Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет «Информатика и системы управления» (ИУ)

Кафедра «Информационная безопасность» (ИУ8)

Отчет

по научно-исследовательской работе студента

на тему Создание отказоустойчивого распределенного хранилища данных на основе алгоритма Raft

ФИО студента: Мильченко Иван Дмитриевич

Группа: ИУ8-104-2025 Л.д.: 20У633

Специальность: 10.05.01 Компьютерная безопасность

Специализация: 10.05.01 01 Математические методы защиты информации

Научный руководитель НИРС старший преподаватель кафедры ИУ8 Ковынёв Николай Витальевич

Работа выполнена _____ И.Д.Мильченко
дата подпись студента И.О.Фамилия студента

Допуск к защите _____ Н.В.Ковынёв
дата подпись научного руководителя И.О.Фамилия научного руководителя

Отчет принят _____ Д.О.Левиев
дата подпись ответственного за НИРС И.О.Фамилия Ответственного за НИРС

Результаты защиты НИРС			
Дата	Балл	Подпись	ФИО

Москва
2025

РЕФЕРАТ

Отчёт содержит 32 стр., 22 рис., 1 табл., 8 источн.

Ключевые слова: алгоритм Raft, распределённые системы, key-value хранилище, отказоустойчивость, репликация, Go.

Целью проведённой научно-исследовательской работы является проектирование и разработка отказоустойчивого распределённого key-value хранилища, использующего алгоритм Raft для достижения консенсуса между узлами кластера.

В процессе работы выполнено исследование существующих реализаций Raft (*etcd/raft*, *HashiCorp Raft*, *Braft*, *Apache Ratis*, *raft-rs*) и проанализированы их преимущества и ограничения. На основании критериев надёжности, активности сообщества, гибкости интеграции и производительности выбрана библиотека *HashiCorp Raft* на языке Go для последующей реализации механизма консенсуса. Разработана архитектура хранилища в модели "лидер—последователь определён HTTP/RPC API, формат конфигурационных файлов и структура каталогов данных.

Реализованы:

- узел хранилища с поддержкой операций *GET / POST / DELETE*, журнал Raft, снапшоты и механизм восстановления;
- кластерный менеджер *clusterctl*, обеспечивающий запуск, остановку, мониторинг и динамическое масштабирование кластера;
- процедуры репликации логов, смены лидера, безопасного добавления и удаления узлов.

В ходе тестирования подтверждена устойчивость системы к отказам: при остановке лидирующего узла кластер автоматически выбирает нового лидера, а согласованность данных сохраняется. Производственные сценарии репликации и восстановления продемонстрированы на кластере из 5 узлов.

В результате работы создано работоспособное распределённое key-value хранилище, пригодное для применения в сервисах, где критичны высокая доступность и целостность данных. Предложены направления дальнейшего развития: расширение API, интеграция с системами мониторинга, внедрение шифрования и поддержка сложных типов данных.

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 [1] 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.

Далее будет описано каждое решение: его сильные и слабые стороны, удобство в использовании, сравнение с другими реализациями, а затем будет сделан выбор, какую библиотеку использовать.

1.1 Etcd-io

Библиотека Raft [2] 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 [3] — 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 [4] — 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 [5] — 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 [6] — 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 [7] 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.

