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.

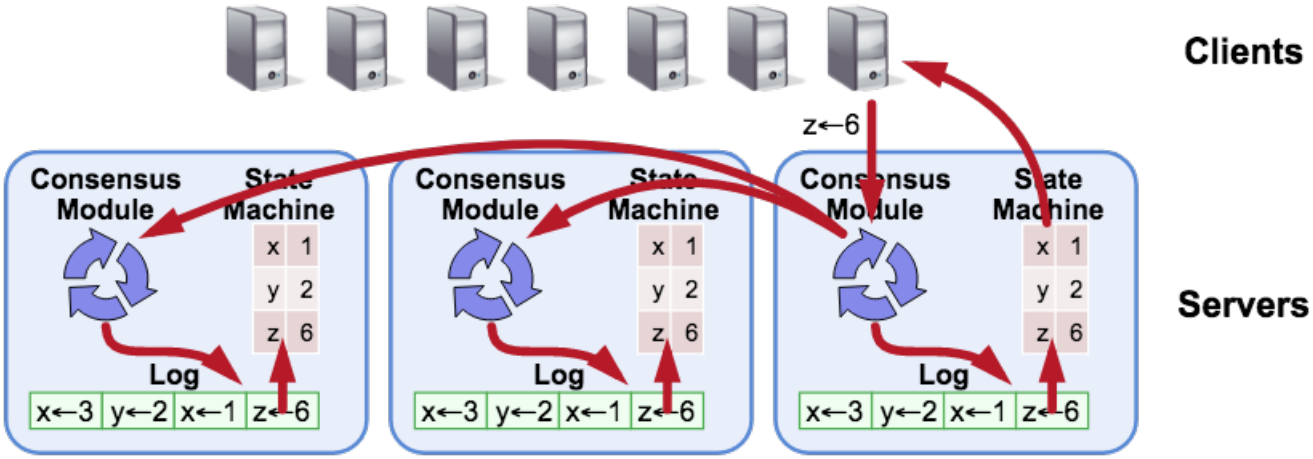

Рисунок 1 – Механизм репликации логов в алгоритме Raft

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.

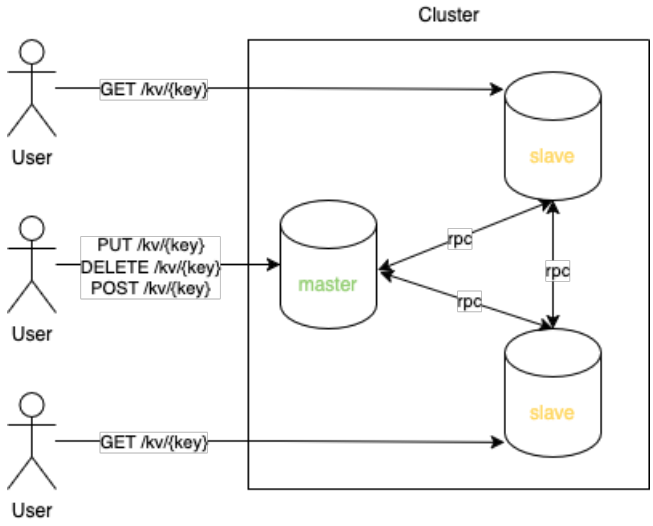

Рисунок 2 – Архитектура key-value хранилища

Клиенты взаимодействуют с системой через HTTP API. Операции записи (PUT, 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 (‘PUT‘, ‘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**
PUT	/kv/{key}	Master	Обновление значения, если он есть и создание его в случае отсутствия
DELETE	/kv/{key}	Master	Удаление ключа
GET	/kv/{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:

Листинг 1 – Пример конфигурационного файла для распределенного кластера

```
cluster:
  name: "raft-cluster"        # Названиекластера
  nodes:                      # Списоквсехузлов
    - id: "node-1"            # Уникальный ID узла
      address: "10.0.0.1:8080"
      role: "voter"           # Роль: voter/non-voter
    - id: "node-2"
      address: "10.0.0.2:8080"
```

```
      role: "voter"
    - id: "node-3"
      address: "10.0.0.3:8080"
      role: "voter"

raft:
  data_dir: "/var/lib/raft"  # Общиенастройки  Raft
  snapshot_interval: 1024
```