СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире обработки данных отказоустойчивость и надежность распределенных систем являются ключевыми требованиями для множества приложений, начиная от облачных сервисов и заканчивая базами данных и системами хранения информации. Одним из фундаментальных механизмов, обеспечивающих согласованность данных в распределенных системах, является алгоритм консенсуса. Среди различных алгоритмов консенсуса, таких как Paxos, Viewstamped Replication и другие, алгоритм Raft выделяется своей простотой понимания, формальной доказанностью корректности и эффективностью работы.

Raft представляет собой алгоритм консенсуса, предназначенный для управления распределенными системами и поддержания согласованности реплицированных логов. Он был предложен Диего Онгаро и Джоном Оустером в 2014 году как более понятная альтернатива Paxos, что делает его привлекательным для реализации в практических системах.

В рамках данной научно-исследовательской работы рассматривается задача создания отказоустойчивого распределенного хранилища данных на основе алгоритма Raft. В первой части работы проводится анализ существующих реализаций Raft, обосновывается выбор одной из библиотек для репликации данных, а также разрабатывается архитектура хранилища с учетом интеграции данной библиотеки. Кроме того, рассматриваются механизмы взаимодействия узлов для достижения консенсуса и обеспечения надежности системы.

Во второй части работы будет осуществлена реализация основных операций хранилища, включая механизм репликации данных, а также проведено тестирование системы на предмет отказоустойчивости.

Таким образом, цель данной работы заключается в проектировании и разработке распределенного хранилища данных, использующего алгоритм Raft для обеспечения согласованности и высокой доступности данных.

ОСНОВНАЯ ЧАСТЬ

Перед разработкой распределенного хранилища данных на основе алгоритма Raft необходимо провести анализ существующих реализаций данного алгоритма. В настоящее время существует множество библиотек и фреймворков, реализующих Raft, каждая из которых обладает своими преимуществами и особенностями. Выбор подходящей библиотеки играет ключевую роль в построении отказоустойчивой системы, поскольку от него зависит производительность, масштабируемость и надежность репликации данных.

Анализ существующих решений позволит выявить их сильные и слабые стороны, определить критерии выбора наиболее подходящей библиотеки, а также учесть возможные ограничения и особенности интеграции в разрабатываемую систему. В данной части работы будут рассмотрены наиболее популярные и широко используемые реализации Raft, их архитектура, функциональные возможности и сферы применения. На основе проведенного анализа будет обоснован выбор конкретной библиотеки для дальнейшего использования в проекте.

1 Анализ существующих реализаций алгоритма

Для анализа были использованы данные с официального ресурса Raft GitHub, который содержит список наиболее популярных и активно поддерживаемых реализаций алгоритма на разных языках программирования. Среди них можно выделить:

1.1 etcd/raft

Библиотека Raft реализует протокол консенсуса Raft для поддержания согласованного состояния распределённого конечного автомата через реплицируемый журнал. Она используется в отказоустойчивых системах, таких как etcd, Kubernetes и CockroachDB.

В отличие от монолитных реализаций, библиотека обеспечивает только алгоритм консенсуса, оставляя сетевое взаимодействие и хранение данных на усмотрение пользователя. Это повышает её гибкость, детерминированность и производительность.

Функционал включает выборы лидера, репликацию журнала, изменения состава кластера и оптимизированные запросы на чтение. Raft представлен в виде детерминированного конечного автомата, что упрощает моделирование и тестирование.

1.2 HashiCorp Raft

HashiCorp Raft — это мощная библиотека на Go, реализующая алгоритм консенсуса Raft. Она предназначена для управления реплицируемыми журналами и согласованными конечными автоматами, что делает её ключевым инструментом для построения распределённых систем с высокой отказоустойчивостью и согласованностью (CP по CAP-теореме).

Raft использует три роли узлов — лидер, кандидат и последователь, — обеспечивая надёжный механизм выбора лидера и согласованной репликации данных. Журнал логов может бесконечно расти, но библиотека поддерживает механизм снапшотов, позволяющий автоматически удалять старые записи без потери согласованности. Кроме того, Raft позволяет динамически обновлять состав кластера, добавляя или удаляя узлы при наличии кворума.

Производительность Raft сопоставима с Paxos: при стабильном лидерстве новый лог подтверждается за один раунд сетевого взаимодействия с большинством узлов. При этом кластер из 3 узлов выдерживает отказ одного, а из 5 — до двух. Для хранения данных HashiCorp предлагает два бэкенда: "raft-mdb"(основной) и "raft-boltdb"(чистая Go-реализация).

1.3 Braft

Braft — это промышленная реализация алгоритма консенсуса Raft на C++ от Baidu, ориентированная на высокую нагрузку и низкие задержки. Она построена на базе "brpc что делает её эффективным решением для отказоустойчивых распределённых систем.

Эта библиотека широко применяется внутри Baidu для построения различных высокодоступных сервисов: хранилищ (KV, блоковых, файловых, объектных), распределённых SQL-систем, модулей управления метаданными и сервисов блокировок.

Основной упор в разработке Braft сделан на производительность и понятность кода, что позволяет инженерам Baidu самостоятельно создавать распределённые системы без глубокой экспертизы в алгоритмах консенсуса. В отличие от других реализаций Raft, Braft оптимизирована для минимизации задержек и высокой пропускной способности, что делает её особенно подходящей для сценариев с высокой нагрузкой.

Сборка библиотеки требует установки brpc, а установка возможна через "vcpkg что облегчает интеграцию. Braft также предоставляет обширную документацию, включая информацию о модели репликации, поддержке других протоколов консенсуса (Paxos, ZAB, QJM) и бенчмарки производительности. Это делает её удобным инструментом для разработки отказоустойчивых распределённых решений.

1.4 Ratis

Apache Ratis — это Java-библиотека, реализующая алгоритм консенсуса Raft, предназначенная для управления реплицируемым журналом. В отличие от Paxos, Raft предлагает более понятную структуру, что делает его удобной основой для построения практических систем.

Основная цель Apache Ratis — предоставить гибкую и удобную встраиваемую реализацию Raft, которая может использоваться любыми системами, нуждающимися в реплицируемом логе. Она поддерживает модульность, позволяя разработчикам легко интегрировать собственные реализации конечных автоматов, журналов Raft, RPC-коммуникаций и механизмов метрик.

Одной из ключевых особенностей Ratis является ориентация на высокую пропускную способность при записи данных, что делает её подходящим выбором не только для классических систем консенсуса, но и для более общих задач репликации данных. Это особенно важно для сценариев, требующих быстрого и надёжного распространения обновлений между узлами распределённой системы.

Проект активно развивается в рамках Apache, обеспечивая поддержку современных требований к отказоустойчивым распределённым сервисам.

1.5 raft-rs

raft-rs — это реализация алгоритма консенсуса Raft на Rust, созданная для управления реплицируемым логом в распределённых системах. Этот crate предлагает мощный и гибкий базовый модуль консенсуса, который можно адаптировать под разные сценарии, но требует реализации собственных компонентов для хранения логов, управления конечными автоматами и сетевого взаимодействия.

Одним из ключевых преимуществ raft-rs является его производительность и модульность. Он поддерживает работу с rust-protobuf и Prost для кодирования rpc-сообщений, что делает интеграцию с различными системами более удобной. В экосистеме Rust этот crate широко используется в таких проектах, как TiKV — распределённая транзакционная база данных.

Проект активно развивается, предлагая инструменты для тестирования (cargo test), форматирования (cargo fmt) и анализа (cargo clippy), а также бенчмаркинг через Criterion. Благодаря этому raft-rs не только удобен для использования, но и предоставляет разработчикам возможности для оценки и оптимизации производительности их решений.

1.6 Обоснование выбора языка программирования

При выборе языка программирования для реализации отказоустойчивого распределенного хранилища важно учитывать такие факторы, как производительность, удобство работы с конкурентностью, а также наличие проверенных решений для репликации данных. В этом контексте Go является оптимальным выбором, поскольку он изначально разрабатывался для высоконагруженных распределенных систем и предлагает мощные механизмы работы с многопоточностью через goroutines и каналы. Кроме того, язык имеет встроенную поддержку профилирования и мониторинга, что критически важно для отказоустойчивых систем.

1.7 Обоснование выбора реализации алгоритма

После анализа существующих реализаций алгоритма Raft на разных языках программирования и выбора Go в качестве основного языка разработки, следующим этапом является выбор конкретной библиотеки для механизма консенсуса и репликации данных.

Основными критериями выбора являются:

- Надежность и проверенность - библиотека должна успешно применяться в промышленных системах.
- Активная поддержка и развитие – регулярные обновления, исправления ошибок и качественная документация.
- Гибкость интеграции - удобные API для работы с реплицируемыми логами и кластерным управлением.
- Производительность – эффективная обработка логов и минимальные накладные расходы на консенсус.

На основе этих критериев в данной работе будет использоваться **HashiCorp Raft** - легковесная и гибкая реализация Raft, разработанная компанией HashiCorp. Она отличается минимальными зависимостями и удобством интеграции, что делает её оптимальным выбором для отказоустойчивых распределённых систем.

Библиотека поддерживает:

а) Динамическое управление кластером – безопасное добавление и удаление узлов без нарушенияконсистентности.