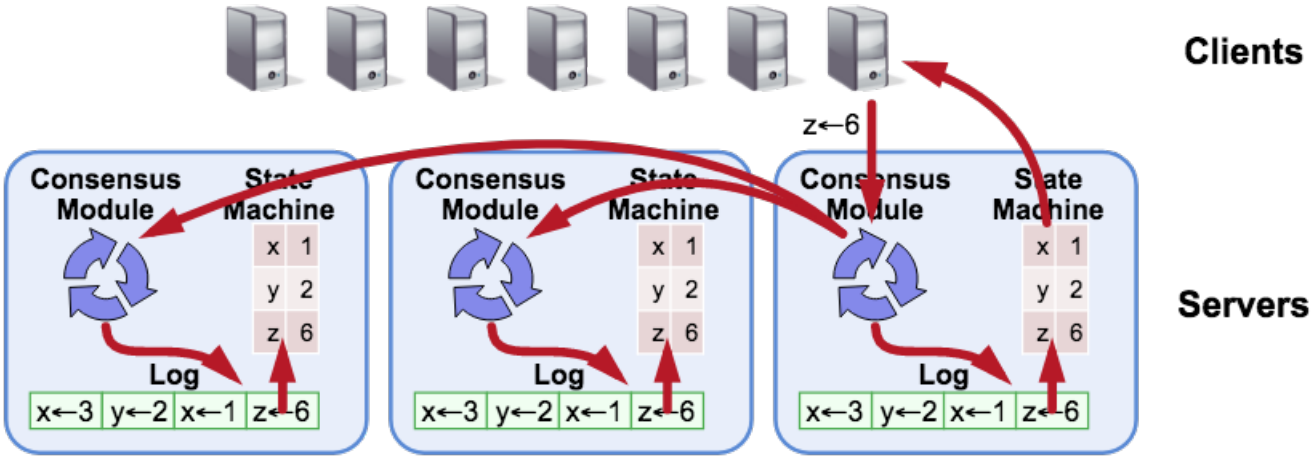

Рисунок 1 – Механизм репликации логов в алгоритме Raft [6]

Процесс репликации логов показаны на рисунке 1. 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.

Такой мехнизм называется *split-brain*, когда между одним кластером возникают, напрмер, сетевые проблемы, в следствии чего один кворум распадается на два.

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.

Далее будут рассмотрена архитектура храшилища, взаимодействие пользователя с сервисом с помощью HTTP API, а также улучшение пользовательского опыта с помощью консольной утилиты с гибкой конфигурацией.

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.

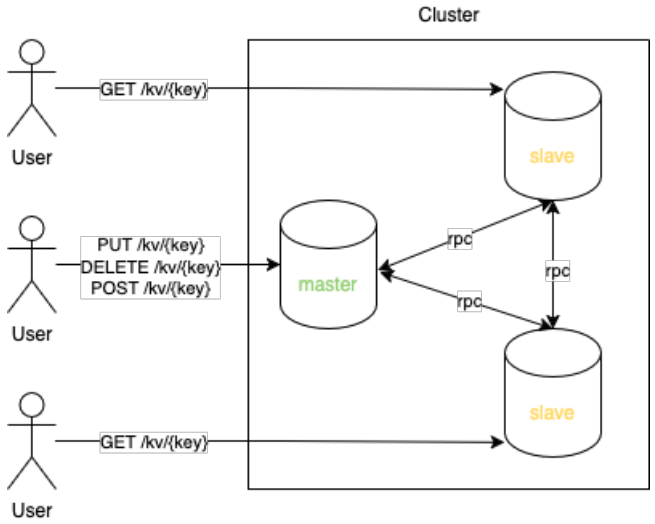

Рисунок 2 – Архитектура key-value хранилища

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 использованием 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. Основные разделы и поля конфигурации обоснованы следующими требованиями:

а) `data_dir`: Указывает директорию для хранения данных. В данном случае, данные будут храниться в папке `var`.

б) `bin_path`: Путь к директории, где находится исполняемый файл для узлов кластера. В данном примере это директория `bin`.

в) `leader`: Указывает на узел, который будет назначен лидером кластера. В данном случае лидер — это `node1`.

г) `cluster`: Секция, которая описывает узлы в кластере.

Каждый узел имеет следующие параметры:

а) `alias`: Уникальный идентификатор узла, который используется для обращения к нему в кластере (например, `node1`, `node2`, `node3`).

б) `http_address`: Адрес, по которому узел будет доступен для HTTP-запросов (например, `127.0.0.1:8080`).

