлен скорее во вспомогательном ключе. Данный аспект работы с драйвером обработан путем введения отдельного метода работы с нативными объектами под названием nativeResource, который создает объект Resource, предоставляемый библиотекой cats-effect и описывающий создание и закрытие/очистку некоего ресурса в рамках предоставленного эффекта, в данном случае таким ресурсом будет любой представитель org.rocksdb.AbstractNativeReference. Resource является монадой, а значит имеет собственный определенный метод flatMap/bind позволяющий определять инициализацию множества ресурсов, которые будут освобождены в обратном порядке их объявления.

Весь интерфейс работы с RocksDB описан в рамках алгебры с зависимостью на абстрактный тип высшего порядка[28] (higher-kinded type parameter), который должен будет удовлетворять классу типа Async. Данный подход позволяет использовать необходимый эффект контроля операций ввода-вывода и конкурентного исполнения, например в сколько потоков одновременно может вестись запись или на какой пул поток должны

переводится блокирующие операции. Все длительные операции с базой данных считаются блокирующими и помещаются внутрь блока оператора `blocking`.

Еще одной из проблем работы с низкоуровневым интерфейсом является работа с бинарным представлением данных. На данный момент решено определить собственный кодек сериализации/десериализации данных из объектов языка Scala в набор байт и обратно. Для того чтобы объект можно было записать/прочитать из базы данных на данный тип должна быть предоставлена реализация классов типов `Encoder` и `Decoder`. Для обоих классов представлены базовые реализации на преобразование строк и объектов для которых предоставлен кодек определенный в библиотеке `scodec`. `Encoder` является контравариантом, а `Decoder` функтором.

Помимо обычных методов взятия и записи значения, в интерфейсе присутствуют методы работы со слепками базы данных, восстановления из сохранных точек, проверка наличия ключа и пакетные операции. Также присутствует возможность остановки деятельности базы данных, без потери записываемых значений, на некоторое время. Данный механизм полезен при глобальном изменении состояния базы данных, например при восстановлении состояния из слепка базы данных, полученного с удаленного узла. Полный список операций с их названиями приведен ниже:

- Взятие значения по ключу (`get`)
- Запись значения по ключу (`put`)
- Удаление значения (`delete`)
- Вероятностная проверка наличия ключа (`isEmpty`)
- Пакетное взятие значений (`batchGet`)
- Пакетная запись значений (`batchPut`)
- Синхронизация операций с дисковым состоянием БД (`flush`)
- Создание слепка базы данных (`backup`)

- Получение идентификаторов существующих слепков (getBackups)
- Восстановление базы данных по идентификатору слепка (restoreDB)
- Взятие значения по ключу из таблицы метаинформации БД (getProperty)
- Форсирование процесса сжатия данных на диске (compact)
- Создание слепка базы данных в конкретной директории (checkpoint)
- Остановка фоновых процессов и вызовов к базе данных (dbGenStop)
- Замена пути к файлам базы данных (hotSwap)

Программный код связанный с модулем работы с RocksDB предоставлен в Приложении А.

4.2. Реализация алгоритма RAFT

Ключевой частью разрабатываемой системы являются алгоритмы выбора лидирующего, репликации данных и достижения распределенного консенсуса по состоянию системы. Реализация алгоритма RAFT включает в себя все данные алгоритмы, так как всегда присутствует информация о текущем лидере, а процесс достижения консенсуса подразумевает наличие репликации данных.

Корректная реализация алгоритма консенсуса является чрезвычайно сложной задачей, потому было принято адаптировать готовое решение, примерно подходящее по используемым технологиям и стилистике написания программного кода. Было выбрано готовое решение raft4s[29], написанное на базе предыдущей версии библиотеки cats-effect 2.0, а также, стандартном для java, netty-grpc. Основными направлениями разработки являлись:

- Изменения и доработка доменной модели для обеспечения лучшей абстракции от эффекта исполнения и расширяемости алгоритма.

Например, был добавлен новый ковариантный параметр типа для большинства интерфейсов алгоритма, для возможности указания конкретного типа машины состояний используемой в данном алгоритме.

- Миграция с cats-effect 2.0 на cats-effect 3.0. Иерархия классов типов между этими версиями поменялась значительно[30], соответственно требуется значительный рефакторинг существующего кода для оптимального разделения требуемых классов типов в каждом из компонентов, а также изменения работы с монадой IO, которая была значительно переработана в новой версии.
- Заменены модули генерации классов из protobuf шаблонов и обмена сообщениями gRPC со связки scalapb и io.grpc.netty на fs2-grpc. Данное изменение позволило использовать абстрактный параметр высшего порядка для описания эффекта исполнения без обязательного шага конвертации в/из CompletableFuture, а также поставляемую с fs2 абстракцию Stream для описания потоковой передачи информации в gRPC.
- В доменную модель добавлены сущности дублирующие сообщения описываемые в protobuf, но не зависящие на них. Именно данные сущности используются внутри системы, а на промежутке взаимодействия с сетевой частью сущности конвертируются с помощью заранее определенной абстракции Transformer.
- Изменение формата сообщений для передачи слепков состояний и журналов между узлами. Если предположить достаточно большое отставание между различными узлами системы, передаваемые изменения в базе данных лидирующего узла могут превысить максимальный размер сообщения gRPC в 4МБ. Данная проблема

решается оптимизацией размера передаваемого сообщения и использованием потоковой передачи частей сообщения.

- Понадобились различные алгоритмические усовершенствования некоторых аспектов реализации алгоритма репликации данных и выбора лидирующего для достижения лучшей производительности и более редкого случая наличия более 1-го лидера в системе. Среди прочего было оптимизировано: ожидание перед изменением состояния узла в состояния кандидата; фиксация конкретных команд в журнал изменений; улучшенная работа с примитивами общего состояния. Также были проведено улучшение идиоматической составляющей в виде использования более строгих и абстрактных методов изменения состояния и передачи данных, исправлено повсеместное использование громоздких конструкций “синтаксического сахара”.

Помимо улучшений и дополнений к существующей реализации алгоритма RAFT, проводились дополнительные изменения и улучшения для работоспособности алгоритма при данном сценарии использования.

Был написан модуль работы с zip-архивами для передачи слепков состояния от одного узла другому, обусловлено это рекурсивной структурой хранения файлов базы данных и необходимости передачи состояния в одном сообщении. Данный модуль также написан в стиле абстракции от эффекта исполнения с соответствующим использованием класса типа Async для различных операций, как блокирующих, так и нет.

Команды записи, чтения, удаления и их пакетные версии используемые для изменения или получения состояния из базы данных RocksDB имеют определенный бинарный кодек, позволяющий эффективно кодировать и декодировать данные команды в сообщения. Для возможности кодирования

любой из представленных команд используется гибридный подход сериализации.

Описанные ранее модификации и дополнения складываются в работающую реализацию алгоритма RAFT, подходящую под решение поставленной задачи. Ниже предоставлено описание основных компонентов алгоритма:

- NodeState – тип локального узла в кластере RAFT, являющийся одним из трех: FollowerNode; CandidateNode; LeaderNode. На каждое отдельное состояние определены операции поведения узла на те или иные события, такие как истечение таймера на получение сообщения от лидера или запроса на голосование за нового лидера.
- ClusterConfigStorage – Сущность хранящая конфигурацию нынешнего кластера и набор его участников.
- LeaderAnnouncer – Компонент описывающий действия для объявления о новом лидере и о перезаписи старого.
- LogPropagator – Компонент описывающий действие репликации данных другим участникам кластера.
- LogStorage – Сущность хранящая поступающие команды на получение или изменение состояния. Поддерживает операции получение и записи конкретной команды по конкретному индексу, а также отчистку команд из хранилища попадающих в заданный диапазон. Хранилище команд поддерживает механизм компрессии, который конфигурируется предикатом из LogCompactionPolicy.
- ClientManager – Сущность контролирующая имеющихся сетевых клиентов и передачи им запрашиваемых запросов.
- StateStorage – Сущность сохранения информации о состоянии данного локального узла. Информация содержит такую информацию как:

нынешняя эпоха; за кого данный узел голосовал на прошлом этапе голосования и номер последней зафиксированной команды.

- StateMachine – Сущность применения команд на чтение или запись в хранилище. Данный интерфейс реализуется с конкретной подсистемой хранения данных, чтобы сохранять в нее значения с достигнутым консенсусом.
- Log – Сущность объединения вышеописанных хранилищ и компонентов, выполняет различные задачи по управлению состоянием журнала.
- Cluster – Сущность управления локальным узлом в кластере: инициализация узла; присоединение к кластеру; выход из кластера; и другие.
- Raft – Сущность управления состоянием локального узла, его журналом команд, состоянием хранилища, конфигурацией, метаинформации кластера. Так же тут начинают обрабатываться запросы поступающие от других узлов кластера или команды от пользователя.

Дополнительно для облегченного понимания внутреннего устройства системы была добавлена документация формата ScalaDoc описывающая ключевые компоненты и их методы, описанные выше.

Программный код связанный с модулем работы алгоритма RAFT предоставлен в Приложении Б.

4.3. Модификация алгоритма консенсуса

По причинам описанным в разделе 2.2 использование только лишь обычного алгоритма RAFT становится нецелесообразно. Алгоритм требует модификаций или наличие послаблений в требованиях к согласованности. В данной работе было сделано и то и другое.

Возможность релаксации требований к согласованности операций в кластере для увеличения производительности операции чтения и разгрузки лидирующего узла описывается еще в оригинальной публикации авторов алгоритма RAFT[4]. Суть состоит в делегировании обработки команд на чтение любому типу узла, в том числе подписчикам. Тем самым будет нарушена линеаризуемость порядка команд и узел может вернуть устаревшую информацию, однако если такие гарантии не требуются, то таким образом можно сэкономить большое количество обращений к лидирующему узлу.

Второй оптимизацией становится введение модификаций в построение кластера RAFT. Применяется, ранее кратко обговоренный, подход Мульти-RAFT[17], суть которого заключается в разбиении пространства ключей хранилища на каждом из узлов и присвоения ответственности за каждый из выделенных наборов ключей одной группе, где группой является отдельный RAFT кластер. Для оптимизации использования ресурсов, необходимо ввести ограничение на количество лидирующих позиций, которые может занять один узел сети. Так как в данном случае разбиение будет проводиться на то же количество групп, что и количество узлов в сети, то максимальным числом установленных лидеров на одном узле является 1. Рассмотрим процесс старта такого кластера и обработку команд подробнее.

На диаграммах представленных ниже будут использоваться следующие обозначения:



Рисунок 11 - Обозначения на диаграммах состояний Мульти-RAFT

При старте кластера мы имеем ограниченное количество узлов имеющих информацию о всех других узлах, путем конфигурации. Первая

задача – создание оркеструющего кластера, наверняка данный этап мог быть эффективнее выполнен стандартным алгоритмом Паксос, однако этот этап повторяется нечасто, а реализация еще одного алгоритма консенсуса является слишком затратным процессом, потому используем алгоритм RAFT. После выбора лидирующего узла, он подсчитывает общее количество узлов в системе и создает такое же количество RAFT групп и каждой присваивает идентификатор и соответствующий ей остаток который используется для определения зоны ответственности за те или иные значения. Данная информация рассыпается подписчикам и в порядке очереди инициализируются новые RAFT кластеры, соответствующие своим группам.

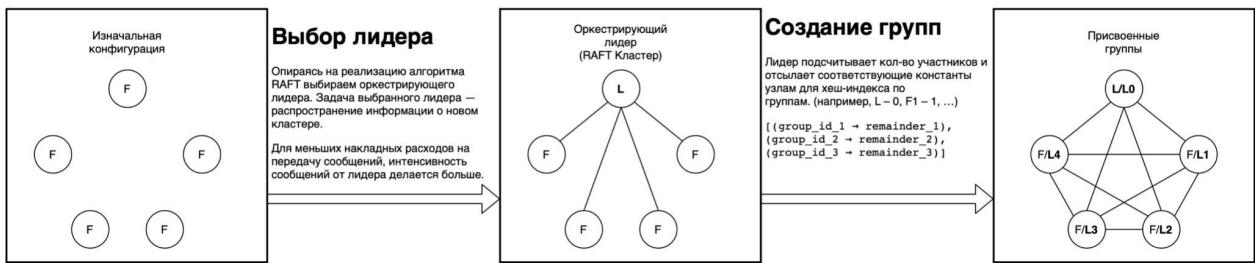


Рисунок 12 - Описание процесса старта кластера

Обработка команд изменения состояния выполняется в три этапа: нахождение ответственного узла на запись данного значения; пересылка ему данной команды; установление консенсуса между узлами по поводу данного значения. Вычисление ответственного за данный ключ происходит путем расчета хеш-значения ключа по заранее заданному алгоритму и делению этого значения на количество узлов в сети, получившийся остаток используется как ключ в таблице RAFT групп у каждой из которых известен соответствующий лидер. В случае совпадения локального узла и узла ответственного за данный ключ, пересылка не требуется. Стоит добавить что полезным будет свойство линеаризуемости порядка операций, хотя бы в пределах локального узла. Для достижения данного свойства, даже при

условии несовпадения ответственного за узел и локального узла, мы сохраняем обновленное значение в локальный краткосрочный буфер, до момента его фиксации путем репликации данных от лидера. В случае если данное значение будет запрошено до его фиксации, данный узел все равно сможет отдать его пользователю.

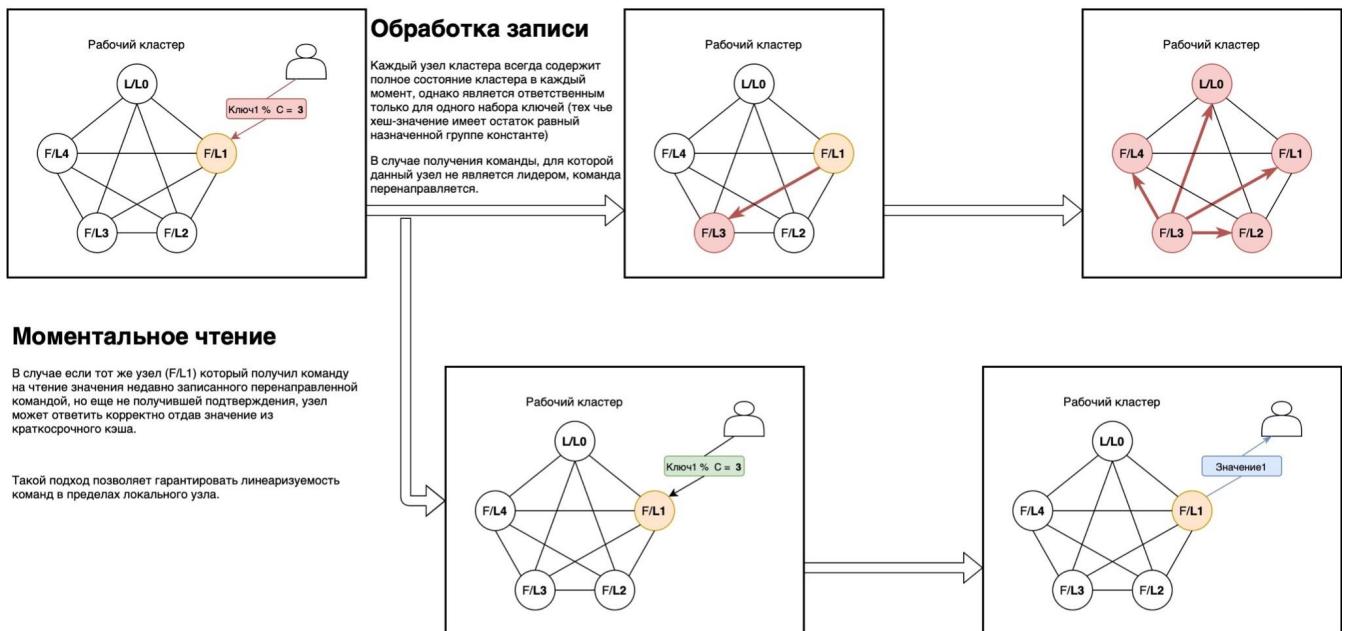


Рисунок 13 - Описание процесса изменения состояния кластера

Обработка команд получения состояния имеет несколько вариантов исполнения. При запросе значения у узла, который по совместительству является и ответственным за это значение, ответ становится моментальным и по факту является операцией чтения значения из подсистемы хранения данных, в данном случае RocksDB.

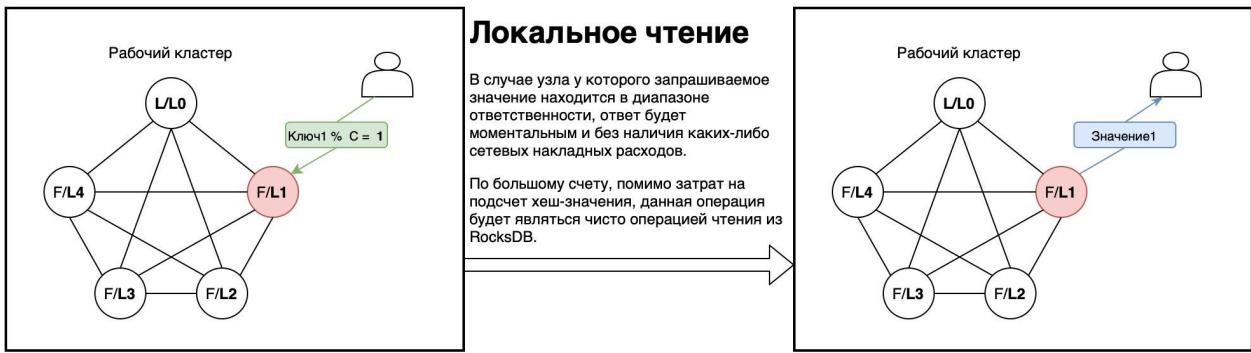


Рисунок 14 - Описание процесса чтения из локального узла

Такая же операция выполняется в случае когда для данного значения, узел не является ответственным, однако пользователь в запросе не требует строгой согласованности, и понимает, что может получить устаревшие данные о значении. Оба описанных сценария позволяют избежать использования сетевого запроса при чтении значения, что значительно ускоряет скорость обработки данной команды при их высокой частоте. Однако может присутствовать необходимость в получении строго согласованного значения по заданному ключу, в таком случае команда на запрос значения пересыпается ответственному узлу, после получения соответствующего значения оно обновляется в локальном хранилище и затем отдается пользователю, его запросившего.

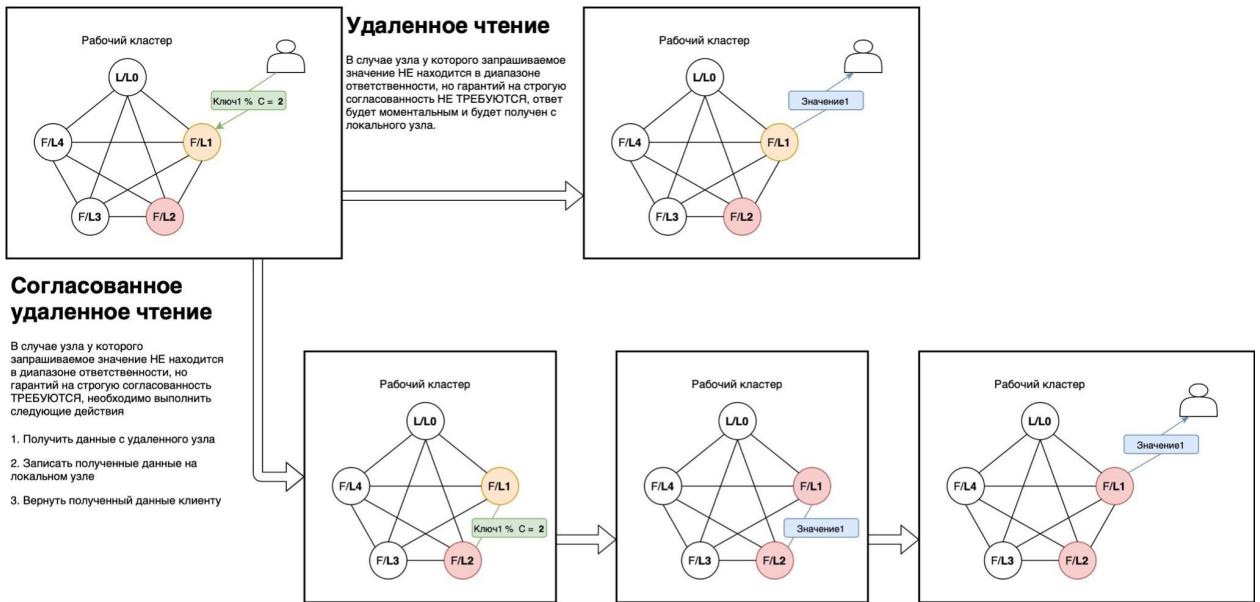


Рисунок 15 - Описание процесса чтения из удаленного узла

При изменении состава участников в кластере Мульти-RAFT, производится схожая операция присоединения или удаления участника сети, как и в обычном RAFT с единственным отличием, в начале данную операцию выполняет оркестрирующий кластер, а затем обновленная информация об участниках рассыпается и все остальные RAFT группы, которые затем корректно обрабатывают изменение. В конце, одна RAFT группа либо создается, либо удаляется в зависимости от изменения количества узлов по сравнению с прошлой конфигурацией.

Программный код связанный с проведенными модификациями Мульти-RAFT предоставлен в Приложении В.

4.4. Разработка Scala API

Наличие Scala API в разработанной системе, вместо только запущенного в фоновом режиме процесса синхронизации узлов, обуславливается необходимостью применения механизма обратного давления в случае превышения максимально возможной пропускной способности кластера, разграничение характера операций на строго согласованные,

синхронные и не строго согласованные, асинхронные. Также наличие такой прослойки дает возможность с удобством использовать разработанное хранилище в существующих проектах с относительно высокоуровневым API.

Данный программный модуль не вносит никаких изменений в логику работы хранилища, а лишь предоставляет идиоматично реализованный интерфейс со следующими методами:

- Получение значения по ключу (read)
- Получение списка значений по списку ключей (readMany)
- Запись значения (write)
- Запись списка значений (writeMany)
- Удаление значения по ключу (delete)
- Удаление значений по списку ключей (deleteMany)
- А также строго согласованные варианты вышеперечисленных методов.

Все такие операции имеют префикс “consistent-”.

Также был реализован дублирующий вышеописанную функциональность, класс на Java, для удобства использования данного хранилища на любом JVM языке. Хотя стоит заметить, что для корректной работы будет необходимо наличие стандартной библиотеки scala в classpath.

Программный код связанный с реализацией Scala API предоставлен в Приложении Г.

5. Тестирование производительности

Поскольку поставленной целью было повышение производительности систем с высокочастотной запись и чтением из базы данных, путем разработки хранилища с характеристиками производительности выше чем у аналогов, необходимо протестировать производительность разработанной системы и провести сравнительный анализ результатов.

Сравнение разработанной системы будет проводиться против следующих двух систем: Redis и Aerospike. Причинами такому выбору стали:

- Общая специализация обеих систем под обслуживание высокочастотных операций.
- Присутствие различий в данных системах между собой, что дает более полную картину.
- Как было ранее оговорено в разделе 3.1, система которой потребовалась разработка нового хранилища, до этого работала при помощи либо одного, либо другого из выбранных хранилищ.

Тестирование производительности проводилось путем измерения потраченного времени на обработку установленного количества запросов/команд чтения или записи. Для составления полной картины, были выбраны следующие параметры для варьирования между различными этапами тестирования, а именно: количество команд необходимых для обработки; тип команды – чтение или запись; тип выполнения команды – единичный, параллельный, пакетный; для разработанного хранилища также выбирается тип согласованности – строгая или “в конечном счете”.