б) Гибкость хранения логов – возможность использовать различные механизмы хранения, включая BoltDB и MDB.

в) Высокую производительность – механизм снапшотов и эффективная репликация обеспечивают низкие задержки.

HashiCorp Raft применяется в таких продуктах, как Consul и Nomad, что подтверждает её стабильность и эффективность. Таким образом, данная библиотека удовлетворяет всем требованиям, предъявляемым к отказоустойчивой системе репликации данных, и будет использоваться в данной работе в качестве основы для механизма консенсуса.

2 Введение в Raft

Алгоритм Raft подробно рассматривался в предыдущей научной работе, где были детально разобраны его принципы, архитектура и формальные доказательства корректности. В данном разделе будут изложены основные аспекты Raft, необходимые для понимания работы программы, избегая избыточных деталей и сосредоточившись на ключевых механизмах протокола.

Алгоритм Raft был предложен Диего Онгаро и Джоном Оустером в 2014 году как альтернатива Paxos, с акцентом на простоту понимания и удобство реализации. Raft предназначен для управления реплицируемым журналом команд, обеспечивая согласованность данных в распределённых системах.

2.1 Основные принципы Raft

Raft разделяет процесс достижения консенсуса на три ключевых компонента:

- Выбор лидера – механизм, обеспечивающий автоматическое определение единственного управляющего узла (лидера) в кластере.
- Репликация логов – распространение команд от клиента через лидера к остальным узлам (последователям) с гарантией согласованности.
- Безопасность – гарантирует, что все узлы применяют команды в одинаковом порядке и не происходит разделения кластера на противоречивые группы.

Каждый узел в Raft может находиться в одном из трёх состояний:

- Лидер (Leader) – единственный узел, обрабатывающий запросы от клиентов и координирующий репликацию логов.
- Последователь – пассивный узел, принимающий команды от лидера и подтверждающий их выполнение.
- Кандидат – узел, инициирующий процесс выборов в случае отсутствия лидера.

2.2 Выбор лидера

Выбор лидера происходит, когда существующий лидер становится недоступным. Последователи инициируют выборы, переходя в состояние кандидатов и отправляя запросы голосования (RequestVote) остальным узлам. Каждый узел голосует только один раз в одном терминальном периоде (*term*), а кандидат, получивший большинство голосов, становится новым лидером.

Чтобы избежать сплит-голосования, Raft использует случайные таймеры выборов. Это минимизирует вероятность одновременного запуска выборов несколькими узлами.

2.3 Репликация логов

Одной из ключевых задач алгоритма Raft является обеспечение согласованности логов между узлами кластера. Для этого используется механизм репликации записей, который гарантирует, что все узлы имеют одинаковую последовательность команд, применяемых к их состоянию.

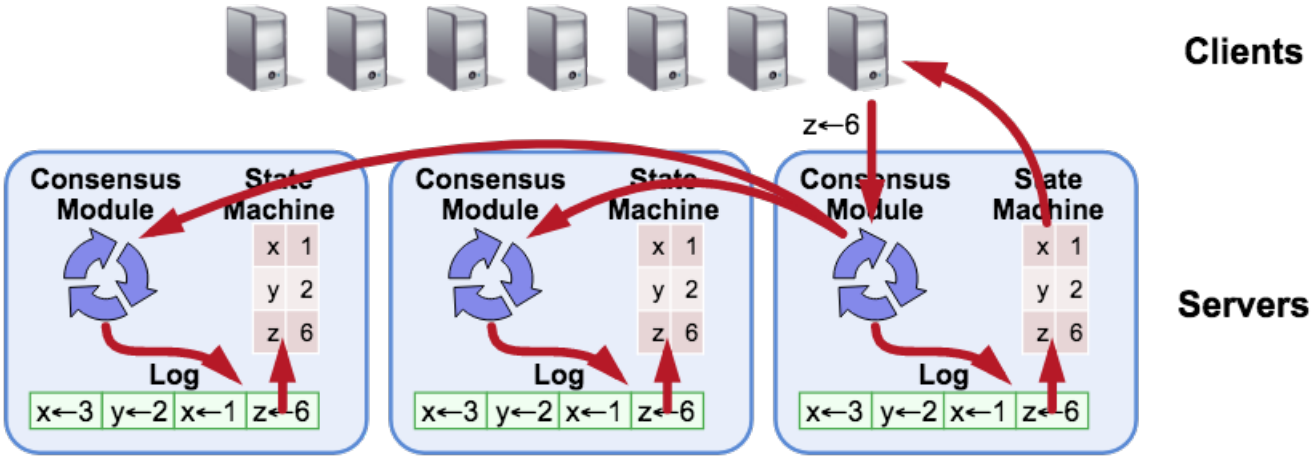

Рисунок 1 – Механизм репликации логов в алгоритме Raft

Процесс репликации логов показаны на рисунке **??**. Клиент отправляет запрос на изменение состояния в кластер, который обрабатывает текущий лидер. Лидер добавляет новую запись в свой лог и рассылает её последователям с помощью сообщений `AppendEntries`. Узлы последовательно добавляют полученные команды в свои журналы, сохраняя порядок операций. Как только большинство узлов подтверждает запись , она считается зафиксированной, после чего команды могут быть применены к конечному автомату каждого узла.

2.4 Механизм безопасности

Raft строго гарантирует, что ни один узел не применит команду до её репликации на большинстве узлов. Кроме того, протокол предотвращает ситуацию, когда новый лидер может иметь устаревшие данные:

- Кандидат должен содержать все подтверждённые записи, иначе он не сможет выиграть выборы.
- Узлы отклоняют запросы от лидера, если у него устаревший термин.
- Если новый лидер обнаруживает конфликтующие записи у последовательных узлов, он удаляет их и заново отправляет правильные данные.

Эти механизмы позволяют Raft поддерживать строгую согласованность даже в случае сбоев.

2.5 Изменение состава кластера

Raft поддерживает безопасное добавление и удаление узлов с помощью механизма *joint consensus*. В этом режиме временно существуют два пересекающихся кворума (старый и новый состав), что гарантирует плавный переход и предотвращает разделение кластера.

3 Проектирование key-value хранилища

В данном разделе будет рассмотрен процесс проектирования распределенного key-value хранилища. Основная цель системы — обеспечить отказоустойчивость и согласованность данных при репликации за счет алгоритма Raft.

3.1 Архитектура

Разрабатываемая система представляет собой распределённое key-value хранилище, работающее на основе алгоритма консенсуса Raft. Архитектура построена по модели "лидер-последователь"(master-slave), как показано на рисунке 2 где один из узлов выполняет роль лидера, а остальные реплицируют его состояние.

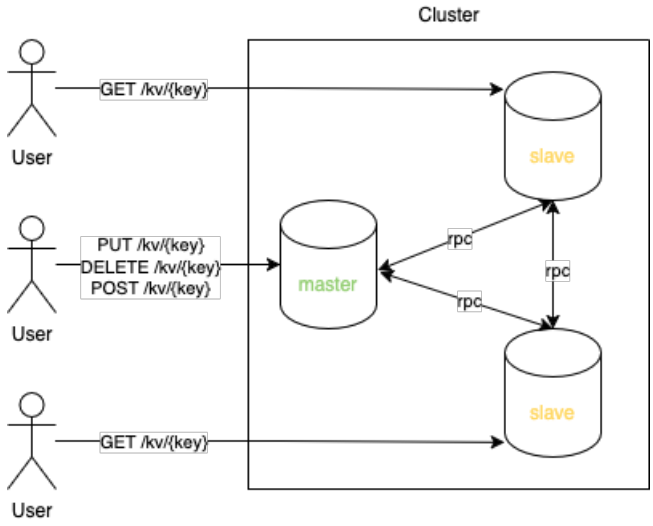

Рисунок 2 – Архитектура key-value хранилища

Клиенты взаимодействуют с системой через HTTP API. Операции записи (POST, DELETE, POST) отправляются только лидеру, который фиксирует изменения в журнале и распространяет их на последовательные узлы. Чтение (GET) может выполняться как на лидере, так и на последовательных узлах, что позволяет балансировать нагрузку.

Обмен данными между узлами осуществляется через бинарный rpc протокол, что обеспечивает эффективную передачу сообщений и низкие задержки. Последовательные узлы не могут принимать изменения напрямую, но участвуют в голосовании при выборе нового лидера в случае сбоя. Лидер принимает запросы на изменение данных и фиксирует их в своём логе. После этого он рассылает команду обновления состояния всем последователям с использова-

нием rpc. Узлы подтверждают получение новой записи и добавляют её в свой лог. Когда большинство узлов кластера подтверждает запись, она считается зафиксированной, и мастер применяет изменения к своему состоянию, после чего сообщает клиенту об успешном завершении операции.

Используемый подход гарантирует согласованность данных и отказоустойчивость, а также позволяет горизонтально масштабировать систему за счёт добавления новых узлов.

3.2 Пользовательский API

Взаимодействие с системой осуществляется через HTTP API, которое предоставляет операции для управления ключами и значениями. Запросы на изменение данных (‘POST‘, ‘DELETE‘) должны направляться лидеру кластера, в то время как чтение (‘GET‘) возможно с любого узла.