в) `rpc_address`: Адрес, по которому узел будет доступен для RPC-запросов (например, `127.0.0.1:9000`).

Конфигурационный файл структурирован в формате YAML, как показано на рисунке 3, что обеспечивает человекочитаемость и простоту интеграции с инструментами автоматизации.

```yaml
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"
```

Рисунок 3 – Пример конфигурационного файла для распределенного кластера

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.

Также была реализована консольная утилита для удобства в старте всех узлов кластера, просмотра статусов каждого узла и остановка всех узлов с помощью написанного файла конфигурации.

Исходный код проекта доступен в публичном доступе в репозитории GitHub [8].

4.1 Демонстрация возможностей хранилища

Исполняемый файл представляет программу, написанную на языке программирования Го, 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
```

Рисунок 4 – Сборка узла хранилища

На рисунке 5 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 5 – 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 на рисунке 6 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
```

Рисунок 6 – Сборка консольной утилиты clusterctl

На рисунке 5 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 7 – 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 рисунке 8 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.

```
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"
  - alias: node4
     http_address: "127.0.0.1:8083"
     rpc_address: "127.0.0.1:9003"
  - alias: node5
     http_address: "127.0.0.1:8084"
     rpc_address: "127.0.0.1:9004"
leader: node1
```

Рисунок 8 – Пример конфигурационного файла для распределенного кластера

5.1 Старт кластера

На рисунке 9 показан вывод команды *start*, которая запускает все инстансы, указанные в конфиге, путь которого передана в аргументах командной строки.

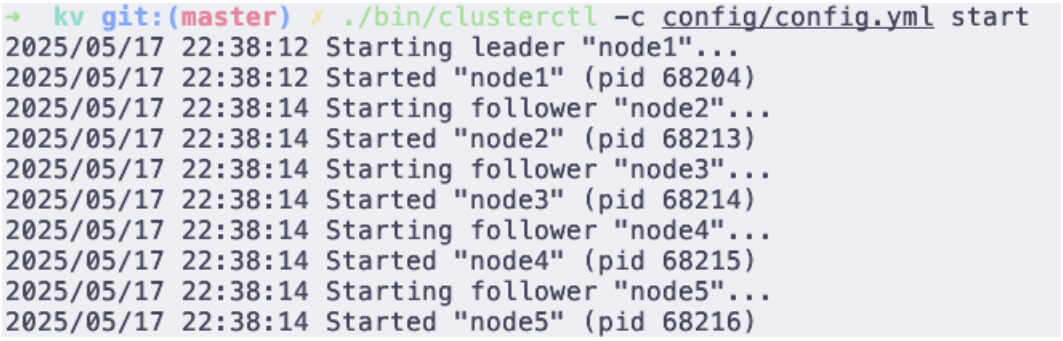

Рисунок 9 – вывод комнды "clusterctl start"

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 10.