Вариация размера количества значений для обработки составляла от 1000 до 100000. Ключи и значения являются случайными байтовыми

строками, предварительно сгенерированные и сохраненные в оперативной памяти. Степень параллелизма при параллельной записи равнялась 5. Тестирование проводилось на системе следующей конфигурации:

- Твердотельный накопитель (SSD) Скорость случайной записи 1.6 ГБ/с. Скорость случайного чтения 2.5 ГБ/с.
- Центральный процессор Intel Core i9 2.4 ГГц 8 ядер (16 vCPU)
- Оперативная память, объемом 16 ГБ и частотой 2400 МГц
- Все хранилища тестировались в нативном представлении, без контейнеризации.
- Тестирование проводилось с помощью утилиты ScalaMeter, предоставляющей функциональность запуска теста в отдельной виртуальной машине Java, прогрева виртуальной машины, генераторов тестовых данных, а также множественный запуск одного теста с последующим получением среднего результата.

Рассмотрим результаты тестирования хранилища Redis:

Таблица 2 - Тестирование производительности Redis

Кол-во значений	Единичная запись	Параллельная запись	Пакетная запись	Единичное чтение	Параллельное чтение	Пакетное чтение
1,000	1,890.74	537.78	19.59	2,069.23	656.21	21.44
11,000	21,453.30	5,491.37	63.62	27,210.45	5,619.82	81.34
21,000	38,597.76	10,172.60	109.94	45,024.24	10,694.46	144.90
31,000	53,960.80	14,479.41	165.09	62,680.80	17,055.83	169.89
41,000	84,060.27	18,273.60	198.55	80,414.22	18,336.72	218.28
51,000	124,798.93	23,597.34	242.71	90,900.13	21,426.36	273.03
61,000	128,887.70	28,292.24	281.54	113,697.67	25,688.12	373.18
71,000	150,955.41	32,409.53	374.04	123,603.79	31,570.20	555.25
81,000	175,190.80	36,123.43	404.93	141,744.33	34,862.95	472.36
91,000	171,902.07	41,591.80	497.66	159,368.45	38,621.23	547.36
101,000	192,272.23	48,634.51	502.08	204,033.85	51,836.93	499.99

Построим графики зависимости затраченного времени от количества значений. Отдельный для записи, отдельный для чтения. Для удобства ось ординат имеет логарифмический масштаб.

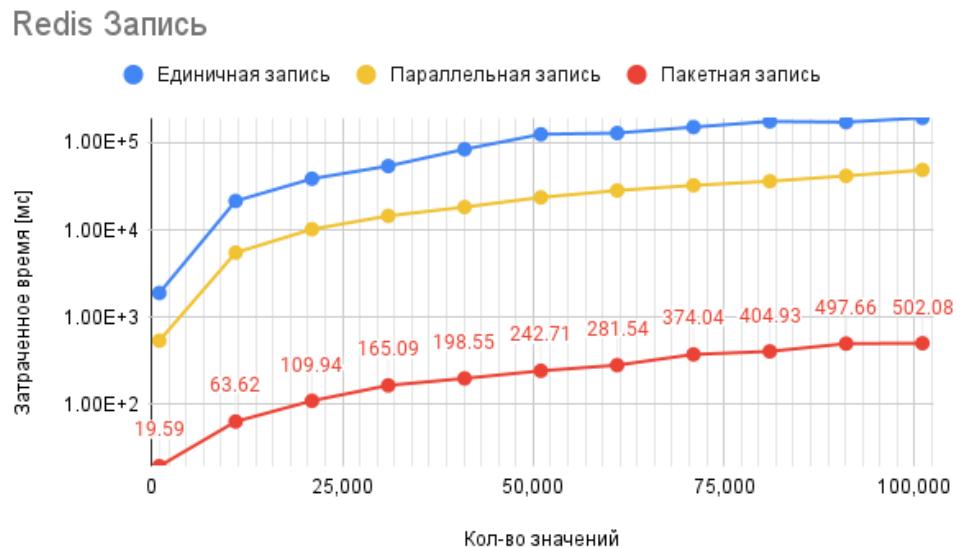


Рисунок 16 - Тестирование записи значений в Redis

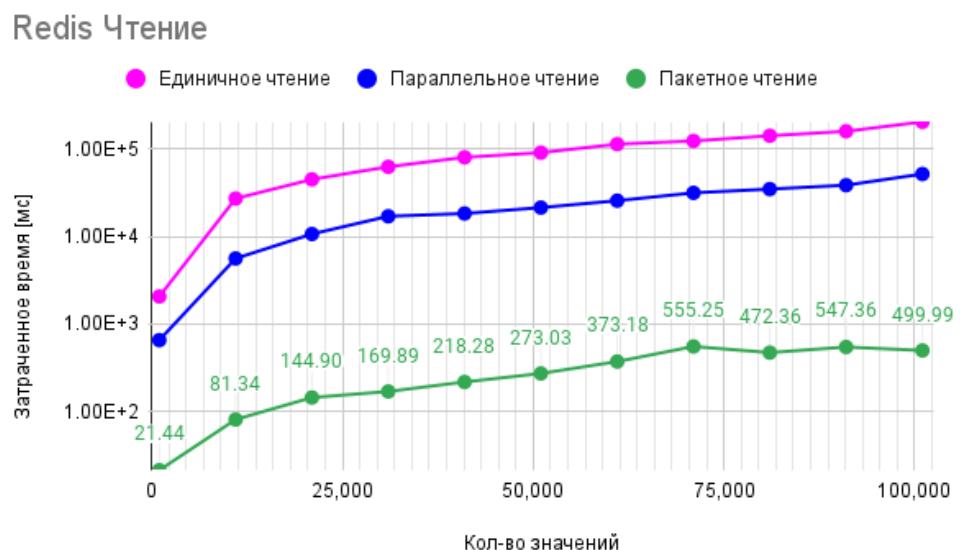


Рисунок 17 - Тестирование чтения значений в Redis

Основной вывод который можно сделать на данном этапе, это то что, даже при условии однопоточной обработки команд в Redis, параллельное исполнение команд дает значительный прирост в скорости, однако наибольшей оптимизацией является пакетная обработка данных. В случае с Redis разница между единичной пересылкой команд и пакетной может составлять до 3-х порядков величины. Учитывая то как устроен Redis протокол и обработка команд внутри базы данных, можно предположить что такую дополнительную задержку вызывает использование сетевого стека.

Такое же тестирование было проведено и с системой Aerospike. Рассмотрим результаты ниже:

Таблица 3 - Тестирование производительности Aerospike

Кол-во значений	Единичная запись	Параллельная запись	Единичное чтение	Параллельное чтение	Пакетное чтение
1,000	1,904.44	831.74	1,847.98	665.06	44.17
11,000	20,306.12	9,172.86	19,241.30	7,714.69	435.69
21,000	41,487.76	17,654.33	37,222.58	14,850.77	831.80
31,000	57,618.99	25,168.90	56,242.28	22,357.49	1,201.21
41,000	70,185.34	32,765.24	72,496.55	28,906.54	1,559.54
51,000	94,244.08	41,022.99	94,126.76	33,798.92	1,937.36
61,000	118,091.57	48,958.22	116,028.53	42,692.58	2,254.38
71,000	136,051.28	54,639.51	138,676.62	50,582.87	2,566.69
81,000	157,523.76	58,930.06	129,281.20	56,424.11	2,958.49
91,000	172,052.14	98,877.69	143,828.65	71,442.45	3,175.02
101,000	194,129.36	77,111.05	158,907.65	72,681.24	3,529.65

В отличие от Redis, Aerospike не поддерживает пакетную запись множества ключей, потому присутствуют результаты только для единичной и параллельной записи.

Построим графики результатов тестирования Aerospike по тому же принципу, что и раньше.

Aerospike Запись

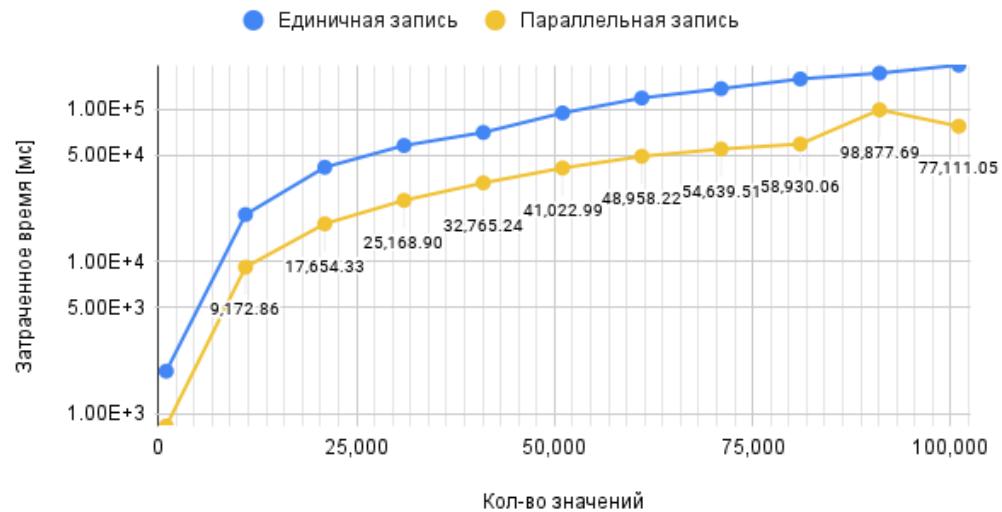


Рисунок 17 - Тестирование записи значений в Aerospike

Aerospike Чтение

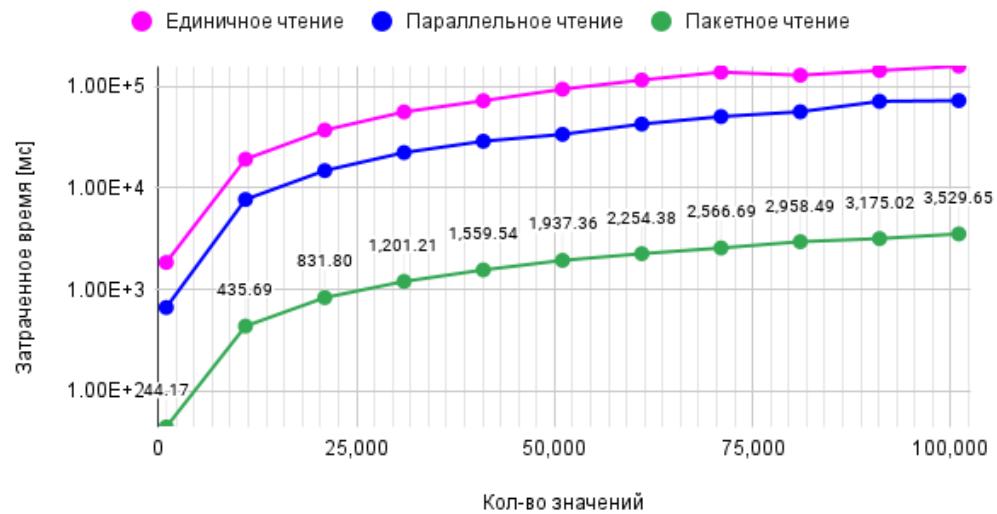


Рисунок 19 - Тестирование чтения значений в Aerospike

Результаты тестирования разработанного хранилища при ослаблении требований к согласованности получаемых значений, требуется лишь согласованность в конечном счете:

Таблица 4 - Тестирование производительности разработанного хранилища

Кол-во значений	Единичная запись	Параллельная запись	Пакетная запись	Единичное чтение	Параллельное чтение	Пакетное чтение
1,000	11.60	11.49	3.56	3.63	3.73	3.84
11,000	133.38	127.83	30.75	44.06	38.46	39.04
21,000	273.09	251.02	67.31	96.96	150.96	83.57
31,000	319.35	347.64	106.88	139.55	198.16	137.12
41,000	413.10	504.28	128.83	179.63	233.79	171.02
51,000	508.66	537.90	146.25	214.73	159.26	188.15
61,000	593.72	580.17	191.45	310.06	415.84	245.14
71,000	718.02	609.04	231.14	313.73	343.65	272.73
81,000	739.30	787.51	253.40	362.19	488.90	295.09
91,000	844.60	1,004.70	317.22	413.91	419.97	353.76
101,000	978.09	1,127.31	354.95	478.57	593.74	356.51

График производительности системы на этот раз строится с линейной шкалой для оси ординат.

Не строго согласованная запись

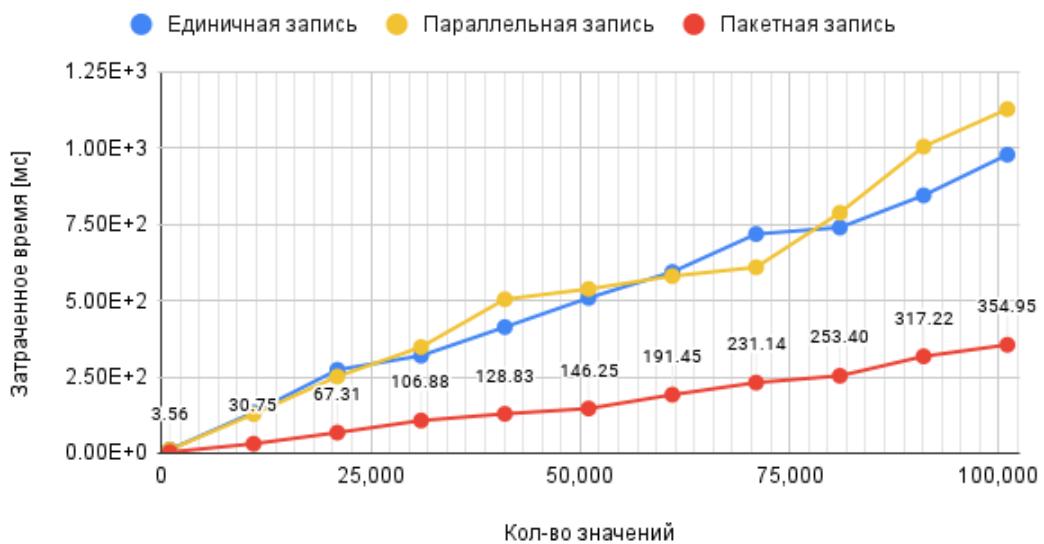


Рисунок 20 - Тестирование записи значений в разработанном хранилище

Не строго согласованное чтение

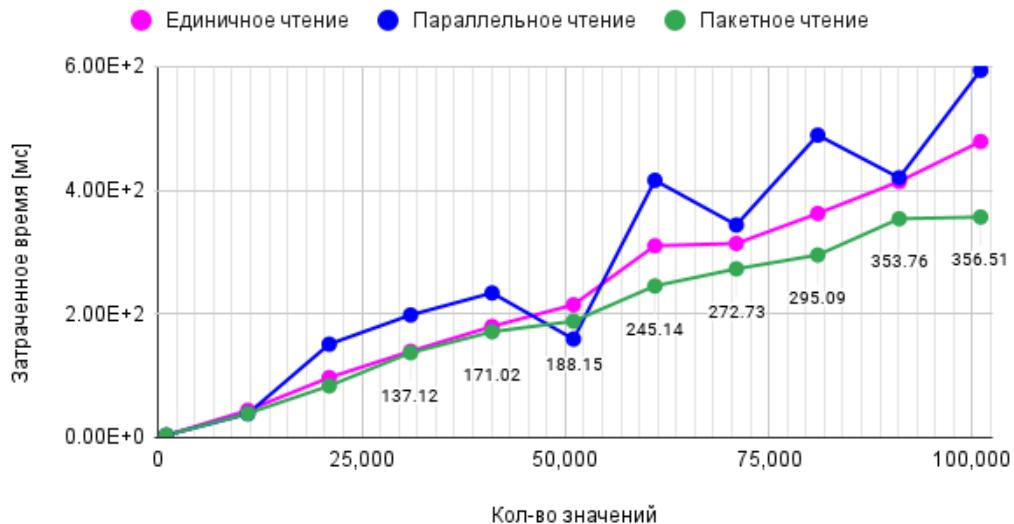


Рисунок 21 - Тестирование чтения значений в разработанном хранилище

Из данных результатов можно выделить два интересных наблюдения. Во-первых, разница между пакетными операциями и единичными имеют разницу, но при чтении практически не различаются. Во-вторых, производительность параллельных операций чтения и записи в некоторых случаях становится медленнее простой последовательной единичной записи, это можно объяснить необходимостью в постоянных переключениях контекста и, возможно, использования некоего однопоточного ресурса внутри базы данных RocksDB.

Измерения производительности рассылки информации по кластеру в согласованном режиме проводились в другом ключе, в данном случае тестировалось отношение скорости передачи значений относительно размера пакетного буфера, используемого при передаче. Этот подход позволит определить оптимальный размер буфера для синхронизации состояний между узлами сети. Для обеспечения адекватной длительности выполнения теста и

получения репрезентативных результатов, во время проведения теста, значения размера буфера и количество передаваемых значений будут меняться относительно друг друга. Табличные результаты представлены ниже:

Таблица 5 - Тестирование производительности согласованной передачи команд

Размер пакета	Затраченное время [мс]	Кол-во команд	Отношение [команд/мс]
10	2,670	10,000	3.75
1,000	1,855	100,000	53.91
5,000	1,290	100,000	77.52
7,500	3,457	500,000	144.63
10,000	6,008	1,000,000	166.44

Построенный график по результатам тестирования:

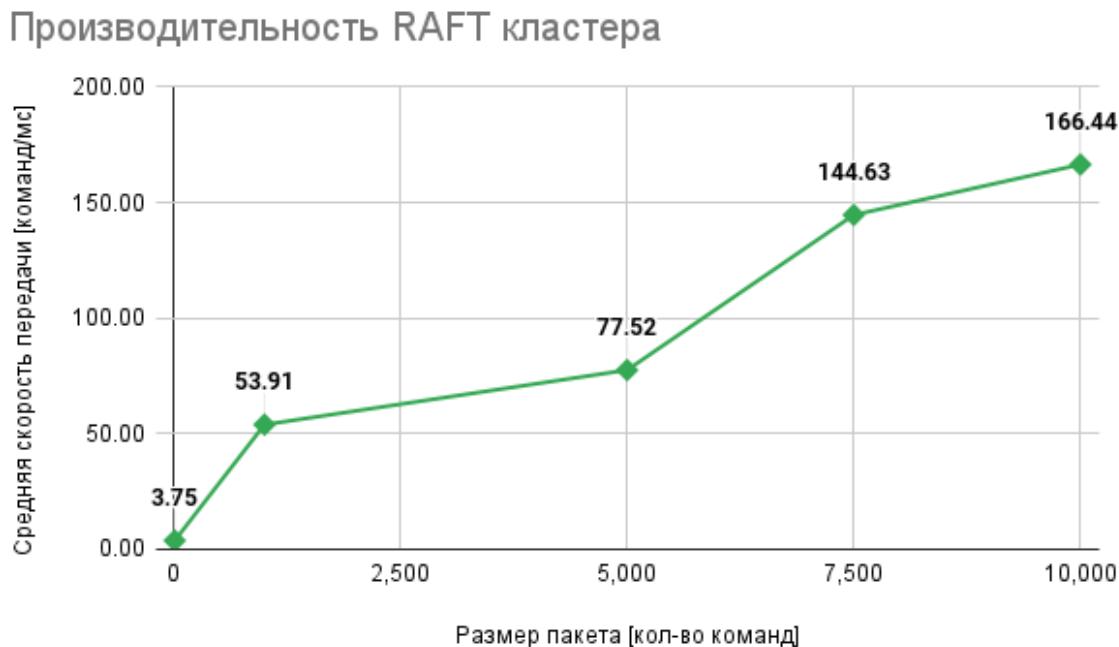


Рисунок 22 - Производительность разработанного RAFT кластера

Анализируя полученные данные можно заметить что увеличение скорости передачи сообщений имеет линейный характер и зависит от размера

пакета на каждом моменте передачи сообщений. Однако повышать размер пакета необходимо с осторожностью, по двум причинам. Во-первых, максимальный размер сообщения gRPC составляет 4 МБ и необходимо не превышать данный размер при пересылке сообщения. Во-вторых, увеличение размера пакета увеличивает задержку распространения значений по кластеру, что приводит к более поздней их доставки до других узлов. График такой задержки предоставлен ниже:



Рисунок 23 - Задержка отсылки пакетов в зависимости от их размера

Предположительно, оптимальный размер пакета лежит в промежутке между 7000 и 10000 команд за раз.

Протестировав каждый из сервисов по отдельности необходимо провести сравнительный анализ всех решений. Ниже представлены графики объединения лучших режимов, по производительности, каждого из хранилищ. Для разработанного решения представлены две линии тренда, для каждого из режимов работы.

Сравнение производительности на запись

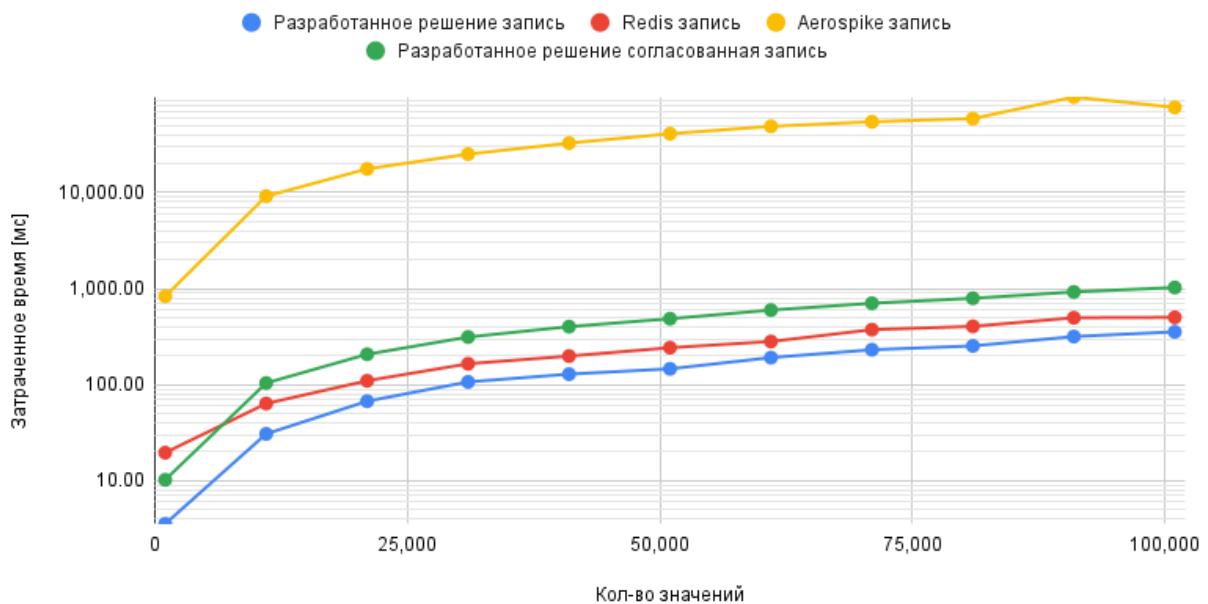


Рисунок 24 - Сравнение производительности на запись

Сравнение производительности на чтение

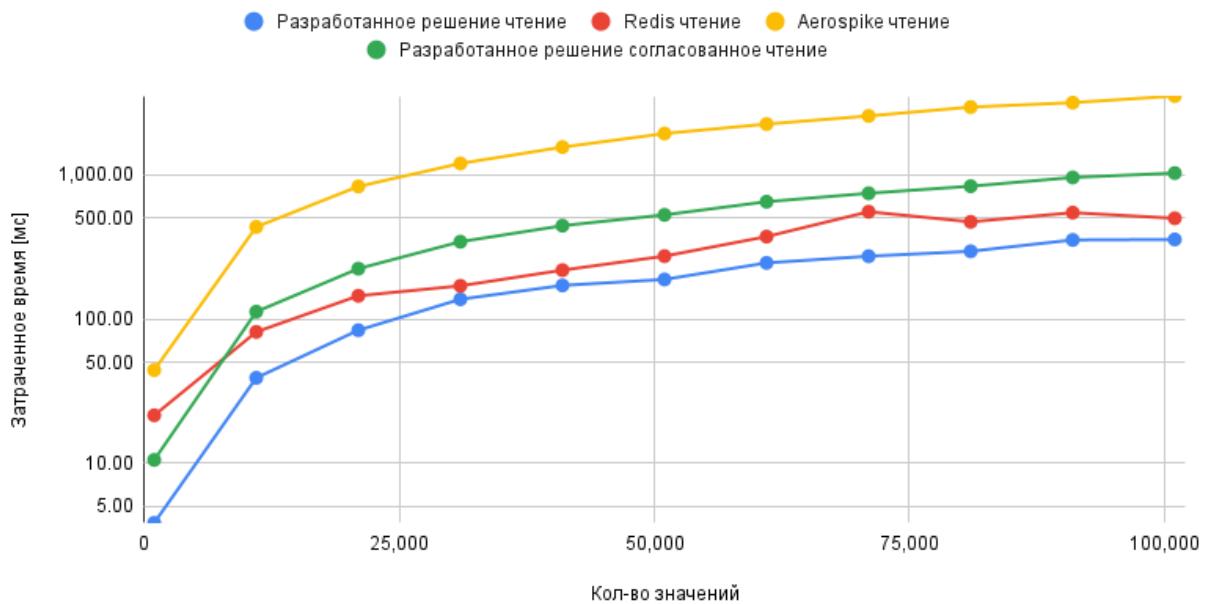


Рисунок 25 - Сравнение производительности на чтение

В дополнение была составлена таблица с характеристиками производительности как разработанного хранилища, так и сравниваемых

аналогов. Таблица содержит максимальные и минимальные показатели производительности для каждого из решений, так как производительность варьируется в зависимости от сценария использования.