Таблица 1 – API кластера KV-хранилища

Метод	**URL**	**Узел**	**Описание**
POST	/key/{key}	Master	Обновление значения, если он есть и создание его в случае отсутствия
DELETE	/key/{key}	Master	Удаление ключа
GET	/key/{key}	Slave	получение значения по ключу

3.3 Проектирование конфига

Конфигурация кластера разработана для обеспечения согласованной работы распределённой системы на основе алгоритма Raft. Она структурирована в формате YAML, как показано в листинге 1, что обеспечивает человекочитаемость и простоту интеграции с инструментами автоматизации. Основные разделы и поля конфигурации обоснованы следующими требованиями:

- **data_dir**: Указывает директорию для хранения данных. В данном случае, данные будут храниться в папке `var`.
- **bin_path**: Путь к директории, где находится исполняемый файл для узлов кластера. В данном примере это директория `bin`.
- **leader**: Указывает на узел, который будет назначен лидером кластера. В данном случае лидер — это `node1`.
- **cluster**: Секция, которая описывает узлы в кластере. Каждый узел имеет следующие параметры:

- `alias`: Уникальный идентификатор узла, который используется для обращения к нему в кластере (например, `node1`, `node2`, `node3`).
- `http_address`: Адрес, по которому узел будет доступен для HTTP-запросов (например, `127.0.0.1:8080`).
- `rpc_address`: Адрес, по которому узел будет доступен для RPC-запросов (например, `127.0.0.1:9000`).

Листинг 1 – Пример конфигурационного файла для распределенного кластера

```
data_dir: var
bin_path: bin
cluster:
  - alias: node1
    http_address: "127.0.0.1:8080"
    rpc_address: "127.0.0.1:9000"
  - alias: node2
    http_address: "127.0.0.1:8081"
    rpc_address: "127.0.0.1:9001"
  - alias: node3
    http_address: "127.0.0.1:8082"
    rpc_address: "127.0.0.1:9002"
leader: node1
```

4 Реализция отказоустойчивого хранилища

Для реализации понадобилось написать хранилище, которое работает по двум протоколам одновременно: HTTP, RPC. Как было сказано выше, первый нужен для взаимодействия с пользователем, второй для взаимодействия внутри кластера между всеми его узлами.

4.1 Демонстрация возможностей хранилища

Исполняемый файл представляет программу, написанную на языке программирования Go, реализующая следующий функционал:

а) Хранение данных в формате ключ-значение.

б) Отказоустойчивость – репликация репликационного лога на все узлы типа Follower.

в) Персистентность – сброс данных, накопившихся в оперативной памяти на диск, что дает возможность восстанавливать состояние узла.

г) Снапшотинг – приложение создает полную копию данных.

Для того, чтобы скомпилировать программу, нужно воспользоваться утилитой Go и исполнить следующую команду из-под корня проекта:

```
go build -o ./bin/node ./cmd/storage/main.go
```

На рисунке 3 представлен вывод команды help для исполняемого файла хранилища.

```
→  kv git:(master) ✗ ./bin/node --help
Usage: ./bin/node [options] <raft-data-path>
  -haddr string
        Set the HTTP bind address (default "localhost:11000")
  -id string
        Node ID. If not set, same as Raft bind address
  -inmem
        Use in-memory storage for Raft
  -join string
        Set join address, if any
  -raddr string
        Set Raft bind address (default "localhost:12000")
```

Рисунок 3 – API для работы с бинарным файлом key-value хранилища

Программа предоставляет набор командных опций, которые используются для настройки и запуска узла в распределённом хранилище. Ниже приведено описание доступных параметров командной строки:

– `haddr string`

  Устанавливает HTTP-адрес, на котором будет доступен узел. По умолчанию значение — `localhost:11000`.

– `id string`

  Устанавливает идентификатор узла. Если не указан, используется значение, равное адресу привязки Raft.

– `inmem`

  Включает использование памяти для хранения данных Raft, вместо использования дискового хранилища.

– `join string`

  Устанавливает адрес узла (как правило он является лидером), к которому новый узел должен подключиться для вступления в кластер.

– `raddr string`

  Устанавливает адрес привязки для Raft. По умолчанию — `localhost:12000`.

Каждый из этих параметров настраивает различные аспекты поведения узла в кластерной системе, включая его сетевую доступность, идентификацию в кластере и метод хранения данных.

Логи каждого узла хранятся по пути `bin_path/alias/node.log`, где `bin_path` и `alias` берутся из конфига. Наглядная демонстрация логирования приложения будет представлена в одном из следующих пунктов.

4.2 Демонстрация возможностей кластерного менеджера

Для того, чтобы не задумываться о каждом узле по отдельности, в частности, как запускать, следить за ним, было принято решение реализовать консольную утилиту, которая по выданному конфигу, формат которого представлен в листинге 1 сможет стартовать кластер одной командой, отслеживать статус всех инстансов хранилищ, а также останавливать кластер.

Утилита была написана с помощью фреймворка cobra, которая выделяется объемным функционалом, мощным инструментарием и легким использованием.

Для того, чтобы скомпилировать программу, нужно воспользоваться утилитой Go и исполнить следующую команду из-под корня проекта:

```
go build -o ./bin/clusterctl ./cmd/clusterctl/main.go
```

На рисунке 3 представлен вывод команды help для исполняемого файла менеджера кластера.

```
→  kv git:(master) ✗ ./bin/clusterctl
CLI utility to manage an hraftd cluster

Usage:
  clusterctl [command]

Available Commands:
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  start       Start the cluster nodes
  status      Show status of the cluster
  stop        Stop the cluster nodes

Flags:
  -c, --config string   path to config file (YAML) (default "config.yaml")
  -h, --help            help for clusterctl

Use "clusterctl [command] --help" for more information about a command.
```

Рисунок 4 – API для работы с бинарным файлом менеджера кластера

Программа предоставляет набор командных опций для управления кластером. Ниже приведены доступные команды и флаги:

- `completion`

  Генерирует скрипт автодополнения для указанной оболочки.
- `help`

  Показывает справку по любой команде.
- `start`

  Запускает узлы кластера согласно его конфигурации.
- `status`

  Показывает статус активных узлов кластера и их роль (лидер или ведомый).
- `stop`

  Останавливает узлы кластера.

  Также доступны следующие флаги:
- `-c, -config string`

  Путь к файлу конфигурации в формате YAML (по умолчанию берется `config.yaml` в текущей директории).
- `-h, -help`

  Показывает справку по программе.

4.3 Дальнейшие шаги развития приложения

Данная работа является наглядным примером работы алгоритма Raft и обладает минимально допустимым функционалом для хорошей демонстрации, однако многих вещей не достает, что можно попытаться исправить в будущем. Для дальнейшего развития программы можно выделить следующие направления:

- Добавление команд в утилиту для управление кластером clusterctl. Например, реализация всех запросов (добавление, удаление, чтение) с данными на кластер, чтобы дополнительно не искать rw-инстанс (лидера).
- Улучшение логирования, использование более эффективных форматов хранения данных, таких как Protocol Buffers.
- Реализация автоматического восстановления узлов, поддержка мультикластерных развертываний для улучшения отказоустойчивости.
- Расширение API для более детализированного мониторинга и управления, интеграция с внешними сервисами (например, Prometheus).
- Шифрование данных, внедрение аутентификации и авторизации для защиты доступа к кластеру.
- Внедрение поддержки более сложных типов данных.

Эти шаги помогут улучшить функциональность, безопасность и производительность системы, обеспечивая её устойчивость и гибкость в реальных условиях эксплуатации.

5 Демонстрация работы кластера

На листинге 2 представлен конфиг кластера, который содержит 5 узлов, лидером будет являться *node1*. Топология выбрана таким образом, чтобы удовлетворяло "правилу большинства которое заключается в том, чтобы кворум (количество узлов, принимающих участие в консенсусе) состоял не менее, чем из $\lceil \frac{N}{2} \rceil + 1$. N – общее количество узлов. Согласно этой формуле, кластер может гарантировать корректность работы вплоть до момента, когда останется не менее 3 узлов.

Листинг 2 – Пример конфигурационного файла для распределенного кластера

```
data_dir: var
bin_path: bin
cluster:
  - alias: node1
    http_address: "127.0.0.1:8080"
    rpc_address: "127.0.0.1:9000"
  - alias: node2
    http_address: "127.0.0.1:8081"
    rpc_address: "127.0.0.1:9001"
  - alias: node3
    http_address: "127.0.0.1:8082"
    rpc_address: "127.0.0.1:9002"
leader: node1
```

5.1 Старт кластера

На рисунке 5 показан вывод команды *start*, которая запускает все инстансы, указанные в конфиге, путь которого передана в аргументах командной строки.

```
→  kv git:(master) ✗ ./bin/clusterctl -c config/config.yml start
2025/05/17 22:38:12 Starting leader "node1"...
2025/05/17 22:38:12 Started "node1" (pid 68204)
2025/05/17 22:38:14 Starting follower "node2"...
2025/05/17 22:38:14 Started "node2" (pid 68213)
2025/05/17 22:38:14 Starting follower "node3"...
2025/05/17 22:38:14 Started "node3" (pid 68214)
2025/05/17 22:38:14 Starting follower "node4"...
2025/05/17 22:38:14 Started "node4" (pid 68215)
2025/05/17 22:38:14 Starting follower "node5"...
2025/05/17 22:38:14 Started "node5" (pid 68216)
```

Рисунок 5 – вывод комнды "clusterctl start"

На выводе представлен лог программы, где говорится о запуске каждого инстанса и ролью, которая соответствует ему. Также указан id процесса (pid) для каждого узла хранилища.