Рисунок 10 – 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 11

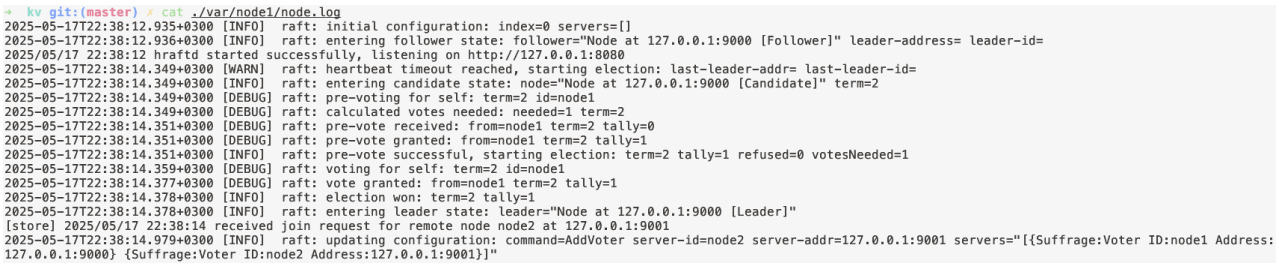

Рисунок 11 – Логи лидера при старте

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 12 продемонстрированы логи лидера о вступлении новых узлов в консенсус.

```
[store] 2025/05/17 22:38:14 received join request for remote node node5 at 127.0.0.1:9004
[store] 2025/05/17 22:38:14 received join request for remote node node3 at 127.0.0.1:9002
[store] 2025/05/17 22:38:14 received join request for remote node node4 at 127.0.0.1:9003
2025-05-17T22:38:14.993+0300 [INFO]  raft: added peer, starting replication: peer=node2
2025-05-17T22:38:14.993+0300 [INFO]  raft: updating configuration: command=AddVoter server-id=node5 server-addr=127.0.0.1:9004 servers="[{Suffrage:Voter ID:node1 Address:
127.0.0.1:9000} {Suffrage:Voter ID:node2 Address:127.0.0.1:9001} {Suffrage:Voter ID:node5 Address:127.0.0.1:9004}]"
[store] 2025/05/17 22:38:14 node node2 at 127.0.0.1:9001 joined successfully
2025-05-17T22:38:15.011+0300 [WARN]  raft: appendEntries rejected, sending older logs: peer="{Voter node2 127.0.0.1:9001}" next=1
2025-05-17T22:38:15.017+0300 [INFO]  raft: added peer, starting replication: peer=node5
2025-05-17T22:38:15.029+0300 [INFO]  raft: pipelining replication: peer="{Voter node2 127.0.0.1:9001}"
2025-05-17T22:38:15.033+0300 [WARN]  raft: appendEntries rejected, sending older logs: peer="{Voter node5 127.0.0.1:9004}" next=1
[store] 2025/05/17 22:38:15 node node5 at 127.0.0.1:9004 joined successfully
2025-05-17T22:38:15.045+0300 [INFO]  raft: updating configuration: command=AddVoter server-id=node3 server-addr=127.0.0.1:9002 servers="[{Suffrage:Voter ID:node1 Address:
127.0.0.1:9000} {Suffrage:Voter ID:node2 Address:127.0.0.1:9001} {Suffrage:Voter ID:node5 Address:127.0.0.1:9004} {Suffrage:Voter ID:node3 Address:127.0.0.1:9002}]"
2025-05-17T22:38:15.049+0300 [INFO]  raft: pipelining replication: peer="{Voter node5 127.0.0.1:9004}"
2025-05-17T22:38:15.057+0300 [INFO]  raft: added peer, starting replication: peer=node3
[store] 2025/05/17 22:38:15 node node3 at 127.0.0.1:9002 joined successfully
2025-05-17T22:38:15.073+0300 [INFO]  raft: updating configuration: command=AddVoter server-id=node4 server-addr=127.0.0.1:9003 servers="[{Suffrage:Voter ID:node1 Address:
127.0.0.1:9000} {Suffrage:Voter ID:node2 Address:127.0.0.1:9001} {Suffrage:Voter ID:node5 Address:127.0.0.1:9004} {Suffrage:Voter ID:node3 Address:127.0.0.1:9002} {Suffra
ge:Voter ID:node4 Address:127.0.0.1:9003}]"
2025-05-17T22:38:15.077+0300 [WARN]  raft: appendEntries rejected, sending older logs: peer="{Voter node3 127.0.0.1:9002}" next=1
2025-05-17T22:38:15.091+0300 [INFO]  raft: added peer, starting replication: peer=node4
2025-05-17T22:38:15.097+0300 [INFO]  raft: pipelining replication: peer="{Voter node3 127.0.0.1:9002}"
[store] 2025/05/17 22:38:15 node node4 at 127.0.0.1:9003 joined successfully
2025-05-17T22:38:15.113+0300 [WARN]  raft: appendEntries rejected, sending older logs: peer="{Voter node4 127.0.0.1:9003}" next=1
2025-05-17T22:38:15.126+0300 [INFO]  raft: pipelining replication: peer="{Voter node4 127.0.0.1:9003}"
```