Таблица 6 - Итоговое сравнение характеристик производительности

Итого производительность	Среднее кол-во команд [команд/мс]					
	Разработанное решение		Redis		Aerospike	
Тип операции	Минимальное	Максимальное	Минимальное	Максимальное	Минимальное	Максимальное
Согласованное чтение	0.38	166.44	-	-	0.64	29.41
Согласованная запись	0.38	166.44	-	-	0.52	1.32
Несогласованное чтение	250.00	333.33	0.50	200.00	-	-
Несогласованная запись	90.91	333.33	0.53	200.00	-	-

Из полученных результатов сравнения характеристик производительности, можно сделать следующие выводы:

- Ввиду предоставления различных гарантий пользователю при выполнении различных операций, Aerospike жертвует достаточно большой частью производительности. Возможно требуется дополнительная настройка конфигурации кластера для увеличения производительности.
- Redis показывает стабильно хорошие результаты, однако не предоставляет возможности для обеспечения строгой согласованности “из коробки”.
- Скорость распространения информации используя реализованный консенсус RAFT очень сильно варьируется в зависимости от настройки максимального размера пересылаемого пакета команд.
- Скорость операций, при отсутствии необходимой строгой согласованности, является наибольшей у разработанного хранилища. Именно такой сценарий использования и рассматривается как целевой.

ЗАКЛЮЧЕНИЕ

Целью данной работы было повышение производительности систем с высокочастотной записью и чтением из базы данных, путем разработки распределенного встроенного хранилища, в качестве замены текущим аналогам.

В ходе процесса разработки были исследованы существующие реализации распределенных хранилищ, с их достоинствами и недостатками. Особое внимание было уделено их внутреннему устройству, алгоритмам репликации данных и поддержания консенсуса в распределенной сети. Были рассмотрены различные варианты локального хранилища, на базе которого будет строиться разрабатываемое хранилище и выбрано наиболее подходящие под критерии разработки.

Основной особенностью разработанного решения было обеспечение минимальной задержки записи и чтения значений при отсутствии необходимой гарантии на строгую согласованность значений, но при этом наличие возможности требовать такую согласованность. Также хранилище предоставляет идиоматичное Scala API для интеграции в существующие системы, с абстракцией от системы контроля эффектов.

Проведено тестирование производительности разработанного хранилища и сравнения его характеристики с аналогами. Результаты тестирования указывают на превосходящую производительность разработанного решения, при использовании системы в целевом режиме работы, что указывает на успешное выполнение поставленной цели работы.

На данный момент разработанное решение интегрируется в коммерческий проект, с целью повышения производительности системы и понижения аппаратных требований к работе сервиса.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. “Эволюция баз данных” [Электронный ресурс]. URL:
<https://pgconf.ru/201811/237652> (дата обращения 24.05.2022)
2. “Redis System Properties” [Электронный ресурс]. URL:
<https://db-engines.com/en/system/Redis> (дата обращения 24.05.2022)
3. “etcd System Properties” [Электронный ресурс]. URL:
<https://db-engines.com/en/system/etcd> (дата обращения 24.05.2022)
4. Ongaro D., Ousterhout J. In Search of an Understandable Consensus Algorithm, Stanford University, 2014 – 18 с
5. “Aerospike System Properties” [Электронный ресурс]. URL:
<https://db-engines.com/en/system/Aerospike> (дата обращения 24.05.2022)
6. “RocksDB System Properties” [Электронный ресурс]. URL:
<https://db-engines.com/en/system/RocksDB> (дата обращения 24.05.2022)
7. “RocksDB Bloom Filter” [Электронный ресурс]. URL:
<https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter> (дата обращения 24.05.2022)
8. “3D XPoint” [Электронный ресурс]. URL:
https://en.wikipedia.org/wiki/3D_XPoint (дата обращения 24.05.2022)
9. “Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB” [Электронный ресурс]. URL:
<https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/> (дата обращения 24.05.2022)
10. “Распределенные данные. Алгоритмы работы современных систем хранения информации” / Петров Алекс – Питер, 2021. – 336 с
11. Lamport L. Paxos Made Simple, 2001 – 11 с
12. Chandra T., Griesemer R., Redstone J. Paxos Made Live - An Engineering Perspective, 2007 – 16 с

13. Lamport L. Fast Paxos, Redmond 2005 – 43 с
14. Moraru I., Andersen D. G., Kaminsky M. There Is More Consensus in Egalitarian Parliaments, Carnegie Mellon University 2013 - 15 с
15. Howard H., Malkhi D., Spiegelman A. Flexible Paxos: Quorum intersection revisited, 2016 – 20 с
16. Howard H., Mortier R. A Generalised Solution to Distributed Consensus, Cambridge 2019 – 20 с
17. “Multi-Raft” [Электронный ресурс]. URL:
<https://tikv.org/deep-dive/scalability/multi-raft/> (дата обращения 24.05.2022)
18. “Production use of Paxos” [Электронный ресурс]. URL:
[https://en.wikipedia.org/wiki/Paxos_\(computer_science\)#Production_use_of_Paxos](https://en.wikipedia.org/wiki/Paxos_(computer_science)#Production_use_of_Paxos) (дата обращения 24.05.2022)
19. “Raft consensus algorithm” [Электронный ресурс]. URL:
<https://raft.github.io/> (дата обращения 24.05.2022)
20. “Tiobe index for May 2022” [Электронный ресурс]. URL:
<https://www.tiobe.com/tiobe-index/> (дата обращения 24.05.2022)
21. “Rocksplicator” [Электронный ресурс]. URL:
<https://github.com/pinterest/rocksplicator> (дата обращения 24.05.2022)
22. “Side effect (computer science)” [Электронный ресурс]. URL:
[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) (дата обращения 24.05.2022)
23. “Controlling side effects in function” [Электронный ресурс]. URL:
<https://stackoverflow.com/questions/65167380/controlling-side-effects-in-function> (дата обращения 24.05.2022)
24. “Cats Effect” [Электронный ресурс]. URL:
<https://typelevel.org/cats-effect/> (дата обращения 24.05.2022)

25. “FS2 Guide” [Электронный ресурс]. URL: <https://fs2.io/#/guide> (дата обращения 24.05.2022)
26. “Protocol Buffers Developer Guide” [Электронный ресурс]. URL:-
<https://web.archive.org/web/20140911110359/https://developers.google.com/protocol-buffers/docs/overview> (дата обращения 24.05.2022)
- 27.“RocksJava Basics, Memory management” [Электронный ресурс]. URL:
<https://github.com/facebook/rocksdb/wiki/RocksJava-Basics#memory-management> (дата обращения 24.05.2022)
- 28.“Kind (type theory)” [Электронный ресурс]. URL:
[https://en.wikipedia.org/wiki/Kind_\(type_theory\)](https://en.wikipedia.org/wiki/Kind_(type_theory)) (дата обращения 24.05.2022)
- 29.“Raft4s” [Электронный ресурс]. URL:
<https://github.com/alirezameskin/raft4s> (дата обращения 24.05.2022)
- 30.“Cats-Effect, Migration Guide” [Электронный ресурс]. URL:
<https://typelevel.org/cats-effect/docs/migration-guide> (дата обращения 24.05.2022)

ПРИЛОЖЕНИЕ А (Модуль работы с RocksDB)

Структура модуля работы с RocksDB:

```
└── RocksEngine.scala
└── codec
    └── Decoder.scala
        └── Encoder.scala
└── utils
    └── NativeResource.scala
```

RocksEngine.scala

```
final class RocksEngine[F[_]: Async](val dbRef: Ref[F, (RocksDB, F[Unit])],
                                      stopWorld: Ref[F, Deferred[F, Unit]])
) {
    private def withPermission[T](f: RocksDB => F[T]): F[T] =
        dbRef.get.flatMap(rdb => stopWorld.get.flatMap(_.get) >> f(rdb._1))
    private val revokePermission: Resource[F, RocksDB] =
        Resource
            .make(Deferred[F, Unit].flatMap(stopWorld.set).void)(_ =>
                stopWorld.get.map(_.complete(())))
            .evalMap(_ => dbRef.get.map(_.1))

    def get[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]] = for {
        encodedK <- Encoder[F, K].encode(key)
        data      <- withPermission(db => Sync[F].delay(db.get(encodedK)))
        decodedV <- Decoder[F, V].decode(data)
    } yield decodedV

    def put[K: Encoder[F, *], V: Encoder[F, *]](key: K, value: V): F[Unit] =
        (Encoder[F, K].encode(key) product Encoder[F, V].encode(value)).flatMap {
            case (encodedK, encodedV) => withPermission(db => Sync[F].delay(db.put(encodedK,
                encodedV)))
        }

    def delete[K: Encoder[F, *]](key: K): F[Unit] =
        Encoder[F, K]
            .encode(key)
            .flatMap(encoded => withPermission(db => Sync[F].delay(db.delete(encoded))))

    def isEmpty[K: Encoder[F, *]](key: K): F[Boolean] =
        Encoder[F, K]
            .encode(key)
            .flatMap(encoded =>
                withPermission(db => Sync[F].delay(!db.keyMayExist(encoded, new
                    Holder[Array[Byte]]()))))

    def poke[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]] =
        for {
            encodedK <- Encoder[F, K].encode(key)
            holder   = new Holder(Array.emptyByteArray)
            canExist <- withPermission(db => Sync[F].delay(db.keyMayExist(encodedK, holder)))
            decodedV <- if (canExist) Decoder[F, V].decode(holder.getValue) else None.pure[F]
        } yield decodedV

    def batchGet[K: Encoder[F, *], V: Decoder[F, *]](keys: List[K]): F[List[Option[V]]] = for {
        encodedKeys <- keys.traverse(Encoder[F, K].encode)
        data        <- withPermission(db =>
            Sync[F].blocking(db.multiGetAsList(encodedKeys.asJava)))
        dataS       <- Sync[F].delay(data.asscala.toList)
        decodedValues <- dataS.traverse(Decoder[F, V].decode)
    } yield decodedValues

    def batchPut[K: Encoder[F, *], V: Encoder[F, *]](entries: Seq[(K, V)]): F[Unit] =
        entries
            .traverse { case (key, value) =>
                Encoder[F, K].encode(key) product Encoder[F, V].encode(value)
            }
}
```

```

    }
    .flatMap(data =>
      (nativeResource[Sync[F]].delay(new WriteOptions())) product
      nativeResource[Sync[F]].delay(new WriteBatch()))
      .evalMap(batch =>
        Sync[F]
        .delay(data.foreach { case (key, value) =>
          batch.put(key, value)
        })
        .as(batch)
      )
      .use { case (opts, batch) =>
        withPermission(db => Sync[F].blocking(db.write(opts, batch)))
      }
    )
  )

  def flush(awaited: Boolean): F[Unit] =
    nativeResource[Sync[F]].delay(new FlushOptions().setWaitForFlush(awaited))
    .use(options => withPermission(db => Sync[F].blocking(db.flush(options))))

  val backupResource: Resource[F, BackupEngine] = for {
    env      <- nativeResource[Sync[F]].delay(Env.getDefault)
    opt      <- nativeResource getProperty("path").map(new BackupEngineOptions(_))
    engine   <- nativeResource[Sync[F]].delay(BackupEngine.open(env, opt))
  } yield engine

  def backup(flushBeforeBackup: Boolean): F[Unit] =
    (for {
      db     <- revokePermission
      be     <- backupResource
      <- Resource.eval(Sync[F].blocking(be.createNewBackup(db, flushBeforeBackup)))
    } yield ()).use_
  }

  def getBackups: F[List[BackupInfo]] =
    backupResource.use(be => Sync[F].blocking(be.getBackupInfo.asScala.toList))

  def restoreDB(backupId: Int): F[Unit] =
    revokePermission.use(db =>
      Sync[F]
      .blocking(db.getLiveFiles().files.asScala)
      .map(_.head)
      .map(path => path.take(path.lastIndexOf('/')))
      .flatMap(dbDir =>
        backupResource.use(be =>
          Sync[F].blocking {
            be.restoreDbFromBackup(backupId, dbDir, dbDir, new RestoreOptions(true))
            be.garbageCollect()
          }
        )
      )
    )
  }

  def getProperty(propName: String): F[String] =
    withPermission(db => Sync[F].delay(db.getProperty(propName)))

  def compact: F[Unit] =
    withPermission(db => Sync[F].blocking(db.compactRange()))

  def checkpoint(path: String): Resource[F, Unit] =
    revokePermission.flatMap(db =>
      nativeResource[Sync[F]].delay(Checkpoint.create(db)).evalMap(cp =>
        Sync[F].blocking(cp.createCheckpoint(path))
      )
    )
  }

  def dbGenStop: Resource[F, Unit] = revokePermission.flatMap(db =>
    Resource.make(Sync[F].blocking {
      db.pauseBackgroundWork()
      db.disableFileDeletions()
    })(_ =>
      Sync[F].blocking {
        db.continueBackgroundWork()
        db.enableFileDeletions(true)
      }
    )
  )
}

```

```

        )
    )

def hotSwap(newDB: Resource[F, RocksDB]): F[Unit] =
  newDB.allocated.flatMap(dbRef.getAndSet(_).flatMap(_._.2))

def close: F[Unit] =
  dbRef.get.flatMap(_._.2)
}

object RocksEngine {
  def apply[F[_]: Async](underlying: Resource[F, RocksDB]): F[RocksEngine[F]] =
    for {
      completedDeferred <- Deferred[F, Unit]
      - completedDeferred.complete(())
      ref           <- Ref.of(completedDeferred)
      dbRef         <- Ref.ofEffect(underlying.allocated)
    } yield new RocksEngine(dbRef, ref)

  def openDB[F[_]: Sync](path: String,
                           options: Options,
                           ttl: Boolean = false)
    : Resource[F, RocksDB] =
    nativeResource(
      Sync[F].blocking(if (ttl) TtlDB.open(options, path) else RocksDB.open(options, path))
    )

  def initDB[F[_]: Sync](path: String,
                           options: Options,
                           ttl: Boolean = false)
    : Resource[F, RocksDB] =
    Resource.eval(Sync[F].blocking(RocksDB.loadLibrary())) >> openDB(path, options, ttl)
}

```

Decoder.scala

```

trait Decoder[F[_], A] {
  def decode(bytes: Array[Byte]): F[Option[A]]
}

object Decoder {
  def apply[F[_], A](implicit ev: Decoder[F, A]): Decoder[F, A] = ev

  implicit def functor[F[_]: Functor]: Functor[Decoder[F, *]] = new Functor[Decoder[F, *]] {
    override def map[A, B](fa: Decoder[F, A])(f: A => B): Decoder[F, B] =
      (bytes: Array[Byte]) => fa.decode(bytes).map(_.map(f))
  }

  implicit def stringDec[F[_]: Sync]: Decoder[F, String] =
    (bytes: Array[Byte]) => Sync[F].delay(Option(bytes).map(new String(_, StandardCharsets.UTF_8)))
}

implicit def scodecDec[F[_]: Sync, N: Codec]: Decoder[F, N] = (data: Array[Byte]) =>
  Sync[F].delay(Codec[N].decode(BitVector(data)).map(_.value).toOption)
}

```

Encoder.scala

```

trait Encoder[F[_], A] {
  def encode(data: A): F[Array[Byte]]
}

object Encoder {
  def apply[F[_], A](implicit ev: Encoder[F, A]): Encoder[F, A] = ev

  implicit def contravariant[F[_]]: Contravariant[Encoder[F, *]] =
    new Contravariant[Encoder[F, *]] {
      override def contramap[A, B](fa: Encoder[F, A])(f: B => A): Encoder[F, B] = (data: B) =>
        fa.encode(f(data))
    }
}

```

```

implicit def stringEnc[F[_]: Sync]: Encoder[F, String] =
  (data: String) => Sync[F].delay(data.getBytes(StandardCharsets.UTF_8))

implicit def scodecEnc[F[_]: Sync, N: Codec]: Encoder[F, N] =
  (data: N) =>
    Sync[F].defer(
      Codec[N]
        .encode(data)
        .fold(
          err => Sync[F].raiseError(new RuntimeException(err.messageWithContext)),
          r => Sync[F].pure(r.toArray)
        )
    )
}

```

NativeResource.scala

```

import cats.effect.{Resource, Sync}
import org.rocksdb.AbstractNativeReference

object NativeResource {
  def nativeResource[F[_]: Sync, A <: AbstractNativeReference](fa: F[A]): Resource[F, A] =
    Resource.make(fa)(a => Sync[F].delay(a.close()))
}

```

ПРИЛОЖЕНИЕ Б (Модуль алгоритма консенсуса RAFT)

Структура модуля алгоритма консенсуса RAFT:

```
├── Cluster.scala
├── Raft.scala
└── impl
    ├── RaftCluster.scala
    ├── RaftImpl.scala
    └── SingleRaftCluster.scala
├── model
    ├── Command.scala
    ├── Configuration.scala
    ├── LogCompactionPolicy.scala
    ├── LogEntry.scala
    ├── Node.scala
    ├── PersistedState.scala
    └── Snapshot.scala
└── node
    ├── CandidateNode.scala
    ├── FollowerNode.scala
    ├── LeaderNode.scala
    └── NodeState.scala
└── protocol
    ├── Action.scala
    ├── AppendEntries.scala
    ├── AppendEntriesResponse.scala
    ├── ClusterConfiguration.scala
    ├── InstallSnapshot.scala
    ├── LogState.scala
    ├── VoteRequest.scala
    └── VoteResponse.scala
└── rpc
    ├── RpcClient.scala
    ├── RpcClientBuilder.scala
    ├── RpcServer.scala
    └── RpcServerBuilder.scala
    └── grpc
        ├── GrpcClientBuilder.scala
        ├── GrpcServerBuilder.scala
        └── impl
            ├── GrpcRaftClient.scala
            └── GrpcRaftService.scala
        └── serializer
            ├── JavaProtoSerializer.scala
            └── ProtoSerializer.scala
        └── transform
            └── instances.scala
└── service
    ├── ClientManager.scala
    ├── ClusterConfigStorage.scala
    ├── ErrorLogging.scala
    ├── LeaderAnnouncer.scala
    ├── Log.scala
    ├── LogPropagator.scala
    └── impl
        ├── ClientManagerImpl.scala
        └── ClusterConfigStorageImpl.scala
```

```

    └── LeaderAnnouncerImpl.scala
    └── LogImpl.scala
    └── LogPropagatorImpl.scala
└── storage
    ├── LogStorage.scala
    ├── SnapshotStorage.scala
    ├── StateMachine.scala
    ├── StateStorage.scala
    └── Storage.scala
    └── impl
        └── memory
            ├── MemoryLogStorage.scala
            ├── MemorySnapshotStorage.scala
            ├── MemoryStateStorage.scala
            └── MemoryStorage.scala
    └── serialisation
        ├── JavaSerializer.scala
        ├── PersistedStateSerializer.scala
        └── Serializer.scala
    └── default
        └── package.scala
└── util
    ├── Logger.scala
    └── console
        └── ConsoleLogger.scala

```

Cluster.scala

```

trait Cluster[F[_], +SM[X[_]] <: StateMachine[X]] {
  val rpc: RpcServer[F]
  val raft: Raft[F, SM]

  def start: F[Node]
  def stop: F[Unit]
  def join(node: Node): F[Node]
  def leave: F[Unit]
  def leader: F[Node]
  def execute[T](command: Command[T]): F[T]
}

```

Raft.scala

```

trait Raft[F[_], +SM[X[_]] <: StateMachine[X]] extends ErrorLogging[F] {
  implicit val logger: Logger[F]
  implicit val ME: MonadError[F, Throwable]

  val nodeId: Node
  val raftId: Int
  val leadersLimit: Option[Semaphore[F]]

  val config: Configuration
  val log: Log[F, SM]
  val storage: Storage[F]

  val leaderAnnouncer: LeaderAnnouncer[F]
  val clientProvider: ClientManager[F]
  val membershipManager: ClusterConfigStorage[F]
}

```

```

val logReplicator: LogPropagator[F]

def setRunning(isRunning: Boolean): F[Unit]

def getRunning: F[Boolean]

def getCurrentState: F[NodeState]

def setCurrentState(state: NodeState): F[Unit]

def background[A](fa: => F[A]): F[Unit]

def updateLastHeartbeat(): F[Unit]

def electionTimeoutElapsed: F[Boolean]

def delayElection(): F[Unit]

def schedule(delay: FiniteDuration)(fa: => F[Unit]): F[Unit]

def initialize: F[Unit] = log.initialize

def start: F[Node]

def join(node: Node): F[Node]

def stop: F[Unit]

def leave: F[Unit]

def listen: F[Node]

def onVote(msg: VoteRequest): F[VoteResponse]

def onVoteResp(msg: VoteResponse): F[Unit]

def onAppendEntries(msg: AppendEntries): F[AppendEntriesResponse]

def onAppendResponse(msg: AppendEntriesResponse): F[Unit]

def onSnapshot(msg: InstallSnapshot): F[AppendEntriesResponse]

def addMember(member: Node): F[Unit]

def removeMember(member: Node): F[Unit]

def onCommand[T](command: Command[T]): F[T]
}

```

RaftCluster.scala

```

object RaftCluster {
  def resource[F[_]: Async: RpcServerBuilder: RpcClientBuilder: Logger, SM[X[_]] <:
  StateMachine[X]](raftId: Int,
    config: Configuration,
    storage: Storage[F],
    stateMachine: SM[F]): Resource[F, Cluster[F, SM]] =
  resource(
    raftId,
    config,
    storage,
    stateMachine,
    if (config.logCompactionThreshold <= 0)
      LogCompactionPolicy.noCompaction
    else
      LogCompactionPolicy.fixedSize(config.logCompactionThreshold)
  )
}