Для того, чтобы узнать текущую топологию кластера, нужно воспользоваться командой *status*. Пример вывода команды представлен на рисунке 6.

```
→  kv git:(master) ✗ ./bin/clusterctl -c config/config.yml status
Leader:   node1 (127.0.0.1:9000)
Followers:
  - node2 (127.0.0.1:9001)
  - node5 (127.0.0.1:9004)
  - node3 (127.0.0.1:9002)
  - node4 (127.0.0.1:9003)
```

Рисунок 6 – вывод комнды "clusterctl status"

На выводе виден список узлов и имя лидера с указанием адресов, которые указывают на RPC протокол.

Далее необходимо рассмотреть поведение узлов (как лидера, так и ведомого) в контексте протокола Raft при старте. Для этого надо проанализировать их логи. На рисунке 7

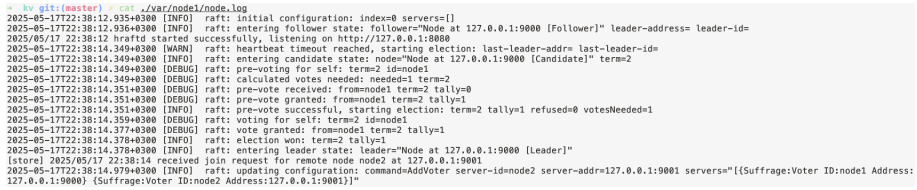

Рисунок 7 – Логи лидера при старте

После инициализации конфигурации Raft узел входит в состояние "Follower ожидая сигналов от лидера. Поскольку сигналов от лидера не поступает, узел переходит в состояние "Candidate"и начинает процесс выборов, увеличивая термин. Он сначала проводит предварительное голосование для себя, затем, получив достаточное количество голосов, запускает основное голосование. Узел выигрывает выборы и становится лидером кластера.

После этого лидер начинает принимать JOIN запросы на добавление новых узлов в кластер и обновляет конфигурацию, добавляя новые узлы, такие как *node2*, *node5*, *node3* и *node4*. На рисунке 8 продемонстрированы логи лидера о вступлении новых узлов в консенсус.

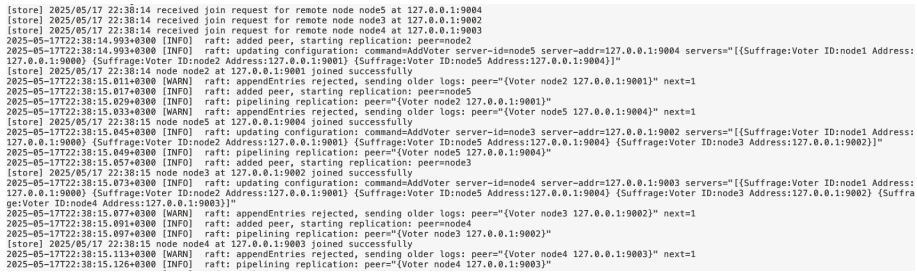

Рисунок 8 – Логи лидера при обработке JOIN запросов

Для каждого нового узла начинается процесс репликации данных с лидера, при этом для синхронизации отправляются старые записи в случае отклонения записей от узлов с устаревшими данными. После успешного присоединения всех узлов кластер начинает работать с обновленной конфигурацией.

Со стороны "ведомых узлов"та же ситуация прсоединения к лидеру представлена на рисунке 9.

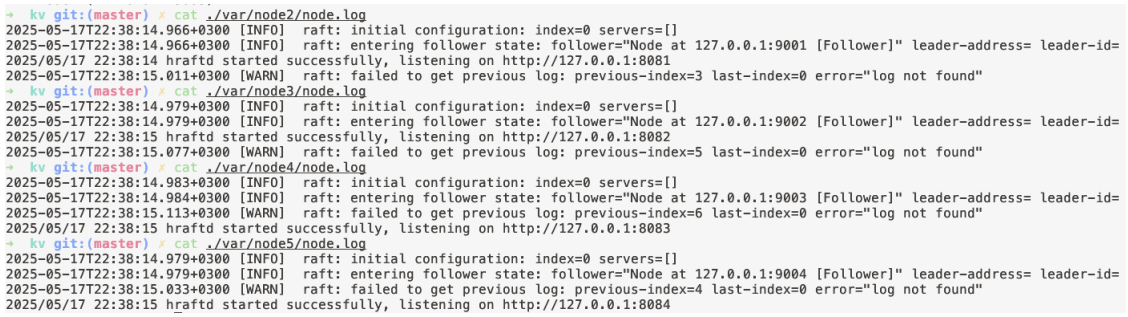

Рисунок 9 – Логи ведомых узлов при присоединении к лидеру

5.2 Операции с данными над кластером

Для того, чтобы вставить данные на кластер, нужно отправить POST запрос на лидер-узел. Для того, чтобы прочитать данные с кластера, достаточно отправить GET запрос с запрашиваемым ключом на любой узел. Отправка запроса представлена на рисунке 8 при помощи утилиты curl.

Рисунок 10 – Вставка данных на кластер

Данные, записанные на лидер, появились на остальных узлах, что говорит об успешной репликации.

Для того, чтобы удалить значение по ключу, необходимо отправить DELETE запрос на лидер-узел. Отправка запроса представлена на рисунке 9.

```
→  kv git:(master) ✗ curl localhost:8084/key/my_first_key
{"my_first_key":"example_value"}%
→  kv git:(master) ✗ curl -X DELETE localhost:8080/key/my_first_key
→  kv git:(master) ✗ curl localhost:8080/key/my_first_key
{"my_first_key":""}%
→  kv git:(master) ✗ curl localhost:8081/key/my_first_key
{"my_first_key":""}%
→  kv git:(master) ✗ curl localhost:8082/key/my_first_key
{"my_first_key":""}%
→  kv git:(master) ✗ curl localhost:8083/key/my_first_key
{"my_first_key":""}%
→  kv git:(master) ✗ curl localhost:8084/key/my_first_key
{"my_first_key":""}%
```

Рисунок 11 – Удаление данных с кластера

Чтение данных узлов показываются на предыдущих двух рисунках, чтобы проверить что данные вставились или удалились. Для этого необходимо отправить GET запрос с ключом, значение которого нужно прочитать.

5.3 Тестирование системы на отказоустойчивость

Теперь, когда все возможности системы были показаны, необходимо проверить выполнение главного критерия, которым должна обладать система – отказоустойчивость.

Перед проведением тестов необходимо заполнить хранилище различными значениями. Заполнение представлено на рисунке 13.

```
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"my_first_key":"example_value"}'
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"1":"1"}'
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"2":"2"}'
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"3":"3"}'
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"4":"4"}'
→  kv git:(master) ✗ curl -X POST localhost:8080/key -d '{"5":"6"}'
→  kv git:(master) ✗ curl localhost:8083/key/my_first_key
{"my_first_key":"example_value"}%
→  kv git:(master) ✗ curl localhost:8083/key/1
{"1":"1"}%
→  kv git:(master) ✗ curl localhost:8082/key/2
{"2":"2"}%
→  kv git:(master) ✗ curl localhost:8081/key/3
{"3":"3"}%
→  kv git:(master) ✗ curl localhost:8084/key/4
{"4":"4"}%
```

Рисунок 12 – Заполнение хранилища данными

Сейчас лидером является узел *node1*. Далее необходимо сэмулировать сутуацию, применимую к реальной жизни, когда узел по какой-либо причине может отказать. Например, возникли проблемы в сети, случился перебой в электричестве, процесс упал с ошибкой и т.д. Мы же просто отправим сигнал перывания на лидирующий узел. Остановка лидера показана на рисунке 14.

Рисунок 13 – Остановка лидирующего узла

Как видно на рисунке, сначала был найден нужный PID узла-лидера, а затем на него был отправлен сигнал прерывания SIGINT.

Необходимо проверить, что случилось с кластером после падение лидера. Вызов команды status показан на рисунке 14.

Рисунок 14 – Статус кластера после выбора нового лидера

Узл *node1* больше не показывается, так как он был остановлен. Теперь лидером является *node3*. Нужно убедиться, каким образом был выбран третий узел. Для этого нужно взгялнуть на логи узлов для детального разбора этапа голосования.

На рисунке 15 представлены логи выбора нового лидера – *node3*.