Рисунок 12 – 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 13.

```
→  kv git:(master) ✗ cat ./var/node2/node.log
2025-05-17T22:38:14.966+0300 [INFO]  raft: initial configuration: index=0 servers=[]
2025-05-17T22:38:14.966+0300 [INFO]  raft: entering follower state: follower="Node at 127.0.0.1:9001 [Follower]" leader-address= leader-id=
2025/05/17 22:38:14 hraftd started successfully, listening on http://127.0.0.1:8081
2025-05-17T22:38:15.011+0300 [WARN]  raft: failed to get previous log: previous-index=3 last-index=0 error="log not found"
→  kv git:(master) ✗ cat ./var/node3/node.log
2025-05-17T22:38:14.979+0300 [INFO]  raft: initial configuration: index=0 servers=[]
2025-05-17T22:38:14.979+0300 [INFO]  raft: entering follower state: follower="Node at 127.0.0.1:9002 [Follower]" leader-address= leader-id=
2025/05/17 22:38:15 hraftd started successfully, listening on http://127.0.0.1:8082
2025-05-17T22:38:15.077+0300 [WARN]  raft: failed to get previous log: previous-index=5 last-index=0 error="log not found"
→  kv git:(master) ✗ cat ./var/node4/node.log
2025-05-17T22:38:14.983+0300 [INFO]  raft: initial configuration: index=0 servers=[]
2025-05-17T22:38:14.984+0300 [INFO]  raft: entering follower state: follower="Node at 127.0.0.1:9003 [Follower]" leader-address= leader-id=
2025-05-17T22:38:15.113+0300 [WARN]  raft: failed to get previous log: previous-index=6 last-index=0 error="log not found"
2025/05/17 22:38:15 hraftd started successfully, listening on http://127.0.0.1:8083
→  kv git:(master) ✗ cat ./var/node5/node.log
2025-05-17T22:38:14.979+0300 [INFO]  raft: initial configuration: index=0 servers=[]
2025-05-17T22:38:14.979+0300 [INFO]  raft: entering follower state: follower="Node at 127.0.0.1:9004 [Follower]" leader-address= leader-id=
2025-05-17T22:38:15.033+0300 [WARN]  raft: failed to get previous log: previous-index=4 last-index=0 error="log not found"
2025/05/17 22:38:15 hraftd started successfully, listening on http://127.0.0.1:8084
```

Рисунок 13 – 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 12 при помощи утилиты curl.

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

Рисунок 14 – 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 13.

```
→  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 15 – 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 17.

```
→  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 16 – 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 18.

```
→  kv git:(master) ✗ ps -a | grep node
68204 ttys013    9:32.11 bin/node -id node1 -haddr 127.0.0.1:8080 -raddr 127.0.0.1:9000 var/node1
68213 ttys013    1:17.34 bin/node -id node2 -haddr 127.0.0.1:8081 -raddr 127.0.0.1:9001 -join 127.0.0.1:8080 var/node2
68214 ttys013    1:17.44 bin/node -id node3 -haddr 127.0.0.1:8082 -raddr 127.0.0.1:9002 -join 127.0.0.1:8080 var/node3
68215 ttys013    1:17.06 bin/node -id node4 -haddr 127.0.0.1:8083 -raddr 127.0.0.1:9003 -join 127.0.0.1:8080 var/node4
68216 ttys013    1:16.97 bin/node -id node5 -haddr 127.0.0.1:8084 -raddr 127.0.0.1:9004 -join 127.0.0.1:8080 var/node5
84161 ttys013    0:00.00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --exclude-dir=.idea --exclude-dir=
.tox --exclude-dir=.venv --exclude-dir=venv node
→  kv git:(master) ✗ kill -SIGINT 68204
```

Рисунок 17 – 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 18.