```

```

def resource[F[_]: Async: RpcServerBuilder: RpcClientBuilder: Logger, SM[X[_]] <:
StateMachine[X]
  ]](raftId: Int,
  config: Configuration,
  storage: Storage[F],
  stateMachine: SM[F],
  compactionPolicy: LogCompactionPolicy[F]
): Resource[F, Cluster[F, SM]] =
  for {
    raft                                <- Resource.eval(RaftImpl.build(raftId, config,
storage, stateMachine, compactionPolicy))
    initMap: Map[Int, Raft[F, StateMachine]] = Map(raftId -> raft)
    rafts                               <- Resource.eval(Ref.of(initMap))
    server                               <-
  Resource.eval(RpcServerBuilder[F].build(config.local, rafts))
    acquire                                = Async[F].delay(new SingleRaftCluster[F,
SM](server, raft))
    cluster                               <- Resource.make(acquire) (_._stop)
  } yield cluster
}

```

RaftImpl.scala

```

class RaftImpl[F[_]: Async, +SM[X[_]] <: StateMachine[X]](
  val raftId: Int,
  val leadersLimit: Option[Semaphore[F]],
  val config: Configuration,
  val membershipManager: ClusterConfigStorage[F],
  val clientProvider: ClientManager[F],
  val leaderAnnouncer: LeaderAnnouncer[F],
  val logReplicator: LogPropagator[F],
  val log: Log[F, SM],
  val storage: Storage[F],
  stateRef: Ref[F, NodeState],
  lastHeartbeatRef: Ref[F, Long],
  isRunning: Ref[F, Boolean]
)(implicit val ME: MonadError[F, Throwable], val logger: Logger[F])
extends Raft[F, SM] {

  override val nodeId: Node = config.local

  override def setRunning(running: Boolean): F[Unit] =
    isRunning.set(running)

  override def getRunning: F[Boolean] =
    isRunning.get

  override def getCurrentState: F[NodeState] =
    stateRef.get

  override def setCurrentState(state: NodeState): F[Unit] =
    stateRef.set(state)

  override def background[A](fa: => F[A]): F[Unit] =
    Concurrent[F].start(fa).void

  override def updateLastHeartbeat(): F[Unit] =
    logger.trace(s"Update Last heartbeat time") >>
    Clock[F].monotonic.flatMap(t => lastHeartbeatRef.set(t.toMillis))

  override def electionTimeoutElapsed: F[Boolean] =
    for {
      node <- getCurrentState
      lh   <- lastHeartbeatRef.get
      now  <- Clock[F].monotonic
    } yield node.isInstanceOf[LeaderNode] || (now.toMillis - lh <
config.heartbeatTimeoutMillis)

  override def delayElection(): F[Unit] =
    for {
      millis <- random(config.electionMinDelayMillis, config.electionMaxDelayMillis)
    }
}

```

```

delay    = FiniteDuration(millis, TimeUnit.MILLISECONDS)
-      <- logger.trace(s"Delay to start the election $delay")
-      <- Temporal[F].sleep(delay)
} yield ()

override def schedule(delay: FiniteDuration)(fa: => F[Unit]): F[Unit] =
  Monad[F]
    .foreverM(Temporal[F].sleep(delay) >> fa)
    .whileM_(isRunning.get)

private def random(min: Int, max: Int): F[Int] =
  Async[F].delay(min + scala.util.Random.nextInt(max - min))

def start: F[Node] =
  errorLogging("Starting Cluster") {
    for {
      - <- setRunning(true)
      - <- logger.info("Cluster is starting")
      node   <- getCurrentState
      -      <- if (node.leader.isDefined) Monad[F].unit else runElection()
      -      <- scheduleElection()
      -      <- scheduleHeartbeat()
      -      <- logger.trace("Waiting for the leader to be elected.")
      leader <- leaderAnnouncer.listen()
      -      <- logger.info(s"A Leader is elected. Leader: '$leader'")
    } yield leader
  }

def join(node: Node): F[Node] =
  errorLogging("Joining to a cluster") {
    for {
      -      <- setRunning(true)
      -      <- logger.info("Cluster is joining")
      res   <- clientProvider.join(node, raftId, nodeId)
      -      <- logger.trace(s"Cluster is joined to $node $res")
      node   <- getCurrentState
      -      <- if (node.leader.isDefined) Monad[F].unit else runElection()
      -      <- scheduleElection()
      -      <- scheduleHeartbeat()
      -      <- logger.trace("Waiting for the leader to be elected.")
      leader <- leaderAnnouncer.listen()
      -      <- logger.info(s"A Leader is elected. Leader: '$leader'")
    } yield leader
  }

def stop: F[Unit] =
  errorLogging("Stopping a Cluster") {
    for {
      -      <- logger.info("Stopping the cluster")
      -      <- setRunning(false)
      -      <- clientProvider.closeConnections()
      -      <- logger.info("Cluster stopped")
    } yield ()
  }

def leave: F[Unit] =
  errorLogging("Leaving a cluster") {
    for {
      -      <- logger.info(s"Node $nodeId is leaving the cluster")
      -      <- removeMember(nodeId)
      -      <- logger.info(s"Node $nodeId left the cluster.")
      -      <- setRunning(false)
    } yield ()
  }

def listen: F[Node] =
  errorLogging("Waiting for the Leader to be elected") {
    leaderAnnouncer.listen()
  }

def onVote(msg: VoteRequest): F[VoteResponse] =
  errorLogging("Receiving VoteRequest") {
    for {
      -      <- logger.trace(

```

```

        s"A Vote request received from ${msg.nodeId}, Term: ${msg.lastLogTerm}, $msg"
    )
logState <- log.state
config   <- membershipManager.getClusterConfiguration
result   <- modifyState(_.onVoteRequest(logState, config, msg))

(response, actions) = result

<- runActions(actions)
<- logger.trace(s"Vote response to the request $response")
<- if (response.voteGranted) updateLastHeartbeat() else Monad[F].unit
} yield response
}

def onVoteResp(msg: VoteResponse): F[Unit] =
errorLogging("Receiving VoteResponse") {
for {
    <- logger.trace(
        s"A Vote response received from ${msg.nodeId}, Granted: ${msg.voteGranted}, $msg"
    )
logState <- log.state
config   <- membershipManager.getClusterConfiguration
actions  <- modifyState(_.onVoteResponse(logState, config, msg))
    <- runActions(actions)
} yield ()
}

def onAppendEntries(msg: AppendEntries): F[AppendEntriesResponse] =
errorLogging(
    s"Receiving an AppendEntries Term: ${msg.term} PreviousLogIndex:${msg.prevLogIndex}"
) {
for {
    <-
    <- logger.trace(
        s"AppendEntries request received from ${msg.leaderId}, contains ${msg.entries.size} entries, $msg"
    )
logState      <- log.state
localPreEntry <- log.get(msg.prevLogIndex)
config        <- membershipManager.getClusterConfiguration
result        <- modifyState(_.onEntries(logState, config, msg, localPreEntry))
    <- updateLastHeartbeat()

(response, actions) = result
    <- runActions(actions)

appended <-
if (response.success) {
    logger.trace(s"Appending entries ${msg.entries}...")
    log.appendEntries(msg.entries, msg.prevLogIndex, msg.leaderCommit)
} else
    Monad[F].pure(false)
<- if (appended) storeState() else Monad[F].unit
    <- logger.trace(s"Did append? $appended (${response.success})")
    <- logger.trace(s"Append entries response $response")
    <- logger.trace(s"Actions $actions")
} yield response
}

def onAppendResponse(msg: AppendEntriesResponse): F[Unit] =
errorLogging("Receiving AppendEntriesResponse") {
for {
    <- logger.trace(s"A AppendEntriesResponse received from ${msg.nodeId}. $msg")
logState <- log.state
config   <- membershipManager.getClusterConfiguration
actions  <- modifyState(_.onEntriesResp(logState, config, msg))
    <- logger.trace(s"Actions $actions")
    <- runActions(actions)
} yield ()
}

def onSnapshot(msg: InstallSnapshot): F[AppendEntriesResponse] =
errorLogging("Receiving InstallSnapshot") {

```

```

    for {
        <- log.installSnapshot(msg.snapshot, msg.lastEntry)
        logState <- log.state
        config   <- membershipManager.getClusterConfiguration
        response <- modifyState(_.onSnapshotInstalled(logState, config))
    } yield response
}

def addMember(member: Node): F[Unit] =
for {
    config <- membershipManager.getClusterConfiguration
    <- addMember(config, member)
} yield ()

private def addMember(config: ClusterConfiguration, member: Node): F[Unit] =
if (config.members.contains(member)) {
    Applicative[F].unit
} else {
    val oldMembers = config.members
    val newMembers = oldMembers + member
    val newConfig = JointClusterConfiguration(oldMembers, newMembers)

    for {
        <- membershipManager.setClusterConfiguration(newConfig)
        <- logger.trace(s"Committing a joint configuration $newConfig")
        <- onCommand[Unit](JointConfigurationCommand(oldMembers, newMembers))
        <- logger.trace("Joint configuration is committed")
        <- onCommand[Unit](NewConfigurationCommand(newMembers))
        <- logger.trace("New configuration is committed")
    } yield ()
}

def removeMember(member: Node): F[Unit] =
for {
    config <- membershipManager.getClusterConfiguration
    <- removeMember(config, member)
} yield ()

private def removeMember(config: ClusterConfiguration, member: Node): F[Unit] =
if (!config.members.contains(member)) {
    Applicative[F].unit
} else {

    val oldMembers = config.members.toSet
    val newMembers = oldMembers - member
    val newConfig = JointClusterConfiguration(oldMembers, newMembers)

    for {
        <- membershipManager.setClusterConfiguration(newConfig)
        <- logger.trace(s"Committing a joint configuration $newConfig")
        <- onCommand[Unit](model.JointConfigurationCommand(oldMembers, newMembers))
        <- logger.trace("Joint configuration is committed")
        <- onCommand[Unit](model.NewConfigurationCommand(newMembers))
        <- logger.trace("New configuration is committed")
    } yield ()
}

def onCommand[T](command: Command[T]): F[T] =
errorLogging("Receiving Command") {
    command match {
        case command: ReadCommand[T] =>
            for {
                <- logger.trace(s"A read comment received $command")
                state_ <- getCurrentState
                result <- onReadCommand(state_, command)
            } yield result

        case command: WriteCommand[T] =>
            for {
                deferred <- Deferred[F, T]
                state_   <- getCurrentState
                <- logger.trace(s"A write command received $command (curState: $state_)")
                config   <- membershipManager.getClusterConfiguration
                actions  <- onWriteCommand(state_, config, command, deferred)
            } yield result
    }
}

```

```

        <- runActions(actions)
    _tryResult <- deferred.tryGet
    result <- tryResult match {
        case Some(value) => value.pure[F]
        case None         => Concurrent[F].cede >> deferred.get
    }
} yield result
}

private def onReadCommand[T](node: NodeState, command: ReadCommand[T]): F[T] =
node match {
case _: LeaderNode =>
for {
    _ <- logger.trace("Current node is the leader, it is running the read command")
    res <- log.applyReadCommand(command)
} yield res

case _: FollowerNode if config.followerAcceptRead =>
for {
    _ <- logger.trace("Current node is a follower, it is running the read command")
    res <- log.applyReadCommand(command)
} yield res

case _ =>
for {
    _ <- logger.trace("Read command has to be ran on the leader node")
    leader <- leaderAnnouncer.listen()
    _ <- logger.trace(s"The current leader is $leader")
    response <- clientProvider.sendCommand(leader, raftId, command)
    _ <- logger.trace("Response for the read command received from the leader")
} yield response
}

private def onWriteCommand[T](node: NodeState,
                           cluster: ClusterConfiguration,
                           command: WriteCommand[T],
                           deferred: Deferred[F, T]
): F[List[Action]] =
node match {
case LeaderNode(`raftId`, _, term, _, _) if cluster.members.size == 1 =>
for {
    entry <- log.append(term, command, deferred)
    _ <- logger.trace(s"Appending the command to the log - ${cluster.members}")
    committed <- log.commitLogs(Map(nodeId -> entry.index))
    _ <- if (committed) storeState() else Monad[F].unit
} yield List.empty
case LeaderNode(`raftId`, _, term, _, _) =>
for {
    _ <- logger.trace(s"Appending the command to the log ${cluster.members}")
    _ <- log.append(term, command, deferred)
    _ <- log.getCommitIndex.flatMap(i => logger.trace(s"After appending CI is: $i"))
} yield node.onReplicateLog(cluster)
case _ =>
for {
    _ <- logger.trace("Write commands should be forwarded to the leader node")
    leader <- leaderAnnouncer.listen()
    _ <- logger.trace(s"The current leader is $leader.")
    response <- clientProvider.sendCommand(leader, raftId, command)
    _ <- logger.trace("Response for the write command received from the leader")
    _ <- deferred.complete(response)
} yield List.empty
}

private def runActions(actions: List[Action]): F[Unit] =
actions.traverse(action => runAction(action).attempt) >> Monad[F].unit

private def runAction(action: Action): F[Unit] =
action match {
case RequestForVote(peerId, request) =>
background {
for {
    _ <- logger.trace(s"Sending a vote request to $peerId. Request: $request")
}
}
}

```

```

        response <- clientProvider.sendVote(peerId, request)
                    <- onVoteResp(response)
    } yield response
}

case ReplicateLog(peerId, term, nextIndex) =>
background {
    errorLogging(s"Replicating logs to $peerId, Term: $term, NextIndex: $nextIndex") {
        for {
            response <- logReplicator.propagateLogs(raftId, peerId, term, nextIndex)
                        <- onAppendResponse(response)
        } yield ()
    }
}

case StoreState =>
storeState()

case CommitLogs(matchIndex) =>
for {
    committed <- log.commitLogs(matchIndex)
                <- if (committed) storeState() else Monad[F].unit
} yield ()

case AnnounceLeader(leaderId, true) =>
leaderAnnouncer.reset() >> leaderAnnouncer.announce(leaderId)

case AnnounceLeader(leaderId, false) =>
logger.trace("Announcing a new leader without resetting") >>
leaderAnnouncer.announce(leaderId)

case ResetLeaderAnnouncer =>
leaderAnnouncer.reset()

}

private def storeState(): F[Unit] =
for {
    <- logger.trace("Storing the new state in the storage")
logState <- log.state
node      <- getCurrentState
 $\_ \leftarrow$  storage.stateStorage.persistState(
    node.toPersistedState.copy(appliedIndex = logState.lastAppliedIndex)
)
} yield ()

private def runElection(): F[Unit] =
for {
    <- delayElection()
logState <- log.state
config   <- membershipManager.getClusterConfiguration
allowed   <- leadersLimit.fold(true.pure)(_.tryAcquire)
actions   <- modifyState(_.onElectionTimer(logState, config, allowed))
                <- runActions(actions)
} yield ()

private def scheduleHeartbeat(): F[Unit] =
background {
    schedule(FiniteDuration(config.heartbeatIntervalMillis, TimeUnit.MILLISECONDS)) {
        for {
            <- Logger[F].trace("Sending heartbeat")
            node      <- getCurrentState
            config   <- membershipManager.getClusterConfiguration
            actions   = if (node.isInstanceOf[LeaderNode]) node.onReplicateLog(config) else
List.empty
                <- runActions(actions)
        } yield ()
    }
}

private def scheduleElection(): F[Unit] =
background {
    schedule(FiniteDuration(config.heartbeatTimeoutMillis, TimeUnit.MILLISECONDS)) {
        for {

```

```

        alive <- electionTimeoutElapsed
        <- if (alive) Monad[F].unit else runElection()
    } yield ()
}

private def modifyState[B] (f: NodeState => (NodeState, B)): F[B] =
  for {
    prevState      <- getCurrentState
    (newState, result) = f(prevState)
    <- setCurrentState(newState)
  } <- (prevState, newState) match {
    case (_: LeaderNode, _: FollowerNode) => leadersLimit.fold(() .pure) (_ .release)
    case (_: CandidateNode, _: LeaderNode) => leadersLimit.fold(() .pure) (_ .acquire)
    case _                                     => () .pure
  }
} yield result
}

object RaftImpl {

  def build[F[_]: Async: RpcClientBuilder: Logger, SM[X[_]] <: StateMachine[X]] (
    raftId: Int,
    config: Configuration,
    storage: Storage[F],
    stateMachine: SM[F],
    compactionPolicy: LogCompactionPolicy[F]
  ): F[Raft[F, SM]] =
    for {
      persistedState <- storage.stateStorage.retrieveState()
      nodeState =
        persistedState
          .map(_.toNodeState(raftId, config.local))
          .getOrElse(FollowerNode(raftId, config.local, OL))
      appliedIndex     = persistedState.map(_.appliedIndex).getOrElse(OL)
      clientProvider <- ClientManagerImpl.build[F](config.members)
      membership      <- ClusterConfigStorageImpl.build[F](config.members.toSet + config.local)
      log <- LogImpl
        .build[F, SM](
          raftId,
          storage.logStorage,
          storage.snapshotStorage,
          stateMachine,
          compactionPolicy,
          membership,
          appliedIndex
        )
      replicator <- LogPropagatorImpl.build[F](config.local, clientProvider, log)
      announcer <- LeaderAnnouncerImpl.build[F]
      heartbeat <- Ref.of[F, Long](OL)
      ref       <- Ref.of[F, NodeState](nodeState)
      running   <- Ref.of[F, Boolean](false)
    } yield new RaftImpl[F, SM](
      raftId,
      None,
      config,
      membership,
      clientProvider,
      announcer,
      replicator,
      log,
      storage,
      ref,
      heartbeat,
      running
    )
}

```

SingleRaftCluster.scala

```

class SingleRaftCluster[F[_]: Monad, +SM[X[_]] <: StateMachine[X]] (val rpc: RpcServer[F],
                                                               val raft: Raft[F, SM]

```

```

) extends Cluster[F, SM] {

  def start: F[Node] =
    raft.initialize >> rpc.start >> raft.start

  def stop: F[Unit] =
    rpc.stop >> raft.stop

  def join(node: Node): F[Node] =
    raft.initialize >> rpc.start >> raft.join(node)

  def leave: F[Unit] =
    raft.leave

  def leader: F[Node] =
    raft.listen

  def execute[T](command: Command[T]): F[T] =
    raft.onCommand(command)
}

```

Command.scala

```

sealed trait Command[OUT] extends Serializable
trait ReadCommand[OUT] extends Command[OUT]
trait WriteCommand[OUT] extends Command[OUT]

object Command {
  implicit def discriminated[OUT]: Discriminated[Command[OUT], Int] = Discriminated(uint8)
}

sealed trait ClusterConfigurationCommand extends WriteCommand[Unit] {
  def toConfig: ClusterConfiguration
}

case class JointConfigurationCommand(oldMembers: Set[Node], newMembers: Set[Node])
  extends ClusterConfigurationCommand {
  override def toConfig: ClusterConfiguration = JointClusterConfiguration(oldMembers,
newMembers)
}

case class NewConfigurationCommand(members: Set[Node]) extends ClusterConfigurationCommand {
  override def toConfig: ClusterConfiguration = NewClusterConfiguration(members)
}

```

Configuration.scala

```

case class Configuration(local: Node,
                        members: Seq[Node] = List.empty,
                        followerAcceptRead: Boolean = true,
                        logCompactionThreshold: Int = 100,
                        electionMinDelayMillis: Int = 150,
                        electionMaxDelayMillis: Int = 300,
                        heartbeatIntervalMillis: Int = 2000,
                        heartbeatTimeoutMillis: Int = 6000
) {
  def nodeId: String = local.toString

  def nodes: List[String] = local.toString :: members.map(_.toString).toList
}

```

LogCompactionPolicy.scala

```

/** Policy which defines when to apply compaction stage
 */
trait LogCompactionPolicy[F[_]] {

```

```

/** Check whether should compaction start
 * @param state Current state of [[Log]]
 * @param stateMachine Current state of [[StateMachine]]
 * @return
 */
def eligible(state: LogState, stateMachine: StateMachine[F]): F[Boolean]
}

object LogCompactionPolicy {

  def noCompaction[F[_]: Applicative]: LogCompactionPolicy[F] =
    (_: LogState, _: StateMachine[F]) => false.pure[F]

  def fixedSize[F[_]: Applicative](logsCount: Int): LogCompactionPolicy[F] =
    (state: LogState, _: StateMachine[F]) =>
      (state.lastAppliedIndex > logsCount && state.lastAppliedIndex % logsCount == 0).pure[F]
}

```

LogEntry.scala

```
case class LogEntry(term: Long, index: Long, command: Command[_])
```

Node.scala

```

case class Node(host: String, port: Int) {
  override def toString: String = id
  def id: String = s"$host:$port"
}

object Node {
  def fromString(str: String): Option[Node] =
    str.split(":") match {
      case Array(host, ip) => Some(Node(host, ip.toInt))
      case _                => None
    }
}

```

PersistedState.scala

```

case class PersistedState(term: Long, votedFor: Option[Node], appliedIndex: Long = 0L) {
  def toNodeState(raftId: Int, nodeId: Node): NodeState =
    FollowerNode(raftId, nodeId, term, votedFor = votedFor)
}

```

Snapshot.scala

```
case class Snapshot(lastIndex: Long, bytes: ByteBuffer, config: ClusterConfiguration)
```

CandidateNode.scala

```

case class CandidateNode(raftId: Int,
                        currentNode: Node,
                        currentTerm: Long,
                        lastTerm: Long,
                        votedFor: Option[Node] = None,
                        votedReceived: Set[Node] = Set.empty
) extends NodeState {

  private def announceLeader(otherNodes: Set[Node],
                            logState: LogState
): (LeaderNode, List[Action]) = {
  val matchIndex: Map[Node, Long] = otherNodes.map(n => (n, 0L)).toMap

```

```

val nextIndex: Map[Node, Long] = otherNodes.map(n => (n, logState.lastLogIndex + 1)).toMap
val actions: List[ReplicateLog] =
  otherNodes.map(n => ReplicateLog(n, currentTerm, logState.lastLogIndex + 1)).toList

LeaderNode(raftId, currentNode, currentTerm, matchIndex, nextIndex) ->
  (StoreState :: AnnounceLeader(currentNode, resetPrevious = false) :: actions)
}

override def onElectionTimer(logState: LogState,
                           config: ClusterConfiguration,
                           allowed: Boolean
): (NodeState, List[Action]) = {
  val electionTerm = currentTerm + 1
  val lastTerm_ = logState.lastLogTerm.getOrElse(lastTerm)
  val request = VoteRequest(raftId, currentNode, electionTerm, logState.lastLogIndex,
lastTerm_)

  val otherNodes = config.members.filterNot(_ == currentNode)
  val actions = otherNodes.toList.map(nodeId => RequestForVote(nodeId, request))

  if (config.members.size == 1) announceLeader(otherNodes, logState)
  else {
    this.copy(
      currentTerm = electionTerm,
      lastTerm = lastTerm_,
      votedFor = Some(currentNode),
      votedReceived = Set(currentNode)
    ) -> (StoreState :: actions)
  }
}

override def onVoteRequest(logState: LogState,
                           config: ClusterConfiguration,
                           msg: VoteRequest
): (NodeState, (VoteResponse, List[Action])) = {
  val myLogTerm = logState.lastLogTerm.getOrElse(0L)
  val logOK =
    (msg.lastLogTerm > myLogTerm) || (msg.lastLogTerm == myLogTerm && msg.lastLogIndex >=
logState.lastLogIndex)
  val termOK =
    (msg.term > currentTerm) || (msg.term == currentTerm && (votedFor.isEmpty || votedFor
      .contains(msg.nodeId)))

  if (logOK && termOK) {
    FollowerNode(raftId, currentNode, msg.term, Some(msg.nodeId), None) -> (VoteResponse(
      raftId,
      currentNode,
      msg.term,
      voteGranted = true
    ) -> List(StoreState))
  } else {
    this -> (VoteResponse(raftId, currentNode, currentTerm, voteGranted = false) ->
List.empty)
  }
}

override def onVoteResponse(logState: LogState,
                           config: ClusterConfiguration,
                           msg: VoteResponse
): (NodeState, List[Action]) = {
  val votedReceived_ = if (msg.voteGranted) votedReceived + msg.nodeId else votedReceived
  val quorumSize = (config.members.size + 1) / 2

  if (msg.term > currentTerm)
    FollowerNode(raftId, currentNode, msg.term) -> List(StoreState)
  else if (msg.term == currentTerm && msg.voteGranted && votedReceived_.size >= quorumSize)
    announceLeader(config.members.filterNot(_ == currentNode), logState)
  else this.copy(votedReceived = votedReceived_) -> List.empty
}

override def onEntries(logState: LogState,
                        config: ClusterConfiguration,
                        msg: AppendEntries,
                        localPrvLogEntry: Option[LogEntry]