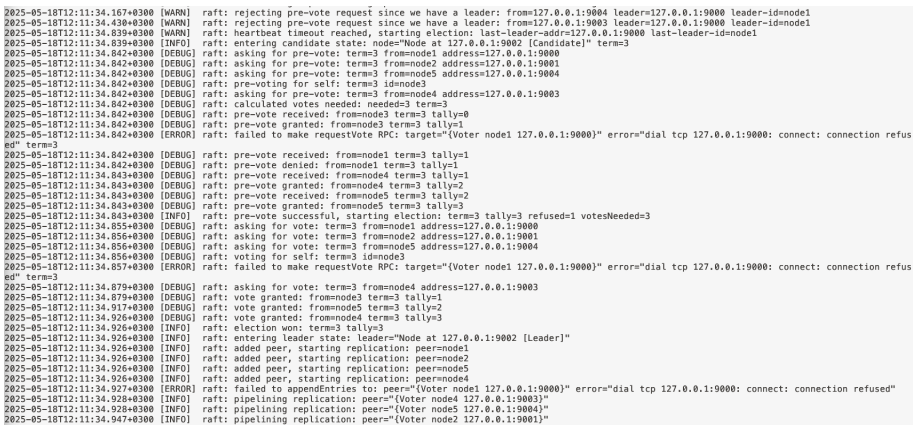

Рисунок 15 – Процесс выбора нового лидера

После того как предыдущий лидер (Узел 1) был остановлен, узлы начали процесс выборов, и Узел 3 перешёл в состояние "Candidate". Он начал запрашивать голоса у других узлов и получал предварительные голоса, включая поддержку от Узла 5. Получив достаточное количество голосов, Узел 3 продолжил выборы, голосуя за себя и получая окончательные голоса. В результате, Узел 3 выиграл выборы, стал лидером и перешёл в состояние "Leader". После этого он начал принимать новые узлы в кластер, включая Узлы 1, 2 и 4, и инициировал репликацию данных для синхронизации всех узлов. В конечном итоге все узлы были успешно синхронизированы, и Узел 3 продолжил оставаться лидером кластера.

Для ведомых узлов представлен процесс голосования на примере *node4*. Его логи представлены на рисунке 16.

```
2025-05-18T12:11:34.167+0300 [WARN]  raft: rejecting pre-vote request since we have a leader: from=127.0.0.1:9004 leader=127.0.0.1:9000 leader-id=node1
2025-05-18T12:11:34.419+0300 [WARN]  raft: heartbeat timeout reached, starting election: last-leader-addr=127.0.0.1:9000 last-leader-id=node1
2025-05-18T12:11:34.419+0300 [INFO]  raft: entering candidate state: node="Node at 127.0.0.1:9003 [Candidate]" term=3
2025-05-18T12:11:34.420+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node1 address=127.0.0.1:9000
2025-05-18T12:11:34.420+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node2 address=127.0.0.1:9001
2025-05-18T12:11:34.420+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node5 address=127.0.0.1:9004
2025-05-18T12:11:34.420+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node3 address=127.0.0.1:9002
2025-05-18T12:11:34.420+0300 [DEBUG] raft: pre-voting for self: term=3 id=node4
2025-05-18T12:11:34.420+0300 [DEBUG] raft: calculated votes needed: needed=3 term=3
2025-05-18T12:11:34.420+0300 [DEBUG] raft: pre-vote received: from=node4 term=3 tally=0
2025-05-18T12:11:34.420+0300 [DEBUG] raft: pre-vote granted: from=node4 term=3 tally=1
2025-05-18T12:11:34.421+0300 [ERROR] raft: failed to make requestVote RPC: target="{Voter node1 127.0.0.1:9000}" error="dial tcp 127.0.0.1:9000: connect: connection refus
ed" term=3
2025-05-18T12:11:34.421+0300 [DEBUG] raft: pre-vote received: from=node1 term=3 tally=1
2025-05-18T12:11:34.421+0300 [DEBUG] raft: pre-vote denied: from=node1 term=3 tally=1
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote received: from=node3 term=2 tally=1
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote denied: from=node3 term=2 tally=1
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote received: from=node2 term=2 tally=1
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote denied: from=node2 term=2 tally=1
2025-05-18T12:11:34.431+0300 [INFO]  raft: pre-vote campaign failed, waiting for election timeout: term=2 tally=1 refused=3 votesNeeded=3
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote received: from=node5 term=3 tally=1
2025-05-18T12:11:34.431+0300 [DEBUG] raft: pre-vote granted: from=node5 term=3 tally=2
2025-05-18T12:11:34.431+0300 [INFO]  raft: pre-vote campaign failed, waiting for election timeout: term=3 tally=2 refused=3 votesNeeded=3
2025-05-18T12:11:34.843+0300 [DEBUG] raft: received a requestPreVote with a newer term, grant the pre-vote
2025-05-18T12:11:34.879+0300 [DEBUG] raft: lost leadership because received a requestVote with a newer term
2025-05-18T12:11:34.925+0300 [INFO]  raft: entering follower state: follower="Node at 127.0.0.1:9003 [Follower]" leader-address= leader-id=
```

Рисунок 16 – Процесс выбора нового лидера со стороны ведомого узла

Когда Узел 3 начал процесс выборов, Узел 1, оставшийся в прежнем статусе, отклонил предварительные запросы на голосование, поскольку у него уже был лидер. Узел 2 и другие узлы начали свои выборы, переходя в состояние "Candidate и начали собирать голоса, но узел 2 столкнулся с ошибкой подключения при попытке голосовать с Узлом 1, так как тот больше не мог участвовать в голосовании. Узел 2 продолжал пытаться собрать голоса, но не смог набрать достаточное количество, поскольку другие узлы также не поддержали его. В итоге Узел 2 и другие узлы вернулись в состояние "Follower ожидая нового лидера, которым стал Узел 3.

Необходимо проверить, что после выбора нового лидера данные остались. На рисунке 18 представлена работа с данными.

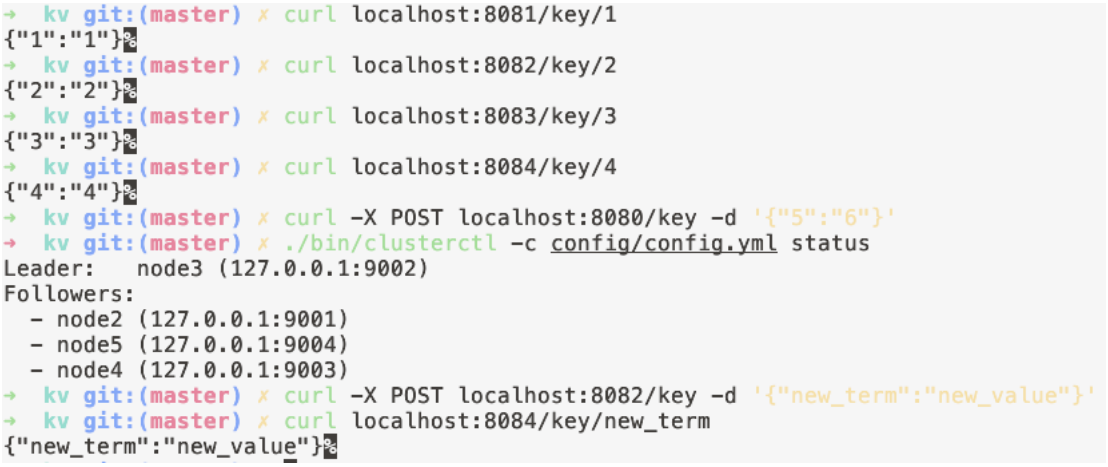

Рисунок 17 – Работа с данными в новом терме

Как видно на рисунке, представленном выше, данные не потерялись. После вставки нового ключа на нового лидера, данные успешно реплецировались на все его реплики. Таким образом был достигнут и проверен консенсус между узлами кластера с помощью алгоритма Raft.

После тестов необходимо остановить кластер. Остановка кластера продемонстрирована на рисунке ??.

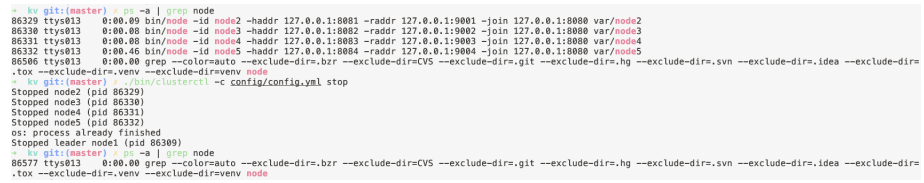

Рисунок 18 – Остановка всего кластера

На рисунке выше показан вывод всех живых узлов до остановки, вызов команды stop, а затем проверка того, что все процессы исчезли, что свидетельствует о корректной работе команды.

ЗАКЛЮЧЕНИЕ

В ходе проектирования и реализации распределённого key-value хранилища на основе алгоритма Raft были достигнуты основные цели: отказоустойчивость, согласованность данных и возможность горизонтального масштабирования системы. Хранилище эффективно справляется с задачами репликации, синхронизации данных и восстановления после сбоев, обеспечивая высокую доступность и целостность данных при отказах узлов.

Использование алгоритма консенсуса Raft позволяет гарантировать, что данные всегда остаются согласованными в распределённой среде, даже в случае сбоя одного или нескольких узлов. Разработанная система легко масштабируется и может добавлять новые узлы без потери данных, обеспечивая возможность добавления, удаления и чтения данных с разных узлов кластера.

Пользовательский API предоставляет удобные инструменты для взаимодействия с хранилищем, позволяя пользователю эффективно управлять данными через простые HTTP запросы. Утилита для управления кластером упрощает процесс настройки и мониторинга системы, позволяя запускать и контролировать состояние всех узлов кластера с помощью нескольких команд.

В рамках тестирования отказоустойчивости было проверено, что система корректно восстанавливает работоспособность при падении лидера, а данные не теряются. Репликация и синхронизация данных между узлами подтверждают стабильность работы кластера в условиях реальных сбоев и изменений конфигурации.