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

Рисунок 18 – 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 19 представлены логи выбора нового лидера – *node3*.

```
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
2025-05-18T12:11:34.430+0300 [WARN]  raft: rejecting pre-vote request since we have a leader: from=127.0.0.1:9003 leader=127.0.0.1:9000 leader-id=node1
2025-05-18T12:11:34.839+0300 [WARN]  raft: heartbeat timeout reached, starting election: last-leader-addr=127.0.0.1:9000 last-leader-id=node1
2025-05-18T12:11:34.839+0300 [INFO]  raft: entering candidate state: node="Node at 127.0.0.1:9002 [Candidate]" term=3
2025-05-18T12:11:34.842+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node1 address=127.0.0.1:9000
2025-05-18T12:11:34.842+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node2 address=127.0.0.1:9001
2025-05-18T12:11:34.842+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node5 address=127.0.0.1:9004
2025-05-18T12:11:34.842+0300 [DEBUG] raft: pre-voting for self: term=3 id=node3
2025-05-18T12:11:34.842+0300 [DEBUG] raft: asking for pre-vote: term=3 from=node4 address=127.0.0.1:9003
2025-05-18T12:11:34.842+0300 [DEBUG] raft: calculated votes needed: needed=3 term=3
2025-05-18T12:11:34.842+0300 [DEBUG] raft: pre-vote received: from=node3 term=3 tally=0
2025-05-18T12:11:34.842+0300 [DEBUG] raft: pre-vote granted: from=node3 term=3 tally=1
2025-05-18T12:11:34.842+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
2025-05-18T12:11:34.842+0300 [DEBUG] raft: pre-vote received: from=node1 term=3 tally=1
2025-05-18T12:11:34.842+0300 [DEBUG] raft: pre-vote denied: from=node1 term=3 tally=1
2025-05-18T12:11:34.843+0300 [DEBUG] raft: pre-vote received: from=node4 term=3 tally=1
2025-05-18T12:11:34.843+0300 [DEBUG] raft: pre-vote granted: from=node4 term=3 tally=2
2025-05-18T12:11:34.843+0300 [DEBUG] raft: pre-vote received: from=node5 term=3 tally=2
2025-05-18T12:11:34.843+0300 [DEBUG] raft: pre-vote granted: from=node5 term=3 tally=3
2025-05-18T12:11:34.843+0300 [INFO]  raft: pre-vote successful, starting election: term=3 tally=3 refused=1 votesNeeded=3
2025-05-18T12:11:34.855+0300 [DEBUG] raft: asking for vote: term=3 from=node1 address=127.0.0.1:9000
2025-05-18T12:11:34.856+0300 [DEBUG] raft: asking for vote: term=3 from=node2 address=127.0.0.1:9001
2025-05-18T12:11:34.856+0300 [DEBUG] raft: asking for vote: term=3 from=node5 address=127.0.0.1:9004
2025-05-18T12:11:34.856+0300 [DEBUG] raft: voting for self: term=3 id=node3
2025-05-18T12:11:34.857+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
2025-05-18T12:11:34.879+0300 [DEBUG] raft: asking for vote: term=3 from=node4 address=127.0.0.1:9003
2025-05-18T12:11:34.879+0300 [DEBUG] raft: vote granted: from=node3 term=3 tally=1
2025-05-18T12:11:34.917+0300 [DEBUG] raft: vote granted: from=node5 term=3 tally=2
2025-05-18T12:11:34.926+0300 [DEBUG] raft: vote granted: from=node4 term=3 tally=3
2025-05-18T12:11:34.926+0300 [INFO]  raft: election won: term=3 tally=3
2025-05-18T12:11:34.926+0300 [INFO]  raft: entering leader state: leader="Node at 127.0.0.1:9002 [Leader]"
2025-05-18T12:11:34.926+0300 [INFO]  raft: added peer, starting replication: peer=node1
2025-05-18T12:11:34.926+0300 [INFO]  raft: added peer, starting replication: peer=node2
2025-05-18T12:11:34.926+0300 [INFO]  raft: added peer, starting replication: peer=node5
2025-05-18T12:11:34.926+0300 [INFO]  raft: added peer, starting replication: peer=node4
2025-05-18T12:11:34.927+0300 [ERROR] raft: failed to appendEntries to: peer="{Voter node1 127.0.0.1:9000}" error="dial tcp 127.0.0.1:9000: connect: connection refused"
2025-05-18T12:11:34.928+0300 [INFO]  raft: pipelining replication: peer="{Voter node4 127.0.0.1:9003}"
2025-05-18T12:11:34.928+0300 [INFO]  raft: pipelining replication: peer="{Voter node5 127.0.0.1:9004}"
2025-05-18T12:11:34.947+0300 [INFO]  raft: pipelining replication: peer="{Voter node2 127.0.0.1:9001}"
```