```

```

): (NodeState, (AppendEntriesResponse, List[Action])) = 
  if (msg.term < currentTerm) {
    this -> (AppendEntriesResponse(
      raftId,
      currentNode,
      currentTerm,
      msg.prevLogIndex,
      success = false
    ) -> List.empty)
  } else if (msg.term > currentTerm) {

    val nextState =
      FollowerNode(raftId, currentNode, msg.term, currentLeader = Some(msg.leaderId))
    val actions = List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = false))

    if (msg.prevLogIndex > 0 && localPrvLogEntry.isEmpty)
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    else if (localPrvLogEntry.isDefined && localPrvLogEntry.get.term != msg.prevLogTerm)
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    else
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex + msg.entries.length,
        success = true
      ) -> actions)
  } else {

    val nextState =
      FollowerNode(raftId, currentNode, msg.term, currentLeader = Some(msg.leaderId))
    val actions = List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = false))

    if (msg.prevLogIndex > 0 && localPrvLogEntry.isEmpty)
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    else if (localPrvLogEntry.isDefined && localPrvLogEntry.get.term != msg.prevLogTerm) {
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    } else
      nextState -> (AppendEntriesResponse(
        raftId,
        currentNode,
        msg.term,
        msg.prevLogIndex + msg.entries.length,
        success = true
      ) -> actions)
  }

override def onEntriesResp(logState: LogState,
                           cluster: ClusterConfiguration,
                           msg: AppendEntriesResponse

```

```

): (NodeState, List[Action]) = this -> List.empty

override def onReplicateLog(config: ClusterConfiguration): List[Action] =
List.empty

override def leader: Option[Node] =
None

override def toPersistedState: PersistedState =
PersistedState(currentTerm, votedFor)

override def onSnapshotInstalled(logState: LogState,
cluster: ClusterConfiguration
): (NodeState, AppendEntriesResponse) =
this -> AppendEntriesResponse(
raftId,
currentNode,
currentTerm,
logState.lastAppliedIndex,
success = false
)
}
}

```

FollowerNode.scala

```

case class FollowerNode(raftId: Int,
currentNode: Node,
currentTerm: Long,
votedFor: Option[Node] = None,
currentLeader: Option[Node] = None
) extends NodeState {

override def onElectionTimer(logState: LogState,
config: ClusterConfiguration,
allowed: Boolean
): (NodeState, List[Action]) =
if (allowed)
CandidateNode(raftId, currentNode, currentTerm, logState.lastLogTerm.getOrElse(0L))
.onElectionTimer(logState, config, allowed) match {
case result @ (_: LeaderNode, _) => result
case (state, actions) if currentLeader.isDefined =>
state -> (ResetLeaderAnnouncer :: actions)
case otherResult => otherResult
}
else (this, List.empty)

override def onVoteRequest(logState: LogState,
config: ClusterConfiguration,
msg: VoteRequest
): (NodeState, (VoteResponse, List[Action])) = {
val myLogTerm = logState.lastLogTerm.getOrElse(0L)
val logOK =
(msg.lastLogTerm > myLogTerm) || (msg.lastLogTerm == myLogTerm && msg.lastLogIndex >=
logState.lastLogIndex)
val termOK =
(msg.term > currentTerm) || (msg.term == currentTerm && (votedFor.isEmpty || votedFor
.contains(msg.nodeId)))

if (logOK && termOK)
this.copy(currentTerm = msg.term, votedFor = Some(msg.nodeId)) -> (VoteResponse(
raftId,
currentNode,
msg.term,
voteGranted = true
) -> List(StoreState))
else
this -> (VoteResponse(raftId, currentNode, currentTerm, voteGranted = false) ->
List.empty)
}

override def onVoteResponse(logState: LogState,
config: ClusterConfiguration,

```

```

        msg: VoteResponse
): (NodeState, List[Action]) =
  this -> List.empty

override def onEntries(logState: LogState,
                      config: ClusterConfiguration,
                      msg: AppendEntries,
                      localPrevLogEntry: Option[LogEntry]
): (NodeState, (AppendEntriesResponse, List[Action])) =
  if (msg.term < currentTerm) {
    this -> (AppendEntriesResponse(
      raftId,
      currentNode,
      currentTerm,
      msg.prevLogIndex,
      success = false
    ) -> List.empty)
  } else {
    if (msg.term > currentTerm) {
      val nextState = this.copy(currentTerm = msg.term, currentLeader = Some(msg.leaderId))
      val actions =
        if (currentLeader.isEmpty)
          List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = false))
        else if (currentLeader.contains(msg.leaderId))
          List(StoreState)
        else
          List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = true))

      if (msg.prevLogIndex > 0 && localPrevLogEntry.isEmpty)
        nextState -> (AppendEntriesResponse(
          raftId,
          currentNode,
          msg.term,
          msg.prevLogIndex,
          success = false
        ) -> actions)
      else if (localPrevLogEntry.isDefined && localPrevLogEntry.get.term != msg.prevLogTerm)
        nextState -> (AppendEntriesResponse(
          raftId,
          currentNode,
          msg.term,
          msg.prevLogIndex,
          success = false
        ) -> actions)
      else
        nextState -> (AppendEntriesResponse(
          raftId,
          currentNode,
          msg.term,
          msg.prevLogIndex + msg.entries.length,
          success = true
        ) -> actions)
    } else {
      val (nextState, actions) =
        if (currentLeader.isEmpty)
          this.copy(currentLeader = Some(msg.leaderId)) -> List(
            AnnounceLeader(msg.leaderId, resetPrevious = false)
          )
        else if (currentLeader.contains(msg.leaderId))
          this -> List.empty[Action]
        else
          this.copy(currentLeader = Some(msg.leaderId)) -> List(
            AnnounceLeader(msg.leaderId, resetPrevious = true)
          )

      if (msg.prevLogIndex > 0 && localPrevLogEntry.isEmpty)
        nextState -> (AppendEntriesResponse(
          raftId,
          currentNode,
          msg.term,
          msg.prevLogIndex,
          success = false
        ) -> actions)
    }
  }
}

```

```

        else if (localPrevLogEntry.isDefined && localPrevLogEntry.get.term != msg.prevLogTerm)
    {
        nextState -> (AppendEntriesResponse(
            raftId,
            currentNode,
            msg.term,
            msg.prevLogIndex,
            success = false
        ) -> actions)
    } else
        nextState -> (AppendEntriesResponse(
            raftId,
            currentNode,
            msg.term,
            msg.prevLogIndex + msg.entries.length,
            success = true
        ) -> actions)
    }
}

override def onEntriesResp(logState: LogState,
                           cluster: ClusterConfiguration,
                           msg: AppendEntriesResponse
): (NodeState, List[Action]) = this -> List.empty

override def onReplicateLog(config: ClusterConfiguration): List[Action] =
List.empty

override def leader: Option[Node] =
currentLeader

override def toPersistedState: PersistedState =
PersistedState(currentTerm, votedFor)

override def onSnapshotInstalled(logState: LogState,
                                 config: ClusterConfiguration
): (NodeState, AppendEntriesResponse) =
this -> AppendEntriesResponse(
    raftId,
    currentNode,
    currentTerm,
    logState.lastLogIndex - 1,
    success = true
)
}
}

```

LeaderNode.scala

```

case class LeaderNode(raftId: Int,
                      currentNode: Node,
                      currentTerm: Long,
                      matchIndex: Map[Node, Long],
                      nextIndex: Map[Node, Long])
extends NodeState {

override def onElectionTimer(logState: LogState,
                             config: ClusterConfiguration,
                             allowed: Boolean
): (NodeState, List[Action]) =
(this, List.empty)

override def onVoteRequest(logState: LogState,
                           config: ClusterConfiguration,
                           msg: VoteRequest
): (NodeState, (VoteResponse, List[Action])) = {
    val lastTerm = logState.lastLogTerm.getOrElse(currentTerm)
    val logOK =
        (msg.lastLogTerm > lastTerm) || (msg.lastLogTerm == lastTerm && msg.lastLogIndex >=
logState.lastLogIndex)
    val termOK = msg.term > currentTerm

    if (logOK && termOK)

```

```

FollowerNode(raftId, currrentNode, msg.term, Some(msg.nodeId)) -> (VoteResponse(
    raftId,
    currrentNode,
    msg.term,
    voteGranted = true
) -> List(StoreState, ResetLeaderAnnouncer))
else {
    val nextIndex_ = nextIndex + (msg.nodeId -> (msg.lastLogIndex + 1))
    val matchIndex_ = matchIndex + (msg.nodeId -> msg.lastLogIndex)

    this.copy(nextIndex = nextIndex_, matchIndex = matchIndex_) -> (VoteResponse(
        raftId,
        currrentNode,
        currentTerm,
        voteGranted = false
) -> List(ReplicateLog(msg.nodeId, currentTerm, msg.lastLogIndex + 1)))
}
}

override def onVoteResponse(logState: LogState,
                           config: ClusterConfiguration,
                           msg: VoteResponse
): (NodeState, List[Action]) =
(this, List.empty)

override def onEntries(state: LogState,
                        config: ClusterConfiguration,
                        msg: AppendEntries,
                        localPrvLogEntry: Option[LogEntry]
): (NodeState, (AppendEntriesResponse, List[Action])) =
if (msg.term < currentTerm) {
    this -> (AppendEntriesResponse(
        raftId,
        currrentNode,
        currentTerm,
        msg.prevLogIndex,
        success = false
) -> List.empty)
} else if (msg.term > currentTerm) {

    val nextState =
        FollowerNode(raftId, currrentNode, msg.term, currentLeader = Some(msg.leaderId))
    val actions = List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = true))

    if (msg.prevLogIndex > 0 && localPrvLogEntry.isEmpty)
        nextState -> (AppendEntriesResponse(
            raftId,
            currrentNode,
            msg.term,
            msg.prevLogIndex,
            success = false
        ) -> actions)
    else if (localPrvLogEntry.isDefined && localPrvLogEntry.get.term != msg.prevLogTerm)
        nextState -> (AppendEntriesResponse(
            raftId,
            currrentNode,
            msg.term,
            msg.prevLogIndex,
            success = false
        ) -> actions)
    else
        nextState -> (AppendEntriesResponse(
            raftId,
            currrentNode,
            msg.term,
            msg.prevLogIndex + msg.entries.length,
            success = true
        ) -> actions)
} else {

    val nextState =
        FollowerNode(raftId, currrentNode, msg.term, currentLeader = Some(msg.leaderId))
    val actions = List(StoreState, AnnounceLeader(msg.leaderId, resetPrevious = true))
}

```

```

    if (msg.prevLogTerm > 0 && localPrvLogEntry.isEmpty)
      nextState -> (AppendEntriesResponse(
        raftId,
        currrentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    else if (localPrvLogEntry.isDefined && localPrvLogEntry.get.term != msg.prevLogTerm) {
      nextState -> (AppendEntriesResponse(
        raftId,
        currrentNode,
        msg.term,
        msg.prevLogIndex,
        success = false
      ) -> actions)
    } else
      nextState -> (AppendEntriesResponse(
        raftId,
        currrentNode,
        msg.term,
        msg.prevLogIndex + msg.entries.length,
        success = true
      ) -> actions)
    }

override def onEntriesResp(logState: LogState,
                           cluster: ClusterConfiguration,
                           msg: AppendEntriesResponse
): (NodeState, List[Action]) =
  if (msg.currentTerm > currentTerm) {
    FollowerNode(raftId, currrentNode, msg.currentTerm, None, None) -> List(
      StoreState,
      ResetLeaderAnnouncer
    )
  } else {
    if (msg.success) {
      val nextIndex_ = nextIndex + (msg.nodeId -> (msg.ack + 1L))
      val matchIndex_ = matchIndex + (msg.nodeId -> msg.ack)

      this.copy(matchIndex = matchIndex_, nextIndex = nextIndex_) -> List(
        CommitLogs(matchIndex_ + (currrentNode -> logState.lastLogIndex))
      )
    } else {
      //If AppendEntries fails because of log inconsistency - decrement nextIndex and retry
      val nodeNextIndex = nextIndex.get(msg.nodeId) match {
        case Some(next) if next == 1 => 1
        case Some(next)           => next - 1
        case None                => 1
      }
      val newNextIndex = nextIndex + (msg.nodeId -> nodeNextIndex)

      this.copy(nextIndex = newNextIndex) -> List(
        ReplicateLog(msg.nodeId, currentTerm, newNextIndex(msg.nodeId))
      )
    }
  }
}

override def onReplicateLog(cluster: ClusterConfiguration): List[Action] =
  cluster.members
  .filterNot(_ == currrentNode)
  .map(peer => ReplicateLog(peer, currentTerm, nextIndex.getOrElse(peer, 1L)))
  .toList

override def leader: Option[Node] =
  Some(currrentNode)

override def toPersistedState: PersistedState =
  PersistedState(currentTerm, Some(currrentNode))

override def onSnapshotInstalled(logState: LogState,
                                 cluster: ClusterConfiguration
): (NodeState, AppendEntriesResponse) =
  this -> AppendEntriesResponse(

```

```

        raftId,
        currrentNode,
        currentTerm,
        logState.lastLogIndex - 1,
        success = false
    )
}

}

```

NodeState.scala

```

/** Node state, describing current state and appropriate actions
 */
trait NodeState {
    val raftId: Int

    /** Transition applied on election timer timing out
     * @param logState Current log state, generated by log storage
     * @param config Current cluster configuration
     * @return Next node state and actions to achieve it
     */
    def onElectionTimer(logState: LogState,
                        config: ClusterConfiguration,
                        allowed: Boolean
    ): (NodeState, List[Action])

    def onVoteRequest(logState: LogState,
                      config: ClusterConfiguration,
                      msg: VoteRequest
    ): (NodeState, (VoteResponse, List[Action]))

    def onEntries(state: LogState,
                  config: ClusterConfiguration,
                  msg: AppendEntries,
                  localPrvLogEntry: Option[LogEntry]
    ): (NodeState, (AppendEntriesResponse, List[Action]))

    def onVoteResponse(logState: LogState,
                      config: ClusterConfiguration,
                      msg: VoteResponse
    ): (NodeState, List[Action])

    def onEntriesResp(logState: LogState,
                      config: ClusterConfiguration,
                      msg: AppendEntriesResponse
    ): (NodeState, List[Action])

    def onReplicateLog(config: ClusterConfiguration): List[Action]

    def onSnapshotInstalled(logState: LogState,
                           config: ClusterConfiguration
    ): (NodeState, AppendEntriesResponse)

    /** @return Current Leader [[Node]], can be [[None]]
     */
    def leader: Option[Node]

    /** @return Generated state to persist
     */
    def toPersistedState: PersistedState
}

```

Action.scala

```

sealed trait Action

case class RequestForVote(peerId: Node, request: VoteRequest)      extends Action
case class ReplicateLog(peerId: Node, term: Long, nextIndex: Long) extends Action
case class CommitLogs(matchIndex: Map[Node, Long])                 extends Action
case class AnnounceLeader(leaderId: Node, resetPrevious: Boolean)   extends Action

```

```

case object ResetLeaderAnnouncer
case object StoreState

```

```

extends Action
extends Action

```

AppendEntries.scala

```

case class AppendEntries(raftId: Int,
                        leaderId: Node,
                        term: Long,
                        prevLogIndex: Long,
                        prevLogTerm: Long,
                        leaderCommit: Long,
                        entries: List[LogEntry]
)

```

AppendEntriesResponse.scala

```

case class AppendEntriesResponse(raftId: Int,
                                 nodeId: Node,
                                 currentTerm: Long,
                                 ack: Long,
                                 success: Boolean
)

```

ClusterConfiguration.scala

```

sealed trait ClusterConfiguration {
  def members: Set[Node]
  def quorumReached(nodes: Set[Node]): Boolean
}

case class NewClusterConfiguration(members: Set[Node]) extends ClusterConfiguration {

  private val quorum = (members.size / 2) + 1

  override def quorumReached(nodes: Set[Node]): Boolean =
    nodes.intersect(members).size >= quorum
}

case class JointClusterConfiguration(oldMembers: Set[Node], newMembers: Set[Node])
  extends ClusterConfiguration {

  private val oldQuorum = (oldMembers.size / 2) + 1
  private val newQuorum = (newMembers.size / 2) + 1

  override def members: Set[Node] =
    oldMembers ++ newMembers

  override def quorumReached(nodes: Set[Node]): Boolean =
    (nodes.intersect(oldMembers).size >= oldQuorum) &&
    (nodes.intersect(newMembers).size >= newQuorum)
}

```

InstallSnapshot.scala

```

case class InstallSnapshot(raftId: Int, snapshot: Snapshot, lastEntry: LogEntry)

```

LogState.scala

```

case class LogState(lastLogIndex: Long, lastLogTerm: Option[Long], lastAppliedIndex: Long = 0)

```

VoteRequest.scala

```
case class VoteRequest(raftId: Int, nodeId: Node, term: Long, lastLogIndex: Long, lastLogTerm: Long)
```

VoteResponse.scala

```
case class VoteResponse(raftId: Int, nodeId: Node, term: Long, voteGranted: Boolean)
```

RpcClient.scala

```
trait RpcClient[F[_]] {
  def sendVote(voteRequest: VoteRequest): F[VoteResponse]

  def sendEntries	appendEntries: AppendEntries): F[AppendEntriesResponse]

  def sendCommand[T](raftId: Int, command: Command[T]): F[T]

  def sendSnapshot(snapshot: InstallSnapshot): F[AppendEntriesResponse]

  def join(raftId: Int, server: Node): F[Boolean]

  def close(): F[Unit]
}
```

RpcClientBuilder.scala

```
trait RpcClientBuilder[F[_]] {
  def build(address: Node)(implicit L: Logger[F]): F[RpcClient[F]]
}

object RpcClientBuilder {
  def apply[F[_]](implicit ev: RpcClientBuilder[F]): RpcClientBuilder[F] = ev
}
```

RpcServer.scala

```
trait RpcServer[F[_]] {
  val server: Server

  def start: F[Unit]
  def stop: F[Unit]
}
```

RpcServerBuilder.scala

```
trait RpcServerBuilder[F[_]] {
  def build(node: Node, rafts: RpcServerBuilder.RaftMap[F])(implicit L: Logger[F]): F[RpcServer[F]]
}

object RpcServerBuilder {
  type RaftMap[F[_]] = Ref[F, Map[Int, Raft[F, StateMachine]]]

  def apply[F[_]](implicit builder: RpcServerBuilder[F]): RpcServerBuilder[F] = builder
}
```

GrpcClientBuilder.scala

```
class GrpcClientBuilder[F[_]: Async](implicit
```

```

commandSer: ProtoSerializer[Command[_]],
configSer: ProtoSerializer[ClusterConfiguration],
objectSer: ProtoSerializer[Any]
) extends RpcClientBuilder[F] {

  override def build(address: Node)(implicit L: Logger[F]): F[RpcClient[F]] =
    GrpcClientBuilder
      .mkStub(address.host, address.port)
      .allocated
      .map(new GrpcRaftClient[F](address, _))
}

object GrpcClientBuilder {
  def mkStub[F[_]: Async](host: String, port: Int): Resource[F, RaftFs2Grpc[F, Metadata]] =
    NettyChannelBuilder
      .forAddress(host, port)
      .disableRetry()
      .usePlaintext()
      .resource[F]
      .flatMap(RaftFs2Grpc.stubResource[F](_))

  def apply[F[_]: Async](implicit
    commandSer: ProtoSerializer[Command[_]],
    configSer: ProtoSerializer[ClusterConfiguration],
    objectSer: ProtoSerializer[Any]
  ): GrpcClientBuilder[F] = new GrpcClientBuilder
}

```

GrpcServerBuilder.scala

```

class GrpcServerBuilder[F[_]: Async](implicit
  commandSer: ProtoSerializer[Command[_]],
  configSer: ProtoSerializer[ClusterConfiguration],
  objectSer: ProtoSerializer[Any]
) extends RpcServerBuilder[F] {
  override def build(node: Node, rafts: RpcServerBuilder.RaftMap[F])(implicit
    L: Logger[F]
  ): F[RpcServer[F]] =
    GrpcServerBuilder.construct(new GrpcRaftService[F](rafts), node.port)
}

object GrpcServerBuilder {
  def construct[F[_]: Async](impl: RaftFs2Grpc[F, Metadata], port: Int): F[RpcServer[F]] =
    RaftFs2Grpc
      .bindServiceResource[F](impl)
      .flatMap(service => NettyServerBuilder.forPort(port).addService(service).resource[F])
      .allocated
      .map { case (serverInst, release) =>
        new RpcServer[F] {
          override val server: Server = serverInst
          override def start: F[Unit] = Async[F].delay(server.start)
          override def stop: F[Unit] = release
        }
      }

  def apply[F[_]: Async](implicit
    commandSer: ProtoSerializer[Command[_]],
    configSer: ProtoSerializer[ClusterConfiguration],
    objectSer: ProtoSerializer[Any]
  ): GrpcServerBuilder[F] = new GrpcServerBuilder
}

```

GrpcRaftClient.scala

```

class GrpcRaftClient[F[_]: Sync](address: Node, stub: (RaftFs2Grpc[F, Metadata],
F[Unit]))(implicit
  logger: Logger[F],
  commandSer: ProtoSerializer[Command[_]],
  configSer: ProtoSerializer[ClusterConfiguration],
  objectSer: ProtoSerializer[Any]
)

```

```

) extends RpcClient[F] {

  private val stubService: RaftFs2Grpc[F, Metadata] = stub._1
  private def emptyMetadata: Metadata = new Metadata()

  override def sendVote(voteRequest: VoteRequest): F[VoteResponse] =
    stubService
      .vote(voteRequest.transformInto[rpc.VoteRequest], emptyMetadata)
      .map(_.transformInto[VoteResponse])
      .onError { error =>
        logger.warn(
          s"An error in sending VoteRequest to node: $address, Error: ${error.getMessage}"
        )
      }

  override def sendEntries	appendEntries: AppendEntries): F[AppendEntriesResponse] =
    stubService
      .appendEntries.appendEntries.transformInto[rpc.AppendEntriesRequest], emptyMetadata)
      .map(_.transformInto[AppendEntriesResponse])
      .onError { error =>
        logger.warn(
          s"An error in sending AppendEntries request to node: $address, Error:
${error.getMessage}"
        )
      }

  override def sendCommand[T](raftId: Int, command: Command[T]): F[T] =
    stubService
      .execute(rpc.CommandRequest(raftId, commandSer.encode(command)), emptyMetadata)
      .map(resp => objectSer.decode(resp.output).asInstanceOf[T])
      .onError { error =>
        logger.warn(
          s"An error in sending a command to node: $address. Command: $command, Error:
${error.getMessage}"
        )
      }

  override def sendSnapshot(snapshot: InstallSnapshot): F[AppendEntriesResponse] =
    stubService
      .installSnapshot(snapshot.transformInto[SnapshotStream[F]], emptyMetadata)
      .map(_.transformInto[AppendEntriesResponse])
      .onError { error =>
        logger.warn(
          s"An error in sending a snapshot to node: $address. Snapshot: $snapshot, Error:
${error.getMessage}"
        )
      }

  override def join(raftId: Int, server: Node): F[Boolean] =
    stubService
      .join(rpc.JoinRequest(raftId, server.host, server.port), emptyMetadata)
      .as(true)

  override def close(): F[Unit] = stub._2
}