В целом, работа демонстрирует успешную реализацию ключевых принципов распределённых систем с использованием алгоритма Raft, обеспечивая высокую доступность и согласованность данных в распределённом хранилище.

ПРИЛОЖЕНИЕ А

Класс реализации узла хранилища

Листинг А.1 – Реализация Raft в хранилище

```
// Package store provides a simple distributed key-value store. The
    keys and
// associated values are changed via distributed consensus, meaning
    that the
// values are changed only when a majority of nodes in the cluster
   agree on
// the new value.
//
// Distributed consensus is provided via the Raft algorithm,
   specifically the
// Hashicorp implementation.
package storage

import (
  "encoding/json"
  "fmt"
  "io"
  "log"
  "maps"
  "net"
  "os"
  "path/filepath"
  "sync"
  "time"

  "github.com/hashicorp/raft"
  raftboltdb "github.com/hashicorp/raft-boltdb/v2"
)

const (
  retainSnapshotCount = 2
  raftTimeout         = 10 * time.Second
)

type command struct {
  Op    string `json:"op,omitempty"`
  Key   string `json:"key,omitempty"`
```

```
  Value string `json:"value,omitempty"`
}

// Node represents a node in the cluster.
type Node struct {
  ID      string `json:"id"`
  Address string `json:"address"`
}

// StoreStatus is the Status a Store returns.
type StoreStatus struct {
  Me        Node   `json:"me"`
  Leader    Node   `json:"leader"`
  Followers []Node `json:"followers"`
}

// Store is a simple key-value store, where all changes are made
   via Raft consensus.
type Store struct {
  RaftDir  string
  RaftBind string
  inmem    bool

  mu sync.Mutex
  m  map[string]string // The key-value store for the system.

  raft *raft.Raft // The consensus mechanism

  logger *log.Logger
}

// New returns a new Store.
func New(inmem bool) *Store {
  return &Store{
    m:      make(map[string]string),
    inmem:  inmem,
    logger: log.New(os.Stderr, "[store] ", log.LstdFlags),
  }
}

// Open opens the store. If enableSingle is set, and there are no
   existing peers,
```

```
// then this node becomes the first node, and therefore leader, of
   the cluster.
// localID should be the server identifier for this node.
func (s *Store) Open(enableSingle bool, localID string) error {
  // Setup Raft configuration.
  config := raft.DefaultConfig()
  config.LocalID = raft.ServerID(localID)

  // Setup Raft communication.
  addr, err := net.ResolveTCPAddr("tcp", s.RaftBind)
  if err != nil {
    return err
  }
  transport, err := raft.NewTCPTransport(s.RaftBind, addr, 3, 10*
     time.Second, os.Stderr)
  if err != nil {
    return err
  }

  // Create the snapshot store. This allows the Raft to truncate
     the log.
  snapshots, err := raft.NewFileSnapshotStore(s.RaftDir,
     retainSnapshotCount, os.Stderr)
  if err != nil {
    return fmt.Errorf("file snapshot store: %s", err)
  }

  // Create the log store and stable store.
  var logStore raft.LogStore
  var stableStore raft.StableStore
  if s.inmem {
    logStore = raft.NewInmemStore()
    stableStore = raft.NewInmemStore()
  } else {
    boltDB, err := raftboltdb.New(raftboltdb.Options{
      Path: filepath.Join(s.RaftDir, "raft.db"),
    })
    if err != nil {
      return fmt.Errorf("new bbolt store: %s", err)
    }
    logStore = boltDB
    stableStore = boltDB
```

```
  }

  // Instantiate the Raft systems.
  ra, err := raft.NewRaft(config, (*fsm)(s), logStore, stableStore,
      snapshots, transport)
  if err != nil {
    return fmt.Errorf("new raft: %s", err)
  }
  s.raft = ra

  if enableSingle {
    configuration := raft.Configuration{
      Servers: []raft.Server{
        {
          ID:      config.LocalID,
          Address: transport.LocalAddr(),
        },
      },
    }
    ra.BootstrapCluster(configuration)
  }

  return nil
}

// Get returns the value for the given key.
func (s *Store) Get(key string) (string, error) {
  s.mu.Lock()
  defer s.mu.Unlock()
  return s.m[key], nil
}

// Set sets the value for the given key.
func (s *Store) Set(key, value string) error {
  if s.raft.State() != raft.Leader {
    return fmt.Errorf("not leader")
  }

  c := &command{
    Op:    "set",
    Key:   key,
    Value: value,
```

```
  }
  b, err := json.Marshal(c)
  if err != nil {
    return err
  }

  f := s.raft.Apply(b, raftTimeout)
  return f.Error()
}

// Delete deletes the given key.
func (s *Store) Delete(key string) error {
  if s.raft.State() != raft.Leader {
    return fmt.Errorf("not leader")
  }

  c := &command{
    Op:  "delete",
    Key: key,
  }
  b, err := json.Marshal(c)
  if err != nil {
    return err
  }

  f := s.raft.Apply(b, raftTimeout)
  return f.Error()
}

// Join joins a node, identified by nodeID and located at addr, to
   this store.
// The node must be ready to respond to Raft communications at that
    address.
func (s *Store) Join(nodeID, addr string) error {
  s.logger.Printf("received join request for remote node %s at %s",
      nodeID, addr)

  configFuture := s.raft.GetConfiguration()
  if err := configFuture.Error(); err != nil {
    s.logger.Printf("failed to get raft configuration: %v", err)
    return err
  }
```

```
  for _, srv := range configFuture.Configuration().Servers {
    // If a node already exists with either the joining node's ID
       or address,
    // that node may need to be removed from the config first.
    if srv.ID == raft.ServerID(nodeID) || srv.Address == raft.
       ServerAddress(addr) {
      // However if *both* the ID and the address are the same,
         then nothing -- not even
      // a join operation -- is needed.
      if srv.Address == raft.ServerAddress(addr) && srv.ID == raft.
         ServerID(nodeID) {
        s.logger.Printf("node %s at %s already member of cluster,
           ignoring join request", nodeID, addr)
        return nil
      }

      future := s.raft.RemoveServer(srv.ID, 0, 0)
      if err := future.Error(); err != nil {
        return fmt.Errorf("error removing existing node %s at %s: %
           s", nodeID, addr, err)
      }
    }
  }

  f := s.raft.AddVoter(raft.ServerID(nodeID), raft.ServerAddress(
     addr), 0, 0)
  if f.Error() != nil {
    return f.Error()
  }
  s.logger.Printf("node %s at %s joined successfully", nodeID, addr
     )
  return nil
}

// Status returns information about the Store.
func (s *Store) Status() (StoreStatus, error) {
  leaderServerAddr, leaderId := s.raft.LeaderWithID()
  leader := Node{
    ID:      string(leaderId),
    Address: string(leaderServerAddr),
  }
```

```
  servers := s.raft.GetConfiguration().Configuration().Servers
  followers := []Node{}
  me := Node{
    Address: s.RaftBind,
  }
  for _, server := range servers {
    if server.ID != leaderId {
      followers = append(followers, Node{
        ID:      string(server.ID),
        Address: string(server.Address),
      })
    }

    if string(server.Address) == s.RaftBind {
      me = Node{
        ID:      string(server.ID),
        Address: string(server.Address),
      }
    }
  }

  status := StoreStatus{
    Me:        me,
    Leader:    leader,
    Followers: followers,
  }

  return status, nil
}

type fsm Store

// Apply applies a Raft log entry to the key-value store.
func (f *fsm) Apply(l *raft.Log) any {
  var c command
  if err := json.Unmarshal(l.Data, &c); err != nil {
    panic(fmt.Sprintf("failed to unmarshal command: %s", err.Error
       ()))
  }

  switch c.Op {
```

```
  case "set":
    return f.applySet(c.Key, c.Value)
  case "delete":
    return f.applyDelete(c.Key)
  default:
    panic(fmt.Sprintf("unrecognized command op: %s", c.Op))
  }
}

// Snapshot returns a snapshot of the key-value store.
func (f *fsm) Snapshot() (raft.FSMSnapshot, error) {
  f.mu.Lock()
  defer f.mu.Unlock()

  // Clone the map.
  o := make(map[string]string)
  maps.Copy(o, f.m)

  return &fsmSnapshot{store: o}, nil
}

// Restore stores the key-value store to a previous state.
func (f *fsm) Restore(rc io.ReadCloser) error {
  o := make(map[string]string)
  if err := json.NewDecoder(rc).Decode(&o); err != nil {
    return err
  }

  // Set the state from the snapshot, no lock required according to
  // Hashicorp docs.
  f.m = o
  return nil
}

func (f *fsm) applySet(key, value string) any {
  f.mu.Lock()
  defer f.mu.Unlock()
  f.m[key] = value
  return nil
}

func (f *fsm) applyDelete(key string) any {
```

```
  f.mu.Lock()
  defer f.mu.Unlock()
  delete(f.m, key)
  return nil
}

type fsmSnapshot struct {
  store map[string]string
}

func (f *fsmSnapshot) Persist(sink raft.SnapshotSink) error {
  err := func() error {
    // Encode data.
    b, err := json.Marshal(f.store)
    if err != nil {
      return err
    }

    // Write data to sink.
    if _, err := sink.Write(b); err != nil {
      return err
    }

    // Close the sink.
    return sink.Close()
  }()
  if err != nil {
    sink.Cancel()
  }

  return err
}

func (f *fsmSnapshot) Release() {}
```