Рисунок 19 – 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 20.

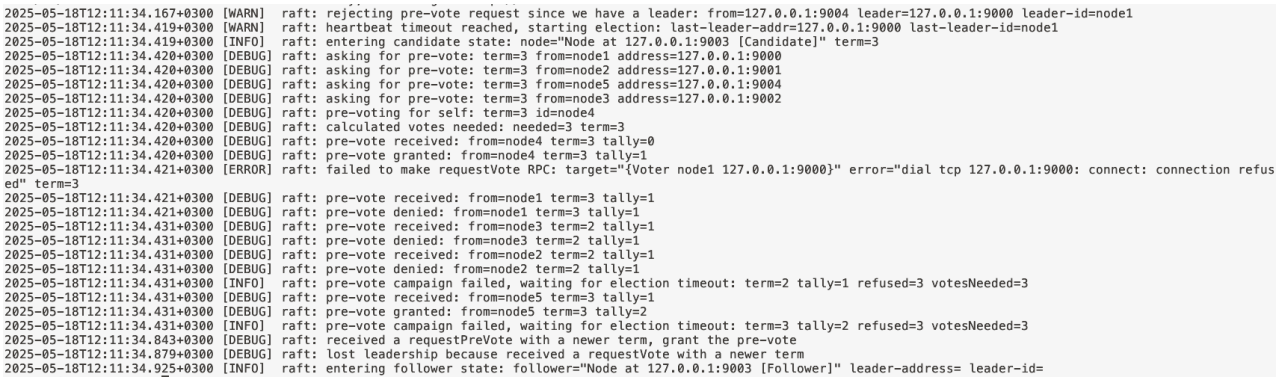

Рисунок 20 – Процесс выбора нового лидера со стороны ведомого узла

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 22 представлена работа с данными.