```

GrpcRaftService.scala

```

class GrpcRaftService[F[_]: Async](raftRef: Ref[F, Map[Int, Raft[F, StateMachine]]])(implicit
  val logger: Logger[F],
  commandSer: ProtoSerializer[Command[_]],
  configSer: ProtoSerializer[ClusterConfiguration],
  objectSer: ProtoSerializer[Any])
  extends rpc.RaftFs2Grpc[F, Metadata] {

  private def useRaftById[T](raftId: Int)(op: Raft[F, StateMachine] => F[T]): F[T] =
    raftRef.get.flatMap(rafts =>
      rafts.get(raftId) match {
        case Some(raft) => op(raft)
        case None =>
          new RuntimeException(
            s"Failed to find raft for id: $raftId (has only ${rafts.keys.mkString(", ")})"
          )
      }
    )
}

```

```

        ).raiseError
    }
}
override def vote(request: rpc.VoteRequest, ctx: Metadata): F[rpc.VoteResponse] =
useRaftById(request.raftId) (
    .onVote(request.transformInto[VoteRequest])
    .map(_.transformInto[rpc.VoteResponse])
    .onError { error =>
        logger.warn(s"Error during the VoteRequest process. Error ${error.getMessage}")
    }
)

override def appendEntries(request: rpc.AppendEntriesRequest,
                           ctx: Metadata
): F[rpc.AppendEntriesResponse] =
useRaftById(request.raftId) (
    .onAppendEntries(request.transformInto[AppendEntries])
    .map(_.transformInto[rpc.AppendEntriesResponse])
)

override def execute(request: rpc.CommandRequest, ctx: Metadata): F[rpc.CommandResponse] =
useRaftById(request.raftId) (
    .onCommand(commandSer.decode(request.command))
    .map(response => rpc.CommandResponse(request.raftId, objectSer.encode(response)))
    .onError { error =>
        logger.warn(s"An error during the command process. Error ${error.getMessage}")
    }
)

override def installSnapshot(request: SnapshotStream[F],
                            ctx: Metadata
): F[rpc.AppendEntriesResponse] =
request
    .transformInto[F[InstallSnapshot]]
    .flatMap(request =>
        useRaftById(request.raftId) (
            .onSnapshot(request)
            .map(_.transformInto[rpc.AppendEntriesResponse])
            .onError { error =>
                logger.warn(s"An error during snapshot installation. Error ${error.getMessage}")
            }
        )
    )
}

override def join(request: rpc.JoinRequest, ctx: Metadata): F[rpc.JoinResponse] =
useRaftById(request.raftId) (
    .addMember(request.transformInto[Node]).as(rpc.JoinResponse(request.raftId))
)
}
}

```

JavaProtoSerializer.scala

```

class JavaProtoSerializer[T] extends ProtoSerializer[T] {

def encode(obj: T): ByteString = {
    val stream: ByteArrayOutputStream = new ByteArrayOutputStream()
    val oos = new ObjectOutputStream(stream)
    oos.writeObject(obj)
    oos.close()

    ByteString.copyFrom(stream.toByteArray)
}

def decode(byteString: ByteString): T = {
    val ois = new ObjectInputStream(byteString.newInput)
    val response = ois.readObject().asInstanceOf[T]
    ois.close()
    response
}
}

```

```

object JavaProtoSerializer {
  def anySerObject[T]: JavaProtoSerializer[T] = new JavaProtoSerializer[T]
}

```

ProtoSerializer.scala

```

trait ProtoSerializer[T] {
  def encode(obj: T): ByteString
  def decode(byteString: ByteString): T
}

object ProtoSerializer {
  def apply[T](implicit ev: ProtoSerializer[T]): ProtoSerializer[T] = ev
}

```

transform/instances.scala

```

object instances {
  implicit def id2Node: Transformer[String, Node] = (str: String) => Node.fromString(str).get
  implicit def node2Id: Transformer[Node, String] = (node: Node) => node.id

  implicit def val2Option[T]: Transformer[T, Option[T]] = (src: T) => Some(src)
  implicit def option2Val[T]: Transformer[Option[T], T] = (src: Option[T]) => src.get

  implicit def protoLogEntry2Domain(implicit
    commandSer: ProtoSerializer[Command[_]]
  ): Transformer[rpc.LogEntry, LogEntry] =
    Transformer
      .define[rpc.LogEntry, LogEntry]
      .withFieldComputed(_.command, e => commandSer.decode(e.command))
      .buildTransformer

  implicit def logEntry2Proto(implicit
    commandSer: ProtoSerializer[Command[_]]
  ): Transformer[LogEntry, rpc.LogEntry] =
    Transformer
      .define[LogEntry, rpc.LogEntry]
      .withFieldComputed(_.command, e => commandSer.encode(e.command))
      .buildTransformer

  implicit val protoVoteReq2Domain: Transformer[rpc.VoteRequest, VoteRequest] =
    Transformer
      .define[rpc.VoteRequest, VoteRequest]
      .withFieldRenamed(_.currentTerm, _.term)
      .withFieldRenamed(_.logLength, _.lastLogIndex)
      .withFieldRenamed(_.logTerm, _.lastLogTerm)
      .buildTransformer

  implicit val voteReq2Proto: Transformer[VoteRequest, rpc.VoteRequest] =
    Transformer
      .define[VoteRequest, rpc.VoteRequest]
      .withFieldComputed(_.nodeId, _.nodeId.id)
      .withFieldRenamed(_.term, _.currentTerm)
      .withFieldRenamed(_.lastLogIndex, _.logLength)
      .withFieldRenamed(_.lastLogTerm, _.logTerm)
      .buildTransformer

  implicit val protoVoteResp2Domain: Transformer[rpc.VoteResponse, VoteResponse] =
    Transformer
      .define[rpc.VoteResponse, VoteResponse]
      .withFieldRenamed(_.granted, _.voteGranted)
      .buildTransformer

  implicit val voteResp2Proto: Transformer[VoteResponse, rpc.VoteResponse] =
    Transformer
      .define[VoteResponse, rpc.VoteResponse]
      .withFieldComputed(_.nodeId, _.nodeId.id)
      .withFieldRenamed(_.voteGranted, _.granted)
      .buildTransformer
}

```

```

implicit def protoAppendReq2Domain(implicit
  commandSer: ProtoSerializer[Command[_]])
): Transformer[rpc.AppendEntriesRequest, AppendEntries] =
  Transformer
    .define[rpc.AppendEntriesRequest, AppendEntries]
    .withFieldRenamed(_.logLength, _.prevLogIndex)
    .withFieldRenamed(_.logTerm, _.prevLogTerm)
    .buildTransformer

implicit def appendReq2Proto(implicit
  commandSer: ProtoSerializer[Command[_]])
): Transformer[AppendEntries, rpc.AppendEntriesRequest] =
  Transformer
    .define[AppendEntries, rpc.AppendEntriesRequest]
    .withFieldComputed(_.leaderId, _.leaderId.id)
    .withFieldRenamed(_.prevLogIndex, _.logLength)
    .withFieldRenamed(_.prevLogTerm, _.logTerm)
    .buildTransformer

implicit val protoAppendResp2Domain: Transformer[rpc.AppendEntriesResponse,
AppendEntriesResponse] =
  Transformer.derive

implicit val appendResp2Proto: Transformer[AppendEntriesResponse, rpc.AppendEntriesResponse] =
  Transformer
    .define[AppendEntriesResponse, rpc.AppendEntriesResponse]
    .withFieldComputed(_.nodeId, _.nodeId.id)
    .buildTransformer

private val snapLimit: Int = 1 << 20
type SnapshotStream[F[_]] = fs2.Stream[F, rpc.InstallSnapshotRequest]

implicit def protoSnapshotReq2Domain[F[_]: Sync](implicit
  commandSer: ProtoSerializer[Command[_]],
  configSer: ProtoSerializer[ClusterConfiguration])
): Transformer[SnapshotStream[F], F[InstallSnapshot]] =
  (src: SnapshotStream[F]) =>
    src.compile
      .fold(List.empty[ByteString] -> Option.empty[InstallSnapshot]) {
        case (
          buffers, None),
          rpc.InstallSnapshotRequest(-1, -1, None, snapshotChunk, ByteString.EMPTY, _)
        ) =>
          (buffers :+ snapshotChunk) -> None
        case (
          buffers, Some(
            rpc
              .InstallSnapshotRequest(
                raftId,
                lastIndexId,
                lastEntry,
                ByteString.EMPTY,
                config,
                _)
            )
          ) =>
          buffers -> Some(
            InstallSnapshot(
              raftId,
              Snapshot(lastIndexId, null, configSer.decode(config)),
              lastEntry.transformInto[rpc.LogEntry].transformInto[LogEntry]
            )
          )
      }
    .map { case (buffers, Some(snapshotToFill)) =>
      val newBB = ByteBuffer.allocate(buffers.map(_.size).sum)
      buffers.foreach(binStr => newBB.put(binStr.asReadOnlyByteBuffer()))
      snapshotToFill.copy(snapshot = snapshotToFill.snapshot.copy(bytes = newBB))
    }

implicit def snapshotReq2Proto[F[_]: Sync](implicit
  commandSer: ProtoSerializer[Command[_]],
  configSer: ProtoSerializer[ClusterConfiguration])
): Transformer[InstallSnapshot, SnapshotStream[F]] = {

```

```

    case InstallSnapshot(raftId, Snapshot(lastIndex, bytes, config), lastEntry) =>
      val dataStream =
        fs2.Stream
          .chunk(fs2.Chunk.ByteBuffer(bytes.asReadOnlyBuffer()))
          .chunkLimit(snapLimit)
          .map(byteChunk => ByteString.copyFrom(byteChunk.toByteBuffer))
          .map(rpc.InstallSnapshotRequest(-1, -1, None, _, ByteString.EMPTY))
      val configStream = fs2.Stream.eval(
        Sync[F].delay(
          rpc.InstallSnapshotRequest(
            raftId,
            lastIndex,
            Some(lastEntry.transformInto[rpc.LogEntry]),
            ByteString.EMPTY,
            configSer.encode(config)
          )
        )
      )
      dataStream ++ configStream
    }

  implicit val joinReq2Node: Transformer[rpc.JoinRequest, Node] = Transformer.derive
}

```

ClientManager.scala

```

trait ClientManager[F[_]] {
  def sendVote(serverId: Node, voteRequest: VoteRequest): F[VoteResponse]
  def sendEntries(serverId: Node, appendEntries: AppendEntries): F[AppendEntriesResponse]
  def sendSnapshot(serverId: Node, snapshot: InstallSnapshot): F[AppendEntriesResponse]
  def sendCommand[T](serverId: Node, raftId: Int, command: Command[T]): F[T]
  def join(serverId: Node, raftId: Int, newNode: Node): F[Boolean]
  def closeConnections(): F[Unit]
}

```

ClusterConfigStorage.scala

```

/** Storage for current cluster configuration
 */
trait ClusterConfigStorage[F[_]] {
  /** @return Current nodes in the cluster
   */
  def members: F[Set[Node]]

  /** Applies new configuration to the cluster
   * @param newConfig New config to apply
   */
  def setClusterConfiguration(newConfig: ClusterConfiguration): F[Unit]

  /** Get current cluster configuration
   * @return Current [[ClusterConfiguration]]
   */
  def getClusterConfiguration: F[ClusterConfiguration]
}

```

LeaderAnnouncer.scala

```

trait LeaderAnnouncer[F[_]] {

```

```

def announce(leader: Node): F[Unit]
def reset(): F[Unit]
def listen(): F[Node]
}

Log.scala

/** Log entity. Stores events, manages state, applies commands
 */
trait Log[F[_], +SM[X[_]] <: StateMachine[X]] {
  val raftId: Int
  val logStorage: LogStorage[F]
  val snapshotStorage: SnapshotStorage[F]
  val stateMachine: SM[F]
  val clusterConfigStorage: ClusterConfigStorage[F]
  val compactionPolicy: LogCompactionPolicy[F]
  def transactional[A](t: => F[A]): F[A]
  /** @return Last committed index
   */
  def getCommitIndex: F[Long]
  /** Updated committed index
   * @param index new committed index
   */
  def setCommitIndex(index: Long): F[Unit]
  /** Initialize log and state machine state:
   * 1. Updates state machine up to latest snapshot
   * 2. Sets new commit index
   * 3. Applies commands from the log
   */
  def initialize: F[Unit]
  /** Composes current log state
   * @return Up to date [[LogState]]
   */
  def state: F[LogState]
  /** Get log entry by index
   * @param index specified index
   * @return optional log entry for index [[LogEntry]]
   */
  def get(index: Long): F[Option[LogEntry]]
  /** Apply read command
   * @param command Defined [[ReadCommand]]
   * @param T Expected command result
   * @return result of the command, specified by type
   */
  def applyReadCommand[T](command: ReadCommand[T]): F[T]
  /** Get all log entries which have higher or equal log index
   * @param leaderId Leader Node id
   * @param term Current RAFT term
   * @param nextIndex Index for start collection of entries
   * @return [[AppendEntries]] with all entries
   */
  def getAppendEntries(leaderId: Node, term: Long, nextIndex: Long): F[AppendEntries]
  /** Appends new command to log
   * and puts it in "awaiting commit" set
   * @param term Current RAFT term
   */
}

```

```

* @param command Command to process
* @param deferred Supplied deferred, which will be completed after command's commit
* @tparam T Command result type
* @return New [[LogEntry]]
*/
def append[T](term: Long, command: Command[T], deferred: Deferred[F, T]): F[LogEntry]

/** Appends new entries to the log, with respect to already applied by leader logs
 * Afterwards checks if compaction is needed
* @param entries New log entries
* @param leaderPrevLogIndex Leader's previous last log index
* @param leaderCommit Leader's last commit
* @return [[Boolean]] on whether there were any commits made
*/
def appendEntries(entries: List[LogEntry],
                  leaderPrevLogIndex: Long,
                  leaderCommit: Long
): F[Boolean]

/** Commit logs, based on nodes quorum
 * @param matchIndex State of every node index
* @return [[Boolean]] on whether there were any commits made
*/
def commitLogs(matchIndex: Map[Node, Long]): F[Boolean]

/** @return Optional latest [[Snapshot]]
 */
def latestSnapshot: F[Option[Snapshot]]

/** Apply snapshot to the logs state
 * @param snapshot Snapshot to apply
* @param lastEntry Last entry from snapshot ???
*/
def installSnapshot(snapshot: Snapshot, lastEntry: LogEntry): F[Unit]
}

```

LogPropagator.scala

```

/** Policy on how to propagate logs to other nodes
 */
trait LogPropagator[F[_]] {

/** Propagate logs from current node (leader)
 * @param raftId Id of current Raft Group
* @param peerId Id of receiving node
* @param term Current RAFT term
* @param nextIndex Index to pick latter logs
* @return [[AppendEntriesResponse]] contains status of logs propagation
*/
def propagateLogs(raftId: Int,
                  peerId: Node,
                  term: Long,
                  nextIndex: Long
): F[AppendEntriesResponse]
}

```

ClientManagerImpl.scala

```

class ClientManagerImpl[F[_]: MonadThrow: RpcClientBuilder: Logger](
  val clientsRef: Ref[F, Map[Node, RpcClient[F]]],
  val members: Seq[Node]
) extends ErrorLogging[F] with ClientManager[F] {

  def sendVote(serverId: Node, voteRequest: VoteRequest): F[VoteResponse] =
    for {
      client <- getClient(serverId)
      attempt <- client.sendVote(voteRequest).attempt
      result <- logErrors(serverId, attempt)
    } yield result
}

```

```

def sendEntries(serverId: Node, appendEntries: AppendEntries): F[AppendEntriesResponse] =
  errorLogging("Sending AppendEntries") {
    for {
      client   <- getClient(serverId)
      _        <- Logger[F].trace(s"Sending request $appendEntries to $serverId client")
      attempt <- client.sendEntries(appendEntries).attempt
      _        <- Logger[F].trace(s"Attempt $attempt")
      result   <- logErrors(serverId, attempt)
    } yield result
  }

def sendSnapshot(serverId: Node, snapshot: InstallSnapshot): F[AppendEntriesResponse] =
  for {
    client   <- getClient(serverId)
    attempt <- client.sendSnapshot(snapshot).attempt
    response <- logErrors(serverId, attempt)
  } yield response

def sendCommand[T](serverId: Node, raftId: Int, command: Command[T]): F[T] =
  for {
    client   <- getClient(serverId)
    attempt <- client.sendCommand(raftId, command).attempt
    result   <- logErrors(serverId, attempt)
  } yield result

def join(serverId: Node, raftId: Int, newNode: Node): F[Boolean] =
  for {
    client   <- getClient(serverId)
    attempt <- client.join(raftId, newNode).attempt
    result   <- logErrors(serverId, attempt)
  } yield result

private def logErrors[T](peerId: Node, result: Either[Throwable, T]): F[T] =
  result match {
    case Left(error) =>
      Logger[F].warn(s"An error during communication with $peerId. Error: $error") >>
      error.raiseError[F, T]
    case Right(value) =>
      Monad[F].pure(value)
  }

private def getClient(serverId: Node): F[RpcClient[F]] =
  clientsRef.get.flatMap(_.get(serverId)) match {
    case Some(client) => client.pure[F]
    case None =>
      RpcClientBuilder[F]
        .build(serverId)
        .flatMap(client => clientsRef.update(_ + (serverId -> client)).as(client))
  }

def closeConnections(): F[Unit] =
  Logger[F].trace("Close all connections to other members") >>
  clientsRef.get.flatMap(_.values.toList.traverse(_.close())).void
}

object ClientManagerImpl {

  def build[F[_]: Monad: Sync: RpcClientBuilder: Logger](
    members: Seq[Node]
  ): F[ClientManagerImpl[F]] =
    Ref.of[F, Map[Node, RpcClient[F]]](Map.empty).map(new ClientManagerImpl[F](_, members))
}

```

ClusterConfigStorageImpl.scala

```

class ClusterConfigStorageImpl[F[_]: Monad](configurationRef: Ref[F, ClusterConfiguration])
  extends ClusterConfigStorage[F] {

  def members: F[Set[Node]] =
    configurationRef.get.map(_.members)

```

```

def setClusterConfiguration(newConfig: ClusterConfiguration): F[Unit] =
  configurationRef.set(newConfig)

def getClusterConfiguration: F[ClusterConfiguration] =
  configurationRef.get

}

object ClusterConfigStorageImpl {
  def build[F[_]: Monad: Sync](members: Set[Node]): F[ClusterConfigStorageImpl[F]] =
    Ref
      .of[F, ClusterConfiguration](NewClusterConfiguration(members))
      .map(new ClusterConfigStorageImpl[F](_))
}

```

LeaderAnnouncerImpl.scala

```

class LeaderAnnouncerImpl[F[_]: Monad: Concurrent: Logger](
  val deferredRef: Ref[F, Deferred[F, Node]])
) extends LeaderAnnouncer[F] {

  def announce(leader: Node): F[Unit] =
    Logger[F].info(s"A new leader is elected among the members. New Leader is '$leader'.") >>
    deferredRef.get.flatMap(_.complete(leader)).void

  def reset(): F[Unit] =
    Logger[F].debug("Resetting the Announcer.") >>
    Deferred[F, Node].flatMap(deferredRef.set)

  def listen(): F[Node] =
    deferredRef.get.flatMap(_.get)
}

object LeaderAnnouncerImpl {
  def build[F[_]: Monad: Concurrent: Logger]: F[LeaderAnnouncerImpl[F]] =
    Ref.ofEffect(Deferred[F, Node]).map(new LeaderAnnouncerImpl[F](_))

}

```

LogImpl.scala

```

class LogImpl[F[_]: MonadCancel[*[_], Throwable]: Logger, +SM[X[_]] <: StateMachine[X]](
  val raftId: Int,
  val logStorage: LogStorage[F],
  val snapshotStorage: SnapshotStorage[F],
  val stateMachine: SM[F],
  val clusterConfigStorage: ClusterConfigStorage[F],
  val compactionPolicy: LogCompactionPolicy[F],
  commitIndexRef: Ref[F, Long],
  semaphore: Semaphore[F]
) extends Log[F, SM] {

  private val awaitingCommands = TrieMap[Long, Deferred[F, Any]]()

  override def transactional[A](code: => F[A]): F[A] =
    semaphore.permit.use(_ => code)

  override def getCommitIndex: F[Long] =
    commitIndexRef.get

  override def setCommitIndex(index: Long): F[Unit] =
    Logger[F].trace(s"Set commit index to $index") >>
    commitIndexRef.set(index)

  def initialize: F[Unit] =
    transactional {
      for {
        _snapshot <- Logger[F].debug("Initializing log")
        latestSnapshot <- snapshotStorage.map	restoreSnapshot).getOrElse(Monad[F].unit)
        _ <- latestSnapshot
      } yield
    }
}

```

```

commitIndex      <- getCommitIndex
                <- Logger[F].debug(s"Latest committed index $commitIndex")
stateMachineIndex <- stateMachine.appliedIndex
                <- Logger[F].debug(s"State machine applied index $stateMachineIndex")
                <-
                - if (stateMachineIndex > commitIndex) setCommitIndex(stateMachineIndex)
                  else (stateMachineIndex + 1 to commitIndex).toList.traverse(commit).void
} yield ()
}

def state: F[LogState] =
for {
  lastIndex <- logStorage.lastIndex
  lastTerm <-
    if (lastIndex > 0) logStorage.get(lastIndex).map(_.map(_.term))
    else Monad[F].pure(None)
  commitIndex <- getCommitIndex
} yield LogState(lastIndex, lastTerm, commitIndex)

def get(index: Long): F[Option[LogEntry]] =
logStorage.get(index)

def applyReadCommand[T](command: ReadCommand[T]): F[T] =
if (stateMachine.applyRead.isDefinedAt(command))
  stateMachine.applyRead(command).asInstanceOf[F[T]]
else new RuntimeException("Could not run the command").raiseError

def getAppendEntries(leaderId: Node, term: Long, nextIndex: Long): F[AppendEntries] =
for {
  <- Logger[F].trace(s"Getting append entries since $nextIndex")
  lastIndex <- logStorage.lastIndex
  lastEntry <-
    if (nextIndex > 1) logStorage.get(nextIndex - 1) else Monad[F].pure(None)
  commitIndex <- getCommitIndex
  entries   <- (nextIndex to lastIndex).toList.traverse(i =>
logStorage.get(i).map(_.get))
  prevLogIndex = lastEntry.map(_.index).getOrElse(0L)
  prevLogTerm  = lastEntry.map(_.term).getOrElse(0L)
} yield AppendEntries(raftId, leaderId, term, prevLogIndex, prevLogTerm, commitIndex,
entries)

def append[T](term: Long, command: Command[T], deferred: Deferred[F, T]): F[LogEntry] =
transactional {
  for {
    lastIndex <- logStorage.lastIndex
    logEntry   = LogEntry(term, lastIndex + 1, command)
    <-
    - Logger[F].trace(s"Appending a command to the log. Term: $term, Index: ${logEntry.index}")
    - <- logStorage.put(logEntry.index, logEntry)
    - = awaitingCommands.put(logEntry.index, deferred.asInstanceOf[Deferred[F, Any]])
    - <- Logger[F].trace(s"Entry appended. Term: $term, Index: ${lastIndex + 1}")
  } yield logEntry
}

def appendEntries(entries: List[LogEntry],
                 leaderPrevLogIndex: Long,
                 leaderCommit: Long
): F[Boolean] =
transactional {
  for {
    lastIndex     <- logStorage.lastIndex
    appliedIndex  <- getCommitIndex
    -             <- truncateInconsistentLogs(entries, leaderPrevLogIndex, lastIndex)
    -             <- putEntries(entries, leaderPrevLogIndex, lastIndex)
    -             <- Logger[F].trace(s"Commit index: $appliedIndex | Entries: $entries")
    committed     <- (appliedIndex + 1 to leaderCommit).toList.traverse(commit)
    -             <- if (committed.nonEmpty) compactLogs() else Monad[F].unit
  } yield committed.nonEmpty
}

private def truncateInconsistentLogs(entries: List[LogEntry],
                                      leaderLogSent: Long,
                                      lastLogIndex: Long