Листинг А.2 – Реализация HTTP в хранилище

```
// Package httpd provides the HTTP server for accessing the
   distributed key-value store.
// It also provides the endpoint for other nodes to join an
   existing cluster.
package httpd
```

```
import (
  "encoding/json"
  "fmt"
  "io"
  "log"
  "net"
  "net/http"
  "strings"

  store "github.com/themilchenko/kv/internal/storage"
)

// Store is the interface Raft-backed key-value stores must
   implement.
type Store interface {
  // Get returns the value for the given key.
  Get(key string) (string, error)

  // Set sets the value for the given key, via distributed
     consensus.
  Set(key, value string) error

  // Delete removes the given key, via distributed consensus.
  Delete(key string) error

  // Join joins the node, identitifed by nodeID and reachable at
     addr, to the cluster.
  Join(nodeID string, addr string) error

  // Show who is me, the leader, and followers
  Status() (store.StoreStatus, error)
}

// Service provides HTTP service.
type Service struct {
  addr string
  ln   net.Listener

  store Store
}
```

```
// New returns an uninitialized HTTP service.
func New(addr string, store Store) *Service {
  return &Service{
    addr:  addr,
    store: store,
  }
}

// Start starts the service.
func (s *Service) Start() error {
  server := http.Server{
    Handler: s,
  }

  ln, err := net.Listen("tcp", s.addr)
  if err != nil {
    return err
  }
  s.ln = ln

  http.Handle("/", s)

  go func() {
    err := server.Serve(s.ln)
    if err != nil {
      log.Fatalf("HTTP serve: %s", err)
    }
  }()

  return nil
}

// Close closes the service.
func (s *Service) Close() {
  s.ln.Close()
}

// ServeHTTP allows Service to serve HTTP requests.
func (s *Service) ServeHTTP(w http.ResponseWriter, r *http.Request)
    {
  if strings.HasPrefix(r.URL.Path, "/key") {
    s.handleKeyRequest(w, r)
```

```
  } else if r.URL.Path == "/join" {
    s.handleJoin(w, r)
  } else if r.URL.Path == "/status" {
    s.handleStatus(w, r)
  } else {
    w.WriteHeader(http.StatusNotFound)
  }
}

func (s *Service) handleJoin(w http.ResponseWriter, r *http.Request
   ) {
  m := map[string]string{}
  if err := json.NewDecoder(r.Body).Decode(&m); err != nil {
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  if len(m) != 2 {
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  remoteAddr, ok := m["addr"]
  if !ok {
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  nodeID, ok := m["id"]
  if !ok {
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  if err := s.store.Join(nodeID, remoteAddr); err != nil {
    w.WriteHeader(http.StatusInternalServerError)
    return
  }
}

func (s *Service) handleStatus(w http.ResponseWriter, r *http.
   Request) {
```

```
  if r.Method != "GET" {
    w.WriteHeader(http.StatusMethodNotAllowed)
  }

  status, err := s.store.Status()
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }
  // Set the Content-Type header to application/json
  w.Header().Set("Content-Type", "application/json")

  // Encode the response struct to JSON
  statusJson, err := json.Marshal(status)
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }

  // write it to the response writer
  _, err = w.Write(statusJson)
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }
}

func (s *Service) handleKeyRequest(w http.ResponseWriter, r *http.
   Request) {
  getKey := func() string {
    parts := strings.Split(r.URL.Path, "/")
    if len(parts) != 3 {
      return ""
    }
    return parts[2]
  }

  switch r.Method {
  case "GET":
    k := getKey()
    if k == "" {
      w.WriteHeader(http.StatusBadRequest)
```

```
    }
    v, err := s.store.Get(k)
    if err != nil {
      w.WriteHeader(http.StatusInternalServerError)
      return
    }

    b, err := json.Marshal(map[string]string{k: v})
    if err != nil {
      w.WriteHeader(http.StatusInternalServerError)
      return
    }
    io.WriteString(w, string(b))

  case "POST":
    // Read the value from the POST body.
    m := map[string]string{}
    if err := json.NewDecoder(r.Body).Decode(&m); err != nil {
      w.WriteHeader(http.StatusBadRequest)
      return
    }
    for k, v := range m {
      if err := s.store.Set(k, v); err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
      }
    }

  case "DELETE":
    k := getKey()
    fmt.Println(k)
    if k == "" {
      w.WriteHeader(http.StatusBadRequest)
      return
    }
    if err := s.store.Delete(k); err != nil {
      w.WriteHeader(http.StatusInternalServerError)
      return
    }

  default:
    w.WriteHeader(http.StatusMethodNotAllowed)
```

```
  }
  return
}

// Addr returns the address on which the Service is listening
func (s *Service) Addr() net.Addr {
  return s.ln.Addr()
}
```

Листинг А.3 – Точка входа в программу

```
package main

import (
  "bytes"
  "encoding/json"
  "flag"
  "fmt"
  "log"
  "net/http"
  "os"
  "os/signal"

  httpd "github.com/themilchenko/kv/internal/http"
  store "github.com/themilchenko/kv/internal/storage"
)

// Command line defaults.
const (
  DefaultHTTPAddr = "localhost:11000"
  DefaultRaftAddr = "localhost:12000"
)

// Command line parameters.
var (
  inmem    bool
  httpAddr string
  raftAddr string
  joinAddr string
  nodeID   string
)
```

```
func init() {
  flag.BoolVar(&inmem, "inmem", false, "Use in-memory storage for
     Raft")
  flag.StringVar(&httpAddr, "haddr", DefaultHTTPAddr, "Set the HTTP
      bind address")
  flag.StringVar(&raftAddr, "raddr", DefaultRaftAddr, "Set Raft
     bind address")
  flag.StringVar(&joinAddr, "join", "", "Set join address, if any")
  flag.StringVar(&nodeID, "id", "", "Node ID. If not set, same as
     Raft bind address")
  flag.Usage = func() {
    fmt.Fprintf(os.Stderr, "Usage: %s [options] <raft-data-path> \n
       ", os.Args[0])
    flag.PrintDefaults()
  }
}

func main() {
  flag.Parse()
  if flag.NArg() == 0 {
    fmt.Fprintf(os.Stderr, "No Raft storage directory specified\n")
    os.Exit(1)
  }

  if nodeID == "" {
    nodeID = raftAddr
  }

  // Ensure Raft storage exists.
  raftDir := flag.Arg(0)
  if raftDir == "" {
    log.Fatalln("No Raft storage directory specified")
  }
  if err := os.MkdirAll(raftDir, 0o700); err != nil {
    log.Fatalf("failed to create path for Raft storage: %s", err.
       Error())
  }

  s := store.New(inmem)
  s.RaftDir = raftDir
  s.RaftBind = raftAddr
  if err := s.Open(joinAddr == "", nodeID); err != nil {
```

```
    log.Fatalf("failed to open store: %s", err.Error())
  }

  h := httpd.New(httpAddr, s)
  if err := h.Start(); err != nil {
    log.Fatalf("failed to start HTTP service: %s", err.Error())
  }

  // If join was specified, make the join request.
  if joinAddr != "" {
    if err := join(joinAddr, raftAddr, nodeID); err != nil {
      log.Fatalf("failed to join node at %s: %s", joinAddr, err.
          Error())
    }
  }

  // We're up and running!
  log.Printf("hraftd started successfully, listening on http://%s",
      httpAddr)

  terminate := make(chan os.Signal, 1)
  signal.Notify(terminate, os.Interrupt)
  <-terminate
  log.Println("hraftd exiting")
}

func join(joinAddr, raftAddr, nodeID string) error {
  b, err := json.Marshal(map[string]string{"addr": raftAddr, "id":
     nodeID})
  if err != nil {
    return err
  }
  resp, err := http.Post(fmt.Sprintf("http://%s/join", joinAddr), "
     application-type/json", bytes.NewReader(b))
  if err != nil {
    return err
  }
  defer resp.Body.Close()
  return nil
}
```

ПРИЛОЖЕНИЕ Б

Класс реализации менеджера для управления кластером

Листинг Б.4 – Написание схемы конфигурации

```
package config

type Config struct {
  Cluster []*Node `mapstructure:"cluster"`
  DataDir string  `mapstructure:"data_dir"`
  Leader  string  `mapstructure:"leader"`
  BinPath string  `mapstructure:"bin_path"`
}

type Node struct {
  Alias       string `mapstructure:"alias"`
  HttpAddress string `mapstructure:"http_address"`
  RpcAddress  string `mapstructure:"rpc_address"`
}
```

Листинг Б.5 – Реализация парсинга конфигурации

```
package config

import (
  "fmt"

  "github.com/spf13/viper"
)

// Load reads the YAML configuration from the given file path
// and unmarshals it into a Config struct.
func Load(path string) (*Config, error) {
  v := viper.New()
  v.SetConfigFile(path)
  v.SetConfigType("yaml")

  // Optional: set defaults
  v.SetDefault("data_dir", "./var")
  v.SetDefault("bin_path", "./bin")

  // Read in the config file
```

```
  if err := v.ReadInConfig(); err != nil {
    return nil, fmt.Errorf("error reading config file: %w", err)
  }

  // Unmarshal into Config struct
  var cfg Config
  if err := v.Unmarshal(&cfg); err != nil {
    return nil, fmt.Errorf("unable to decode config: %w", err)
  }

  // Validation: ensure at least one node is defined
  if len(cfg.Cluster) == 0 {
    return nil, fmt.Errorf("config error: cluster must contain at
       least one node")
  }

  return &cfg, nil
}
```