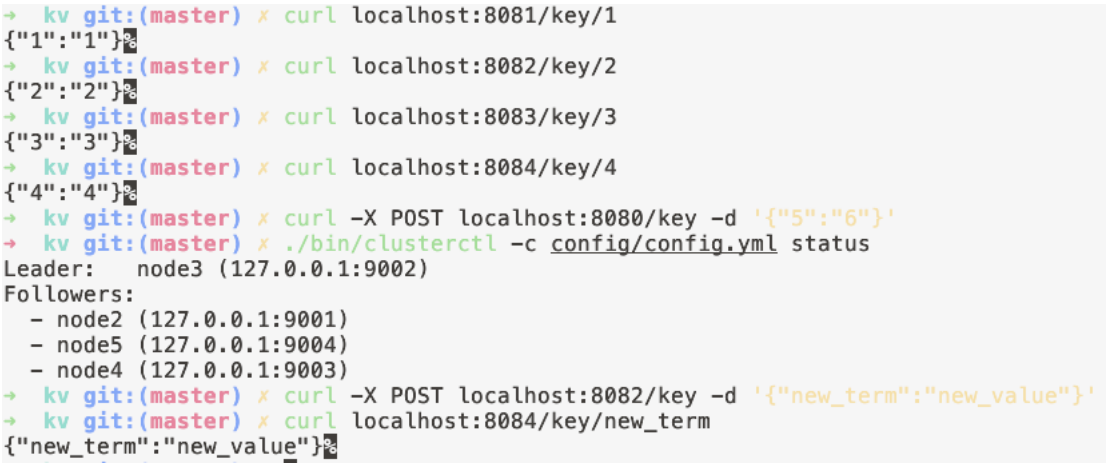

Рисунок 21 – Работа с данными в новом терме

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 22.

```
→  kv git:(master) ✗ ps -a | grep node
86329 ttys013    0:00.09 bin/node -id node2 -haddr 127.0.0.1:8081 -raddr 127.0.0.1:9001 -join 127.0.0.1:8080 var/node2
86330 ttys013    0:00.08 bin/node -id node3 -haddr 127.0.0.1:8082 -raddr 127.0.0.1:9002 -join 127.0.0.1:8080 var/node3
86331 ttys013    0:00.08 bin/node -id node4 -haddr 127.0.0.1:8083 -raddr 127.0.0.1:9003 -join 127.0.0.1:8080 var/node4
86332 ttys013    0:00.46 bin/node -id node5 -haddr 127.0.0.1:8084 -raddr 127.0.0.1:9004 -join 127.0.0.1:8080 var/node5
86506 ttys013    0:00.00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --exclude-dir=.idea --exclude-dir=
.tox --exclude-dir=.venv --exclude-dir=venv node
→  kv git:(master) ✗ ./bin/clusterctl -c config/config.yml stop
Stopped node2 (pid 86329)
Stopped node3 (pid 86330)
Stopped node4 (pid 86331)
Stopped node5 (pid 86332)
os: process already finished
Stopped leader node1 (pid 86309)
→  kv git:(master) ✗ ps -a | grep node
86577 ttys013    0:00.00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --exclude-dir=.idea --exclude-dir=
.tox --exclude-dir=.venv --exclude-dir=venv node
```

Рисунок 22 – 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.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ongaro D., Ousterhout J. In Search of an Understandable Consensus Algorithm (Raft) / Raft. — 2014. — URL: https://raft.github.io/raft.pdf ; [Электронный ресурс]. Дата обращения: 25.05.2025.

2. etcd-io. etcd/raft / GitHub. — 2025. — URL: https://github.com/etcd-io/raft ; [Электронный ресурс]. Дата обращения: 25.05.2025.

3. HashiCorp. HashiCorp Raft / GitHub. — 2025. — URL: https://github.com/hashicorp/raft ; [Электронный ресурс]. Дата обращения: 25.05.2025.

4. Baidu. Braft / GitHub. — 2025. — URL: https://github.com/baidu/braft ; [Электронный ресурс]. Дата обращения: 25.05.2025.

5. Apache. Apache Ratis / GitHub. — 2025. — URL: https://github.com/apache/ratis ; [Электронный ресурс]. Дата обращения: 25.05.2025.

6. TiKV. raft-rs / GitHub. — 2025. — URL: https://github.com/tikv/raft-rs ; [Электронный ресурс]. Дата обращения: 25.05.2025.

7. Google. Go Programming Language / Google. — 2025. — URL: https://go.dev/ ; [Электронный ресурс]. Дата обращения: 25.05.2025.

8. Иван М. KV — Distributed Key-Value Store with Raft Consensus / GitHub. — 2025. — URL: https://github.com/themilchenko/kv ; [Электронный ресурс]. Дата обращения: 25.05.2025.