```

```

): F[Unit] =
  if (entries.nonEmpty && lastLogIndex > leaderLogSent) {
    Logger[F].trace(s"Truncating log entries from the Log after: $leaderLogSent") >>
    logStorage.get(lastLogIndex).flatMap {
      case Some(entry) if entry.term != entries.head.term =>
        logStorage.deleteAfter(leaderLogSent)
      case _ => Monad[F].unit
    }
  } else Monad[F].unit

private def putEntries(entries: List[LogEntry],
                      leaderPrevLogIndex: Long,
                      logLastIndex: Long
): F[Unit] = {
  val logEntries =
    if ((leaderPrevLogIndex + entries.length) > logLastIndex)
      entries.drop((logLastIndex - leaderPrevLogIndex).toInt)
    else List.empty

  Logger[F].trace(s"Putting entries $logEntries to log storage") >>
  logEntries.traverse(entry => logStorage.put(entry.index, entry)).void
}

def commitLogs(matchIndex: Map[Node, Long]): F[Boolean] =
  transactional {
  for {
    lastIndex   <- logStorage.lastIndex
    commitIndex <- getCommitIndex
    committed   <- (commitIndex + 1 to
lastIndex).toList.traverse(commitIfMatched(matchIndex, _))
    <- if (committed.contains(true)) compactLogs() else Monad[F].unit
  } yield committed.contains(true)
}

private def applyCommand(index: Long, command: Command[_]): F[Unit] = {
  val output = command match {
    case command: ClusterConfigurationCommand =>
      clusterConfigStorage.setClusterConfiguration(command.toConfig)
    case command: ReadCommand[_] =>
      stateMachine.applyRead.apply(command)
    case command: WriteCommand[_] =>
      stateMachine.applyWrite.apply((index, command))
  }

  output.flatMap { result =>
    awaitingCommands.get(index) match {
      case Some(deferred) => deferred.complete(result).map(_ =>
awaitingCommands.remove(index))
      case None           => Monad[F].unit
    }
  }
}

private def commitIfMatched(matchIndex: Map[Node, Long], index: Long): F[Boolean] =
  for {
    config <- clusterConfigStorage.getClusterConfiguration
    matched = config.quorumReached(matchIndex.filter(_.value >= index).keySet)
    result <- if (matched) commit(index).as(true) else Monad[F].pure(false)
  } yield result

private def commit(index: Long): F[Unit] =
  for {
    _       <- Logger[F].trace(s"Committing the log entry at index $index")
    entry  <- logStorage.get(index)
    _       <- Logger[F].trace(s"Committed the log entry at index $index $entry")
    _       <- applyCommand(index, entry.get.command)
    _       <- setCommitIndex(index)
  } yield ()

def latestSnapshot: F[Option[Snapshot]] =
  snapshotStorage.retrieveSnapshot

def installSnapshot(snapshot: Snapshot, lastEntry: LogEntry): F[Unit] =
  transactional {

```

```

for {
    lastIndex <- logStorage.lastIndex
    - <-
    - if (lastIndex >= snapshot.lastIndex)
        new RuntimeException("A new snapshot is already applied").raiseError
    else Monad[F].unit
    - <- Logger[F].trace(s"Installing a snapshot, $snapshot")
    - <- snapshotStorage.saveSnapshot(snapshot)
    - <- Logger[F].trace("Restoring state from snapshot")
    - <- restoreSnapshot(snapshot)
    - <- logStorage.put(lastEntry.index, lastEntry)
    - <- setCommitIndex(snapshot.lastIndex)
} yield ()
}

private def compactLogs(): F[Unit] =
for {
    logState <- state
    eligible <- compactionPolicy.eligible(logState, stateMachine)
    - <- if (eligible) takeSnapshot() else Monad[F].unit
} yield ()

private def takeSnapshot(): F[Unit] =
for {
    - <- Logger[F].trace("Starting to take snapshot")
    snapshot <- stateMachine.takeSnapshot
    config <- clusterConfigStorage.getClusterConfiguration
    - <- snapshotStorage.saveSnapshot(Snapshot(snapshot._1, snapshot._2, config))
    - <- Logger[F].trace(s"Snapshot is stored $snapshot")
    - <- Logger[F].trace(s"Deleting logs before ${snapshot._1}")
    - <- logStorage.deleteBefore(snapshot._1)
    - <- Logger[F].trace(s"Logs before ${snapshot._1} are deleted.")
} yield ()

private def restoreSnapshot(snapshot: Snapshot): F[Unit] =
Logger[F].trace(
  s"Restoring Snapshot(index: ${snapshot.lastIndex}), Config: ${snapshot.config}"
) >>
clusterConfigStorage.setClusterConfiguration(snapshot.config) >>
stateMachine.restoreSnapshot(snapshot.lastIndex, snapshot.bytes) >>
Logger[F].trace("Snapshot is restored.")

object LogImpl {
def build[F[_]: Concurrent: Logger, SM[X[_]] <: StateMachine[X]](
  raftId: Int,
  logStorage: LogStorage[F],
  snapshotStorage: SnapshotStorage[F],
  stateMachine: SM[F],
  compactionPolicy: LogCompactionPolicy[F],
  membershipManager: ClusterConfigStorage[F],
  lastCommitIndex: Long
): F[LogImpl[F, SM]] =
  for {
    lock           <- Semaphore[F](1)
    commitIndexRef <- Ref.of[F, Long](lastCommitIndex)
  } yield new LogImpl(
    raftId,
    logStorage,
    snapshotStorage,
    stateMachine,
    membershipManager,
    compactionPolicy,
    commitIndexRef,
    lock
  )
}
}

```

LogPropagatorImpl.scala

```

class LogPropagatorImpl[F[_]: Concurrent: Logger](leaderId: Node,
                                                log: Log[F, StateMachine],

```

```

        clients: ClientManager[F],
        installingRef: Ref[F, Set[Node]]}

) extends LogPropagator[F] {

  def propagateLogs(raftId: Int,
                    peerId: Node,
                    term: Long,
                    nextIndex: Long
      ): F[AppendEntriesResponse] = {
    for {
      <- Logger[F].trace(s"Replicating logs to $peerId. Term: $term, nextIndex: $nextIndex")
      _snapshot <- log.latestSnapshot
      <- Logger[F].trace(s"Latest snapshot: $snapshot")
      response <-
        if (snapshot.exists(_.lastIndex >= nextIndex)) sendSnapshot(raftId, peerId, snapshot.get)
        else log.appendEntries(leaderId, term, nextIndex).flatMap(clients.sendEntries(peerId, _))
    } yield response
  }

  private def sendSnapshot(raftId: Int,
                           peerId: Node,
                           snapshot: Snapshot
      ): F[AppendEntriesResponse] = {
    val response = for {
      <- Logger[F].trace(s"Installing an Snapshot for peer $peerId, snapshot: $snapshot")
      _logEntry <- log.get(snapshot.lastIndex).map(_.get)
      response <- clients.sendSnapshot(peerId, InstallSnapshot(raftId, snapshot, logEntry))
      <- Logger[F].trace(s"Response after installing snapshot $response")
      _installingRef.update(_ - peerId)
    } yield response
    Concurrent[F].onError(response) { case error =>
      Logger[F].trace(s"Error during snapshot installation $error") >>
      installingRef.update(_ - peerId)
    }
  }

  private def snapshotIsNotInstalling(peerId: Node): F[Unit] =
    installingRef.get.flatMap(set =>
      if (set.contains(peerId))
        Concurrent[F].raiseError(new RuntimeException("Client is installing snapshot"))
      else Concurrent[F].unit
    )
  }

  object LogPropagatorImpl {
    def build[F[_]: Concurrent: Logger](leaderId: Node,
                                        clients: ClientManager[F],
                                        log: Log[F, StateMachine]
                                      ): F[LogPropagatorImpl[F]] =
      Ref.of[F, Set[Node]](Set.empty).map(new LogPropagatorImpl[F](leaderId, log, clients, _))
  }
}

```

LogStorage.scala

```

/** A place where to store logs (commands)
 */
trait LogStorage[F[_]] {

  /** Retrieve last log's index
   * @return last index
   */
  def lastIndex: F[Long]

  /** Get [[LogEntry]] by it's index
   * @param index index of the log
   * @return Log entry [[LogEntry]] or None
   */
}

```

```

*/
def get(index: Long): F[Option[LogEntry]]

/** Put [[LogEntry]] to log storage by index
 * @param index index of new entry
 * @param logEntry Log entry
 * @return identity of logEntry
 */
def put(index: Long, logEntry: LogEntry): F[LogEntry]

/** Delete log applied before provided index
 * @param index predicate index to filter by
 */
def deleteBefore(index: Long): F[Unit]

/** Delete log applied after provided index
 * @param index predicate index to filter by
 */
def deleteAfter(index: Long): F[Unit]
}

```

SnapshotStorage.scala

```

/** Storage of saved snapshots (compacted logs)
 */
trait SnapshotStorage[F[_]] {

  /** Save snapshot to storage
   * @param snapshot provided [[Snapshot]]
   */
  def saveSnapshot(snapshot: Snapshot): F[Unit]

  /** Retrieve snapshot from persistent storage and update local ref
   * @return Optional latest [[Snapshot]]
   */
  def retrieveSnapshot: F[Option[Snapshot]]

  /** Get local latest snapshot
   * @return Optional local [[Snapshot]]
   */
  def getLatestSnapshot: F[Option[Snapshot]]
}

```

StateMachine.scala

```

/** De-facto main storage, where logs are applied (i.e. DB engine)
 */
trait StateMachine[F[_]] {
  type WriteHandler = PartialFunction[(Long, WriteCommand[_]), F[_]]
  type ReadHandler = PartialFunction[ReadCommand[_], F[_]]

  /** Partially defined handler for [[WriteCommand]]
   *
   * @return Result of writing command
   */
  def applyWrite: WriteHandler

  /** Partially defined handler for [[ReadCommand]]
   *
   * @return Result of reading command
   */
  def applyRead: ReadHandler

  /** @return Index of last applied log
   */
  def appliedIndex: F[Long]

  /** Get snapshot info for compaction
   * @return Last stored index [[Long]] and binary data of current state [[ByteBuffer]]
   */
  def takeSnapshot: F[(Long, ByteBuffer)]
}

```

```

    /** Applies snapshot to the state machine
     * @param index new last applied index
     * @param bytes data to restore from
     */
    def restoreSnapshot(index: Long, bytes: ByteBuffer): F[Unit]
}

```

StateStorage.scala

```

trait StateStorage[F[_]] {
  def persistState(state: PersistedState): F[Unit]
  def retrieveState(): F[Option[PersistedState]]
}

```

Storage.scala

```

case class Storage[F[_]](logStorage: LogStorage[F],
                         stateStorage: StateStorage[F],
                         snapshotStorage: SnapshotStorage[F])
)

```

MemoryLogStorage.scala

```

class MemoryLogStorage[F[_]: Sync](itemsRef: Ref[F, Map[Long, LogEntry]],
                                    lastIndexRef: Ref[F, Long])
  extends LogStorage[F] {

  override def lastIndex: F[Long] =
    lastIndexRef.get

  override def get(index: Long): F[Option[LogEntry]] =
    itemsRef.get.map(_.get(index))

  override def put(index: Long, logEntry: LogEntry): F[LogEntry] =
    itemsRef.update(_ + (index -> logEntry)) >>
    lastIndexRef.update(i => Math.max(i, index)) as logEntry

  override def deleteBefore(index: Long): F[Unit] =
    itemsRef.update(items => items.filter(_.l_1 >= index))

  override def deleteAfter(index: Long): F[Unit] =
    itemsRef.update(items => items.filter(_.l_1 <= index))

}

object MemoryLogStorage {
  def empty[F[_]: Sync]: F[MemoryLogStorage[F]] =
    for {
      items <- Ref.of[F, Map[Long, LogEntry]](Map.empty)
      last  <- Ref.of[F, Long](0L)
    } yield new MemoryLogStorage[F](items, last)
}

```

MemorySnapshotStorage.scala

```

class MemorySnapshotStorage[F[_]: Monad](ref: Ref[F, Option[Snapshot]]) extends
SnapshotStorage[F] {

  override def saveSnapshot(snapshot: Snapshot): F[Unit] = ref.set(Some(snapshot))

  override def retrieveSnapshot: F[Option[Snapshot]] = ref.get
}

```

```

    override def getLatestSnapshot: F[Option[Snapshot]] = ref.get
}

object MemorySnapshotStorage {
  def empty[F[_]: Concurrent]: F[MemorySnapshotStorage[F]] =
    Ref.of[F, Option[Snapshot]](None).map(new MemorySnapshotStorage[F](_))
}

```

MemoryStateStorage.scala

```

class MemoryStateStorage[F[_]: Monad](ref: Ref[F, Option[PersistedState]]) extends
StateStorage[F] {

  override def persistState(state: PersistedState): F[Unit] =
    ref.set(Some(state))

  override def retrieveState(): F[Option[PersistedState]] =
    ref.get

}

object MemoryStateStorage {
  def empty[F[_]: Concurrent]: F[MemoryStateStorage[F]] =
    Ref
      .of[F, Option[PersistedState]](Some(PersistedState(0, None, 0L)))
      .map(new MemoryStateStorage[F](_))
}

```

MemoryStorage.scala

```

object MemoryStorage {
  def empty[F[_]: Async]: F[Storage[F]] =
    for {
      snapshotStorage <- MemorySnapshotStorage.empty[F]
      stateStorage     <- MemoryStateStorage.empty[F]
      logStorage       <- MemoryLogStorage.empty[F]
    } yield Storage[F](logStorage, stateStorage, snapshotStorage)
}

```

ConsoleLogger.scala

```

case class ConsoleLogger[F[_]: Console: Monad: Clock](name: String,
                                                       loggingLevel: ConsoleLogger.Level)
  extends Logger[F] {
  import ConsoleLogger.Level._

  private def procLevel(opLevel: ConsoleLogger.Level)(op: => F[Unit]): F[Unit] =
    if (opLevel.value > loggingLevel.value) Monad[F].unit else op

  private def logMsg(
    opLevel: ConsoleLogger.Level
  )(msg: => String, err: Option[Throwable], ctx: Option[Map[String, String]]): F[Unit] =
    procLevel(opLevel)(
      Clock[F].realTime.flatMap(time =>
        opLevel match {
          case Level.Error => Console[F].errorln[String](_)
          case _           => Console[F].println[String](_)
        }).apply(
          s"${LocalDateTime.ofEpochSecond(time.toSeconds, 0, ZoneOffset.UTC).toLocalTime}"
        ($opLevel) $name - $msg" + err
          .fold("")(e => s" ${e.getMessage}") + ctx
          .fold("")(_.mkString("\n Context:\n", ",\n", ""))
        ) >>
        err.fold(() .pure[F])(Console[F].printStackTrace)
      )
    )
}

```

```

override def trace(msg: => String) = logMsg(Trace)(msg, None, None)
override def trace(msg: => String, e: Throwable) = logMsg(Trace)(msg, Some(e), None)
override def trace(msg: => String, ctx: Map[String, String]): F[Unit] =
  logMsg(Trace)(msg, None, Some(ctx))
override def trace(msg: => String, ctx: Map[String, String], e: Throwable): F[Unit] =
  logMsg(Trace)(msg, Some(e), Some(ctx))
override def debug(msg: => String) = logMsg(Debug)(msg, None, None)
override def debug(msg: => String, e: Throwable) = logMsg(Debug)(msg, Some(e), None)
override def debug(msg: => String, ctx: Map[String, String]): F[Unit] =
  logMsg(Debug)(msg, None, Some(ctx))
override def debug(msg: => String, ctx: Map[String, String], e: Throwable): F[Unit] =
  logMsg(Debug)(msg, Some(e), Some(ctx))
override def info(msg: => String) = logMsg(Info)(msg, None, None)
override def info(msg: => String, e: Throwable) = logMsg(Info)(msg, Some(e), None)
override def info(msg: => String, ctx: Map[String, String]): F[Unit] =
  logMsg(Info)(msg, None, Some(ctx))
override def info(msg: => String, ctx: Map[String, String], e: Throwable): F[Unit] =
  logMsg(Info)(msg, Some(e), Some(ctx))
override def warn(msg: => String) = logMsg(Warn)(msg, None, None)
override def warn(msg: => String, e: Throwable) = logMsg(Warn)(msg, Some(e), None)
override def warn(msg: => String, ctx: Map[String, String]): F[Unit] =
  logMsg(Warn)(msg, None, Some(ctx))
override def warn(msg: => String, ctx: Map[String, String], e: Throwable): F[Unit] =
  logMsg(Warn)(msg, Some(e), Some(ctx))
override def error(msg: => String) = logMsg(Error)(msg, None, None)
override def error(msg: => String, e: Throwable) = logMsg(Error)(msg, Some(e), None)
override def error(msg: => String, ctx: Map[String, String]): F[Unit] =
  logMsg(Error)(msg, None, Some(ctx))
override def error(msg: => String, ctx: Map[String, String], e: Throwable): F[Unit] =
  logMsg(Error)(msg, Some(e), Some(ctx))
}

object ConsoleLogger {
  sealed abstract class Level(val value: Byte) extends ByteEnumEntry
  object Level extends ByteEnum[Level] {
    case object Trace extends Level(5)
    case object Debug extends Level(4)
    case object Info extends Level(3)
    case object Warn extends Level(2)
    case object Error extends Level(1)

    override def values: IndexedSeq[Level] = findValues
  }
}

```

ПРИЛОЖЕНИЕ В (Модификация Мульти-RAFT)

Структура модуля модификации Мульти-RAFT:

```
multiraf
├── ClusterStateMachine.scala
│   └── MultiRaft.scala
└── rocks
    └── RocksKVStateMachine.scala
        └── package.scala
└── util
    └── Zipping.scala
```

ClusterStateMachine.scala

```
case class MaxLeaderCommand(membersAllowance: List[(Node, Int)]) extends WriteCommand[Unit]
case class InitClusterCommand(newRaftId: Int, remainder: Int) extends WriteCommand[Unit]
case class ConfirmRaftCommand(raftId: Int) extends WriteCommand[Unit]

case class IsConfirmedCommand(raftId: Int) extends ReadCommand[Boolean]
case class MaxLeadersForCommand(node: Node) extends ReadCommand[Int]
case class RemainderForGroupCommand(raftId: Int) extends ReadCommand[Int]

case class RaftState(remainder: Int, confirmed: Boolean)

class ClusterStateMachine[F[_]: Sync](lastIndex: Ref[F, Long],
                                      val raftCounter: Ref[F, Int],
                                      val maxLeaders: Ref[F, Map[Node, Int]],
                                      val clusterState: Ref[F, Map[Int, RaftState]],
                                      val clusterQueue: Queue[F, Int])
    extends StateMachine[F] {
    override def applyWrite: WriteHandler = {
        case (index, MaxLeaderCommand(allowance)) =>
            maxLeaders.set(allowance.toMap) >> lastIndex.set(index)
        case (index, InitClusterCommand(newRaftId, remainder)) =>
            clusterState.update(
                .updated(newRaftId, RaftState(remainder, confirmed = false))
            ) >> clusterQueue.offer(newRaftId) >> lastIndex.set(index)
        case (index, ConfirmRaftCommand(raftId)) =>
            clusterState.update(_.updatedWith(raftId)(_.map(_.copy(confirmed = true)))) >>
                lastIndex.set(index)
    }
}

override def applyRead: ReadHandler = {
    case IsConfirmedCommand(raftId) =>
        clusterState.get.map(_.get(raftId).exists(_.confirmed))
    case MaxLeadersForCommand(node) =>
        maxLeaders.get.map(_.getOrElse(node, 0))
    case RemainderForGroupCommand(raftId) =>
        clusterState.get.map(_.get(raftId).map(_.remainder).getOrElse(-1))
}
```

override def appliedIndex: F[Long] = lastIndex.get

override def takeSnapshot: F[(Long, ByteBuffer)] =
for {
 state <- clusterState.get
 leaders <- maxLeaders.get
 counter <- raftCounter.get
 index <- lastIndex.get
 bytes = serialize((state, leaders, counter))
} yield (index, bytes)

override def restoreSnapshot(index: Long, bytes: ByteBuffer): F[Unit] =
Sync[F].delay(deserialize(bytes)).flatMap { case (state, leaders, counter) =>
 lastIndex.set(index) >>
 clusterState.set(state) >>
 maxLeaders.set(leaders) >>
 raftCounter.set(counter)

```

}

def nextId: F[Int] =
  raftCounter.getAndUpdate(_ + 1)

type SerType = (Map[Int, RaftState], Map[Node, Int], Int)

private def serialize(items: SerType): ByteBuffer = {
  val stream: ByteArrayOutputStream = new ByteArrayOutputStream()
  val oos = new ObjectOutputStream(stream)
  oos.writeObject(items)
  oos.close()

  ByteBuffer.wrap(stream.toByteArray)
}

private def deserialize(bytes: ByteBuffer): SerType = {
  val ois = new ObjectInputStream(new ByteArrayInputStream(bytes.array()))
  val response = ois.readObject().asInstanceOf[SerType]
  ois.close()

  response
}

object ClusterStateMachine {
  def empty[F[_]: Async]: F[ClusterStateMachine[F]] = for {
    index <- Ref.of(0L)
    counter <- Ref.of(0)
    leaders <- Ref.of(Map.empty[Node, Int])
    state <- Ref.of(Map.empty[Int, RaftState])
    queue <- Queue.unbounded[F, Int]
  } yield new ClusterStateMachine[F](index, counter, leaders, state, queue)
}

```

MultiRaft.scala

```

class MultiRaft[F[_]: Async: Logger](val raft: Raft[F, ClusterStateMachine],
                                      val rpc: RpcServer[F],
                                      curNodeLimit: Semaphore[F],
                                      subConfigTemplate: Configuration,
                                      subLogStorage: => F[LogStorage[F]],
                                      subSnapshotStorage: => F[SnapshotStorage[F]],
                                      subStateStorage: => F[StateStorage[F]],
                                      subStateMachine: => F[StateMachine[F]])
  extends Cluster[F, ClusterStateMachine] {
  private val mainCluster = new SingleRaftCluster[F, ClusterStateMachine](rpc, raft)
  private val procFiberRef = Ref.unsafe(Option.empty[Fiber[F, Throwables, Unit]])

  val innerClusters: Ref[F, Map[Int, SingleRaftCluster[F, StateMachine]]] =
    Ref.unsafe(Map.empty)

  private def routeByHash(bytes: ByteVector): F[SingleRaftCluster[F, StateMachine]] =
    innerClusters.get.map(remsToClusters =>
      remsToClusters.apply(bytes.hashCode % remsToClusters.size)
    )

  override def start: F[Node] =
    for {
      leader <- mainCluster.start
      allMembers <- mainCluster.raft.membershipManager.members.map(_.toList)
      limit <- curNodeLimit.available.map(_.toInt)
      _ <- mainCluster.raft.getCurrentState match {
        case _: LeaderNode =>
          mainCluster.execute(MaxLeaderCommand(allMembers.map(_ -> limit))) >>
            allMembers.zipWithIndex.traverse { case (_, remainder) =>
              mainCluster.raft.log.stateMachine.nextId.flatMap(nextRaftId =>
                mainCluster.execute(InitClusterCommand(nextRaftId, remainder))
              )
            }
      }
    }