Листинг Б.6 – Реализация команды start

```
package start

import (
  "fmt"
  "log"
  "os"
  "os/exec"
  "path/filepath"
  "time"

  "github.com/themilchenko/kv/internal/config"

  "github.com/spf13/cobra"
)

// save PID to file in dataDir
func savePID(dataDir string, pid int) error {
  pidFile := filepath.Join(dataDir, "node.pid")
  pidStr := fmt.Sprintf("%d", pid)
```

```
  if err := os.WriteFile(pidFile, []byte(pidStr), 0o644); err !=
     nil {
    return fmt.Errorf("cannot write pid file %s: %w", pidFile, err)
  }

  return nil
}

func startNode(cfg *config.Config, node *config.Node) error {
  dataDir := filepath.Join(cfg.DataDir, node.Alias)
  if err := os.MkdirAll(dataDir, 0o755); err != nil {
    return fmt.Errorf("cannot create data dir %s: %w", dataDir, err
       )
  }

  bin := filepath.Join(cfg.BinPath, "node")
  logFile := filepath.Join(dataDir, "node.log")

  f, err := os.OpenFile(logFile, os.O_CREATE|os.O_WRONLY|os.
     O_APPEND, 0o644)
  if err != nil {
    return fmt.Errorf("cannot open log %s: %w", logFile, err)
  }

  var cmd *exec.Cmd

  if node.Alias == cfg.Leader {
    log.Printf("Starting leader %q...", node.Alias)

    cmd = exec.Command(bin,
      "-id", node.Alias,
      "-haddr", node.HttpAddress,
      "-raddr", node.RpcAddress,
      dataDir,
    )
  } else {
    log.Printf("Starting follower %q...", node.Alias)

    leaderHttpAddr := ""
    for _, n := range cfg.Cluster {
      if n.Alias == cfg.Leader {
        leaderHttpAddr = n.HttpAddress
```

```
            break
          }
        }

        cmd = exec.Command(bin,
          "-id", node.Alias,
          "-haddr", node.HttpAddress,
          "-raddr", node.RpcAddress,
          "-join", leaderHttpAddr,
          dataDir,
        )
      }

      cmd.Stdout = f
      cmd.Stderr = f

      if err := cmd.Start(); err != nil {
        return fmt.Errorf("failed to start node %s: %w", node.Alias,
           err)
      }

      pid := cmd.Process.Pid
      log.Printf("Started %q (pid %d)", node.Alias, pid)

      if err := savePID(dataDir, pid); err != nil {
        return err
      }

      return nil
    }

    // Cmd returns the start command
    func Cmd(cfg **config.Config, binPath *string) *cobra.Command {
      return &cobra.Command{
        Use:   "start",
        Short: "Start the cluster nodes",
        RunE: func(cmd *cobra.Command, args []string) error {
          c := *cfg // dereference loaded config

          // launch leader then followers
          var leaderNode *config.Node
```

```
      for _, n := range c.Cluster {
        if n.Alias == c.Leader {
          leaderNode = n
          break
        }
      }

      if leaderNode == nil {
        return fmt.Errorf("leader %s not in cluster", c.Leader)
      }

      if err := startNode(c, leaderNode); err != nil {
        return err
      }

      time.Sleep(2 * time.Second)

      for _, n := range c.Cluster {
        if n.Alias == c.Leader {
          continue
        }

        if err := startNode(c, n); err != nil {
          return err
        }
      }
      return nil
    },
  }
}
```

Листинг Б.7 – Реализация команды status

```
package status

import (
  "encoding/json"
  "errors"
  "fmt"
  "io"
  "net/http"
```

```
  "github.com/themilchenko/kv/internal/config"

  "github.com/spf13/cobra"
)

// statusResponse represents the JSON returned by the node's /
   status endpoint
type statusResponse struct {
  Me        nodeInfo   `json:"me"`
  Leader    nodeInfo   `json:"leader"`
  Followers []nodeInfo `json:"followers"`
}

type nodeInfo struct {
  ID      string `json:"id"`
  Address string `json:"address"`
}

// Cmd returns the status command.
func Cmd(cfg **config.Config) *cobra.Command {
  return &cobra.Command{
    Use:   "status",
    Short: "Show status of the cluster",
    RunE: func(cmd *cobra.Command, args []string) error {
      c := *cfg

      errs := make([]error, 0)
      var status statusResponse
      inactiveAddrs := make(map[string]struct{})

      for _, n := range c.Cluster {
        url := fmt.Sprintf("http://%s/status", n.HttpAddress)

        resp, err := http.Get(url)
        if err != nil {
          inactiveAddrs[n.Alias] = struct{}{}
          errs = append(errs, fmt.Errorf("failed to query status
             endpoint %s: %w", url, err))

          continue
        }
        defer resp.Body.Close()
```

```
        if resp.StatusCode == http.StatusOK {
          body, err := io.ReadAll(resp.Body)
          if err != nil {
            return fmt.Errorf("unable to read response body: %w",
               err)
          }

          if err := json.Unmarshal(body, &status); err != nil {
            return fmt.Errorf("invalid JSON from status endpoint: %
               w", err)
          }

          break
        }

        errs = append(errs, fmt.Errorf("unexpected status code %d
           from %s", resp.StatusCode, url))
      }

      if len(errs) == len(c.Cluster) {
        return errors.Join(errs...)
      }

      fmt.Printf("Leader:   %s (%s)\n", status.Leader.ID, status.
         Leader.Address)
      fmt.Println("Followers:")
      for _, f := range status.Followers {
        if _, ok := inactiveAddrs[f.ID]; !ok {
          fmt.Printf("  - %s (%s)\n", f.ID, f.Address)
        }
      }

      return nil
    },
  }
}
```

Листинг Б.8 – Реализация команды stop

```
package stop
```

```
import (
  "errors"
  "fmt"
  "os"
  "path/filepath"
  "strconv"
  "syscall"

  "github.com/spf13/cobra"
  "github.com/themilchenko/kv/internal/config"
)

// readPID reads the pid from node.pid file in the given dataDir
func readPID(dataDir string) (int, error) {
  pidFile := filepath.Join(dataDir, "node.pid")
  data, err := os.ReadFile(pidFile)
  if err != nil {
    return 0, fmt.Errorf("cannot read pid file %s: %w", pidFile,
       err)
  }
  pidStr := string(data)
  pid, err := strconv.Atoi(pidStr)
  if err != nil {
    return 0, fmt.Errorf("invalid pid %q in file %s: %w", pidStr,
       pidFile, err)
  }
  return pid, nil
}

// Cmd returns the stop command
func Cmd(cfg **config.Config, _ *string) *cobra.Command {
  return &cobra.Command{
    Use:   "stop",
    Short: "Stop the cluster nodes",
    RunE: func(cmd *cobra.Command, args []string) error {
      c := *cfg
      // stop followers then leader to avoid split-brain
      for _, n := range c.Cluster {
        if n.Alias == c.Leader {
          continue
        }
        dataDir := filepath.Join(c.DataDir, n.Alias)
```

```
    pid, err := readPID(dataDir)
    if err != nil {
      fmt.Fprintf(os.Stderr, "warning: %v\n", err)
      continue
    }
    proc, err := os.FindProcess(pid)
    if err != nil {
      fmt.Fprintf(os.Stderr, "warning: cannot find process %d:
         %v\n", pid, err)
      continue
    }
    if err := proc.Signal(syscall.SIGTERM); err != nil {
      fmt.Fprintf(os.Stderr, "warning: failed to terminate %s (
         pid %d): %v\n", n.Alias, pid, err)
    } else {
      fmt.Printf("Stopped %s (pid %d)\n", n.Alias, pid)
    }
  }
  // now stop leader
  // find leader node
  var leaderNode *config.Node
  for _, n := range c.Cluster {
    if n.Alias == c.Leader {
      leaderNode = n
      break
    }
  }
  if leaderNode != nil {
    dataDir := filepath.Join(c.DataDir, leaderNode.Alias)
    pid, err := readPID(dataDir)
    if err != nil {
      return err
    }
    proc, err := os.FindProcess(pid)
    if err != nil {
      return fmt.Errorf("cannot find leader process %d: %w",
         pid, err)
    }
    if err := proc.Signal(syscall.SIGTERM); err != nil {
      if errors.Is(err, os.ErrProcessDone) {
        fmt.Println(err)
      } else {
```

```
            return fmt.Errorf("failed to terminate leader %s (pid %
                d): %w", leaderNode.Alias, pid, err)
          }
        }
        fmt.Printf("Stopped leader %s (pid %d)\n", leaderNode.Alias
            , pid)
      }
      return nil
    },
  }
}
```

Листинг Б.9 – Точка входа в программу

```
package main

import (
  "github.com/themilchenko/kv/internal/cli/root"
)

func main() {
  root.Execute()
}
```