```

```

procFiber <-
  fs2.Stream
    .fromQueueUnterminated(mainCluster.raft.log.stateMachine.clusterQueue)
    .evalMap(requestedId => makeSubCluster(requestedId))
    .evalTap(_.start)
    .evalTap(cluster => mainCluster.execute(ConfirmRaftCommand(cluster.raft.raftId)))
    .evalTap(cluster =>
      mainCluster.raft.log.stateMachine.clusterState.get
        .map(_.apply(cluster.raft.raftId))
        .flatMap(remainder => innerClusters.update(_.updated(remainder.remainder,
      cluster)))
    )
    .compile
    .drain
    .start
  _ <- procFiberRef.set(Some(procFiber))
} yield leader

private def makeSubCluster(raftId: Int): F[SingleRaftCluster[F, StateMachine]] =
  for {
    logStorage      <- subLogStorage
    snapshotStorage <- subSnapshotStorage
    stateStorage    <- subStateStorage
    stateMachine    <- subStateMachine
    storage         = Storage(logStorage, stateStorage, snapshotStorage)
    members         <- mainCluster.raft.membershipManager.members
    curNode         = mainCluster.raft.nodeId
    config          =
      subConfigTemplate.copy(local = curNode, members = members.filter(_ != curNode).toList)
    membershipManagement = mainCluster.raft.membershipManager
    clientProvider   = mainCluster.raft.clientProvider
    persistedState   <- stateStorage.retrieveState()
    appliedIndex     = persistedState.map(_.appliedIndex).getOrElse(0L)
    log <- LogImpl.build(
      raftId,
      logStorage,
      snapshotStorage,
      stateMachine,
      LogCompactionPolicy.fixedSize(100),
      membershipManagement,
      appliedIndex
    )
    announcer <- LeaderAnnouncerImpl.build[F]
    replicator <- LogPropagatorImpl.build[F](config.local, clientProvider, log)
    heartbeat <- Ref.of[F, Long](0L)
    nodeState <- Ref.of[F, NodeState](
      persistedState
        .map(_.toNodeState(raftId, config.local))
        .getOrElse(FollowerNode(raftId, config.local, 0L))
    )
    running <- Ref.of[F, Boolean](false)
  } yield new SingleRaftCluster[F, StateMachine]{
    rpc,
    new RaftImpl[F, StateMachine]{
      raftId,
      Some(curNodeLimit),
      config,
      membershipManagement,
      clientProvider,
      announcer,
      replicator,
      log,
      storage,
      nodeState,
      heartbeat,
      running
    }
  }

override def stop: F[Unit] = mainCluster.stop

override def join(node: Node): F[Node] =
  mainCluster
    .join(node)

```

```

    .flatTap(_ => innerClusters.get.flatMap(_.toList.traverse(_.2.join(node))))}

override def leave: F[Unit] = mainCluster.leave

override def leader: F[Node] = mainCluster.leader

private def routeMany[T](many: List[T],
                        key: T => ByteVector
  ): F[List[(SingleRaftCluster[F], StateMachine), List[T]]] =
  many
    .traverse(value => routeByHash(key(value)).map(_ -> value))
    .map(_.groupBy(_.1.raft.raftId))
    .map(grouped =>
      grouped.view.map { case (_, list) =>
        list.head._1 -> list.map(_.2)
      }
    )
    .map(_.toList)

override def execute[T](command: Command[T]): F[T] =
  command match {
    case rc: RocksCommand =>
      rc match {
        case c @ WriteSingleCommand(key, _) => routeByHash(key).flatMap(_.execute(c))
        case c @ DeleteSingleCommand(key) => routeByHash(key).flatMap(_.execute(c))
        case c @ ReadSingleCommand(key) => routeByHash(key).flatMap(_.execute(c))
        case WriteManyCommand(many) =>
          routeMany[ByteVector, ByteVector](many, _.1)
            .flatMap(_.traverse { case (cluster, data) =>
              cluster.execute[Unit](WriteManyCommand(data))
            })
            .void
            .asInstanceOf[F[T]]
        case DeleteManyCommand(many) =>
          routeMany[ByteVector](many, identity)
            .flatMap(_.traverse { case (cluster, data) =>
              cluster.execute[Unit](DeleteManyCommand(data))
            })
            .void
            .asInstanceOf[F[T]]
        case ReadManyCommand(many) =>
          routeMany[ByteVector](many, identity)
            .flatMap(_.flatTraverse { case (cluster, data) =>
              cluster.execute[List[Option[ByteVector]]](ReadManyCommand(data))
            })
            .asInstanceOf[F[T]]
      }
    case otherCommand => mainCluster.execute(otherCommand)
  }
}

object MultiRaft {
  def apply[F[_]: Async: Console](name: String,
                                    config: Configuration,
                                    subConfig: Configuration,
                                    rocksPath: String
  )(implicit
    commandSer: ProtoSerializer[Command[_]],
    configSer: ProtoSerializer[ClusterConfiguration],
    objectSer: ProtoSerializer[Any]
  ): Resource[F, MultiRaft[F]] =
    implicit val logger: Logger[F] = ConsoleLogger[F](name, ConsoleLogger.Level.Info)

  implicit val clientBuilder: GrpcClientBuilder[F] = GrpcClientBuilder[F]
  implicit val serverBuilder: GrpcServerBuilder[F] = GrpcServerBuilder[F]

  for {
    rocksOptions <- nativeResource(Async[F].delay(new Options().setCreateIfMissing(true)))
    rocksEngine <-
      Resource eval RocksEngine[F](RocksEngine.initDB(rocksPath, rocksOptions, ttl = false))
    multiRaft = for {
      stateMachine <- ClusterStateMachine.empty[F]
      raftId       <- stateMachine.nextId
      storage       <- MemoryStorage.empty[F]
    }
  }
}

```

```

compaction = if (config.logCompactionThreshold >= 0)
    LogCompactionPolicy.fixedSize(config.logCompactionThreshold)
  else LogCompactionPolicy.noCompaction
mainRaft <-
    RaftImpl.build[F, ClusterStateMachine](
        raftId,
        config,
        storage,
        stateMachine,
        compaction
    )
initMap: Map[Int, Raft[F, StateMachine]] = Map(mainRaft.raftId -> mainRaft)
    <- Ref.of(initMap)
    server
    limitSemaphore
    <- Semaphore[F] (1)
} yield new MultiRaft[F](
    mainRaft,
    server,
    limitSemaphore,
    subConfig,
    MemoryLogStorage.empty[F],
    MemorySnapshotStorage.empty[F],
    MemoryStateStorage.empty[F],
    Async[F].delay(RocksKVStateMachine[F](rocksEngine))
)
cluster <- Resource.make(multiRaft) (_.stop)
} yield cluster
}
}
}

```

RocksKVStateMachine.scala

```

class RocksKVStateMachine[F[_]: Async] (engine: RocksEngine[F]) extends StateMachine[F] {
  import RocksKVStateMachine._

  implicit def byteVecEncoder: Encoder[F, ByteVector] = (data: ByteVector) =>
    Async[F].delay(data.toArray)
  implicit def byteVecDecoder: Decoder[F, ByteVector] = (data: Array[Byte]) =>
    Async[F].delay(Option(ByteVector(data)))

  override def applyWrite: WriteHandler = {
    case (index, WriteSingleCommand(key, value)) => engine.put(key, value) >> setIndex(index)
    case (index, WriteManyCommand(many)) => engine.batchPut(many) >> setIndex(index)
    case (index, DeleteSingleCommand(key)) => engine.delete(key) >> setIndex(index)
    case (index, DeleteManyCommand(keys)) =>
      keys.traverse(engine.delete[ByteVector]) >> setIndex(index)
  }: WriteHandler

  override def applyRead: ReadHandler = {
    case ReadSingleCommand(key) => engine.get[ByteVector, ByteVector](key)
    case ReadManyCommand(keys) => engine.batchGet[ByteVector, ByteVector](keys)
  }: ReadHandler

  override def appliedIndex: F[Long] =
    engine.get[String, Long](appliedIndexKey).map(_.getOrElse(0))

  private def setIndex(index: Long): F[Unit] =
    engine.put(appliedIndexKey, index)

  override def takeSnapshot: F[(Long, ByteBuffer)] =
    for {
      <- engine.dbGenStop
      tmpDir <- Resource.make(Files[F].createTempDirectory)(Files[F].deleteRecursively)
      <- engine.checkpoint(tmpDir.toString)
      tmpZip <- Resource.make(Files[F].createTempFile)(Files[F].delete)
      <- Resource eval Zipping.convertToZip(tmpDir.toNioPath.toFile,
tmpZip.toNioPath.toFile)
    } yield tmpZip.use(zip =>
    for {
      zipSize <- Files[F].size(zip).map(_.toInt)
      formedBB <-
    }
  }
}

```

```

    Files[F]
      .readAll(zip)
      .compile
      .foldChunks(ByteBuffer.allocate(zipSize))((bb, chunk) =>
bb.put(chunk.toByteBuffer))
      index <- appliedIndex
    } yield index -> formedBB
  }

override def restoreSnapshot(index: Long, bytes: ByteBuffer): F[Unit] =
  for {
    _ <- engine.dbGenStop
    tmpZip <- Resource.make(Files[F].createTempFile)(Files[F].delete)
    _ <- Resource.eval(fs2.Stream
      .chunk(fs2.Chunk.ByteBuffer(bytes))
      .chunkLimit(1 << 16)
      .flatMap(fs2.Stream.chunk)
      .through(Files[F].writeAll(tmpZip))
      .compile
      .drain
    tmpDir = FPath(s"$tmp/rocksKV${System.currentTimeMillis()}")
    _ <- Resource.eval(Files[F].createDirectory(tmpDir))
    _ <- Resource.eval(Zipping.convertFromZip(tmpZip.toNioPath.toFile,
tmpDir.toNioPath.toFile))
    options <- nativeResource(Async[F].delay(new Options().setCreateIfMissing(false)))
  } yield tmpDir -> options.use { case (dir, opts) =>
  engine.hotSwap(RocksEngine.openDB(dir.toString, opts, ttl = false))
}
}

object RocksKVStateMachine {
  val appliedIndexKey: String = "__applied_index__"

  def apply[F[_]: Async](engine: RocksEngine[F]): RocksKVStateMachine[F] =
    new RocksKVStateMachine(engine)
}

```

rocks/package.scala

```

package object rocks {
  sealed trait RocksCommand
  object RocksCommand {
    val prefix = 0x10
  }

  case class WriteSingleCommand(key: ByteVector, value: ByteVector)
    extends WriteCommand[Unit] with RocksCommand
  case class WriteManyCommand(many: List[(ByteVector, ByteVector)])
    extends WriteCommand[Unit] with RocksCommand
  case class DeleteSingleCommand(key: ByteVector)
    extends WriteCommand[Unit] with RocksCommand
  case class DeleteManyCommand(keys: List[ByteVector])
    extends WriteCommand[Unit] with RocksCommand
  case class ReadSingleCommand(key: ByteVector)
    extends ReadCommand[Option[ByteVector]] with RocksCommand
  case class ReadManyCommand(keys: List[ByteVector])
    extends ReadCommand[List[Option[ByteVector]]] with RocksCommand

  object WriteSingleCommand {
    implicit def discriminator: Discriminator[Command[_, WriteSingleCommand], Int] =
      Discriminator(RocksCommand.prefix & 1)
  }
  object WriteManyCommand {
    implicit def discriminator: Discriminator[Command[_, WriteManyCommand], Int] =
      Discriminator(RocksCommand.prefix & 2)
  }
  object DeleteSingleCommand {
    implicit def discriminator: Discriminator[Command[_, DeleteSingleCommand], Int] =
      Discriminator(RocksCommand.prefix & 3)
  }
  object DeleteManyCommand {
    implicit def discriminator: Discriminator[Command[_, DeleteManyCommand], Int] =

```

```

        Discriminator(RocksCommand.prefix & 4)
    }
    object ReadSingleCommand {
      implicit def discriminator: Discriminator[Command[_], ReadSingleCommand, Int] =
        Discriminator(RocksCommand.prefix & 5)
    }
    object ReadManyCommand {
      implicit def discriminator: Discriminator[Command[_], ReadManyCommand, Int] =
        Discriminator(RocksCommand.prefix & 6)
    }
}

```

Zipping.scala

```

object Zipping {

  def convertToZip[F[_]: Async](fileToZip: File, outFile: File): F[File] =
    Resource
      .fromAutoCloseable(Sync[F].delay(new FileOutputStream(outFile.getAbsolutePath)))
      .flatMap(fos => Resource.fromAutoCloseable(Sync[F].delay(new ZipOutputStream(fos))))
      .use(zipFile(fileToZip, fileToZip.getName, _))
      .as(outFile)

  private def zipFile[F[_]: Async](fileToZip: File,
                                    fileName: String,
                                    zipOut: ZipOutputStream
                                  ): F[Unit] =
    fileToZip match {
      case f if f.isHidden => Async[F].unit
      case f if f.isDirectory =>
        Sync[F].blocking(if (fileName.endsWith("/")) {
          zipOut.putNextEntry(new ZipEntry(fileName))
          zipOut.closeEntry()
        } else {
          zipOut.putNextEntry(new ZipEntry(fileName + "/"))
          zipOut.closeEntry()
        }) >> fileToZip
          .listFiles()
          .toList
          .traverse(child => zipFile(child, fileName + "/" + child.getName, zipOut))
          .void
      case _ =>
        Sync[F].delay(zipOut.putNextEntry(new ZipEntry(fileName))) >>
        Files[F]
          .readAll(FPath(fileToZip.getAbsolutePath))
          .chunks
          .evalMap(chunk => Sync[F].blocking(zipOut.write(chunk.toArray())))
          .compile
          .drain
    }

  def convertFromZip[F[_]: Async](zippedFile: File, outFile: File): F[File] =
    Resource
      .make(Async[F].delay(new ZipInputStream(new FileInputStream(zippedFile))))(zis =>
        Async[F].delay(zis.closeEntry()) >> Async[F].delay(zis.close())
      )
      .use(zis =>
        fs2.Stream
          .repeatEval(Async[F].delay(zis.getNextEntry()))
          .map(Option.apply)
          .unNoneTerminate
          .map(entry => entry -> newFile(outFile, entry))
          .evalMap {
            case (dirEntry, Some(curFile)) if dirEntry.isDirectory =>
              Async[F].blocking {
                if (!curFile.isDirectory && !curFile.mkdirs())
                  throw new IOException(s"Failed to unzip dir entry: $curFile")
              }
            case (_, Some(curFile)) =>
              Async[F]
                .delay(curFile.getParentFile)
                .flatMap(parent =>

```

```

    Async[F].blocking {
      if (!parent.isDirectory && !curFile.getParentFile.mkdirs())
        throw new IOException(s"Failed to unzip file entry: $curFile")
    }
  ) >>
  fs2.io
    .readInputStream(Async[F].pure(zis: InputStream), 1 << 10, closeAfterUse =
  false)
    .through(Files[F].writeAll(FPath(curFile.getAbsolutePath())))
    .compile
    .drain
  }
  .compile
  .drain
)
.as(outFile)

private def newFile(destinationDir: File, zipEntry: ZipEntry): Option[File] =
  Option(new File(destinationDir, zipEntry.getName))
    .filter(_.getCanonicalPath.startsWith(destinationDir.getCanonicalPath + File.separator))
}

```

ПРИЛОЖЕНИЕ Г (Scala API для разработанного хранилища)

Структура модуля Scala API:

```
└── client
    ├── DKV4SClient.scala
    └── DKV4SClientImpl.scala
```

DKV4SClient.scala

```
trait DKV4SClient[F[_]] {

    def read[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]]
    def readMany[K: Encoder[F, *], V: Decoder[F, *]](keys: List[K]): F[List[Option[V]]]

    def write[K: Encoder[F, *], V: Encoder[F, *]](key: K, value: V): F[Unit]
    def writeMany[K: Encoder[F, *], V: Encoder[F, *]](values: List[(K, V))]: F[Unit]

    def delete[K: Encoder[F, *]](key: K): F[Unit]
    def deleteMany[K: Encoder[F, *]](keys: List[K]): F[Unit]

    def consistentRead[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]]
    def consistentReadMany[K: Encoder[F, *], V: Decoder[F, *]](keys: List[K]): F[List[Option[V]]]

    def consistentWrite[K: Encoder[F, *], V: Encoder[F, *]](key: K, value: V): F[Unit]
    def consistentWriteMany[K: Encoder[F, *], V: Encoder[F, *]](values: List[(K, V))]: F[Unit]

    def consistentDelete[K: Encoder[F, *]](key: K): F[Unit]
    def consistentDeleteMany[K: Encoder[F, *]](keys: List[K]): F[Unit]
}
```

DKV4SClientImpl.scala

```
class DKV4SClientImpl[F[_]: Async](directEngine: RocksEngine[F],
                                    multiRaft: MultiRaft[F],
                                    changeQueue: Queue[F, WriteCommand[_]])
  extends DKV4SClient[F] {
    private def prepWrite[K: Encoder[F, *], V: Encoder[F, *]](key: K,
                                                               value: V
                                                               ): F[(ByteVector, ByteVector)] =
      Encoder[F, K].encode(key).map(ByteVector(_)) product Encoder[F, V]
        .encode(value)
        .map(ByteVector(_))

    private def prepWriteMany[K: Encoder[F, *], V: Encoder[F, *]](
      values: List[(K, V))
      ): F[List[(ByteVector, ByteVector))] =
      values.traverse { case (key, value) =>
        Encoder[F, K].encode(key).map(ByteVector(_)) product Encoder[F, V]
          .encode(value)
          .map(ByteVector(_))
    }

    override def read[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]] =
      directEngine.get[K, V](key)

    override def readMany[K: Encoder[F, *], V: Decoder[F, *]](keys: List[K]): F[List[Option[V]]] =
      directEngine.batchGet[K, V](keys)

    override def write[K: Encoder[F, *], V: Encoder[F, *]](key: K, value: V): F[Unit] =
      prepWrite(key, value).flatMap { case (bkey, bvalue) =>
        changeQueue.offer(WriteSingleCommand(bkey, bvalue))
      } >> directEngine.put(key, value)

    override def writeMany[K: Encoder[F, *], V: Encoder[F, *]](values: List[(K, V))): F[Unit] =
      prepWriteMany(values)
        .flatMap(bvalues => changeQueue.offer(WriteManyCommand(bvalues))) >>
```

```

    directEngine.batchPut(values)

override def delete[K: Encoder[F, *]](key: K): F[Unit] =
  Encoder[F, K]
    .encode(key)
    .flatMap(bkey => changeQueue.offer(DeleteSingleCommand(ByteVector(bkey)))) >>
  directEngine.delete(key)

override def deleteMany[K: Encoder[F, *]](keys: List[K]): F[Unit] =
  keys
    .traverse(Encoder[F, K].encode(_).map(ByteVector(_)))
    .flatMap(bkeys => changeQueue.offer(DeleteManyCommand(bkeys))) >>
  keys.traverse(directEngine.delete[K]).void

override def consistentRead[K: Encoder[F, *], V: Decoder[F, *]](key: K): F[Option[V]] =
  Encoder[F, K]
    .encode(key)
    .flatMap(bkey => multiRaft.execute(ReadSingleCommand(ByteVector(bkey))))
    .flatMap {
      case Some(value) => Decoder[F, V].decode(value.toArray)
      case None       => None.pure[F]
    }

override def consistentReadMany[K: Encoder[F, *], V: Decoder[F, *]](
  keys: List[K]
): F[List[Option[V]]] =
  keys
    .traverse(Encoder[F, K].encode)
    .map(_.map(ByteVector(_)))
    .flatMap(bkeys => multiRaft.execute(ReadManyCommand(bkeys)))
    .flatMap(_.traverse {
      case Some(value) => Decoder[F, V].decode(value.toArray)
      case None       => None.pure[F]
    })

override def consistentWrite[K: Encoder[F, *], V: Encoder[F, *]](key: K, value: V): F[Unit] =
  prepWrite(key, value).flatMap { case (bkey, bvalue) =>
    multiRaft.execute(WriteSingleCommand(bkey, bvalue))
  }

override def consistentWriteMany[K: Encoder[F, *], V: Encoder[F, *]](
  values: List[(K, V)]
): F[Unit] =
  prepWriteMany(values).flatMap(bvalues => multiRaft.execute(WriteManyCommand(bvalues)))

override def consistentDelete[K: Encoder[F, *]](key: K): F[Unit] =
  Encoder[F, K]
    .encode(key)
    .flatMap(bkey => multiRaft.execute(DeleteSingleCommand(ByteVector(bkey))))

override def consistentDeleteMany[K: Encoder[F, *]](keys: List[K]): F[Unit] =
  keys
    .traverse(Encoder[F, K].encode(_).map(ByteVector(_)))
    .flatMap(bkeys => multiRaft.execute(DeleteManyCommand(bkeys)))
}

object DVK4SClientImpl {
  private val maxGroup = 1e+5.toInt

  def clientResource[F[_]: Async](directEngine: RocksEngine[F],
                                 multiRaft: MultiRaft[F])
  ): Resource[F, DVK4SClientImpl[F]] =
    Resource
      .make(
        Queue
          .unbounded[F, WriteCommand[_]]
          .mproduct(queue =>
            Stream
              .fromQueueUnterminated(queue, maxGroup)
              .groupAdjacentByLimit(maxGroup) (_ .getClass.getSimpleName)
              .evalMap {
                case (_, reads: Chunk[ReadSingleCommand]) =>
                  Async[F]
                    .delay(

```

```

        ReadManyCommand(
            reads
                .foldLeft(List.newBuilder[ByteVector]) {
                    case (acc, ReadSingleCommand(key)) =>
                        acc.addOne(key)
                }
                .result()
            )
        )
        .flatMap(multiRaft.execute)
    case (_, writes: Chunk[WriteSingleCommand]) =>
        Async[F]
        .delay(
            WriteManyCommand(
                writes
                    .foldLeft(List.newBuilder[(ByteVector, ByteVector)]) {
                        case (acc, WriteSingleCommand(key, value)) => acc.addOne(key -> value)
                    }
                    .result()
            )
        )
        .flatMap(multiRaft.execute)
    case (_, deletes: Chunk[DeleteSingleCommand]) =>
        Async[F]
        .delay(
            DeleteManyCommand(
                deletes
                    .foldLeft(List.newBuilder[ByteVector]) {
                        case (acc, DeleteSingleCommand(key)) =>
                            acc.addOne(key)
                    }
                    .result()
            )
        )
        .flatMap(multiRaft.execute)
    case (_, batchCommand) => batchCommand.traverse(multiRaft.execute(_)).void
}
.compile
.drain
.start
)
) { case (_, fiber) =>
    fiber.cancel
}
.map { case (queue, _) =>
    new DVK4SClientImpl[F](directEngine, multiRaft, queue)
}
}
}

```