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)

Отчёт

по научно-исследовательской работе студента

на тему Разработка георасределённого механизма межкластерной репликации данных и обеспечения отказоустойчивости

ФИО студента:

Группа: ИУ8-114

Специальность:

Специализация:

Научный руководитель НИРС:

Работа выполнена: _____ (Дата) _____ (Подпись) Мильченко И. Д. (И. О. Фамилия)

Допуск к защите: _____ (Дата) _____ (Подпись) (И. О. Фамилия)

Дата защиты НИРС: _____

Результаты защиты: _____

Москва, 2025 г.

РЕФЕРАТ

Отчёт содержит 43 стр., 3 рис., 1 табл., 6 источн.

Ключевые слова: алгоритм Raft, распределённые системы, key-value хранилище, отказоустойчивость, репликация, георасределенные системы, межкластерная репликация, XDRC.

Данная работа посвящена разработке георасределённого механизма межкластерной репликации данных и обеспечению отказоустойчивости на уровне взаимодействия отдельных инсталляций распределённого хранилища. Исследование проводится в два этапа, каждый из которых соответствует отдельному модулю.

Первый модуль направлен на разработку механизма асинхронной репликации данных между изолированными кластерами (ЦОДами). Основная цель данного этапа — обеспечить передачу данных из активного кластера в пассивный без требования строгой согласованности в реальном времени.

Второй модуль посвящён разработке механизма отказоустойчивости и автоматического переключения. Основная задача этого этапа заключается в том, чтобы пассивный кластер мог корректно и надёжно перейти в активное состояние при недоступности основного, а также обеспечить гарантированное достижение консистентности перед переключением. На данном этапе рассматриваются вопросы обнаружения отказов, координации переключения ролей, а также безопасного завершения процесса репликации.

Итоговая система должна обеспечивать полноценную георасределённую репликацию и автоматическое восстановление работоспособности при сбоях.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные распределённые системы предъявляют всё более высокие требования к доступности данных, устойчивости к отказам и способности непрерывно функционировать при различных видах сбоев. По мере увеличения масштабов информационных систем и расширения их географического охвата становится необходимым использовать решения, обеспечивающие согласованность и доступность данных не только внутри одного кластера, но и между удалёнными дата-центрами. Такие механизмы особенно важны для критически важных сервисов, где недоступность данных в одном регионе не должна приводить к остановке работы всей системы.

Настоящая работа направлена на разработку и исследование георасределённого механизма межкластерной репликации данных, а также построение поверх него отказоустойчивой архитектуры с автоматическим переключением ролей кластеров. В отличие от типичных решений, ограниченных внутрикластерной репликацией, здесь рассматривается полноценное взаимодействие независимых кластеров, каждый из которых может функционировать автономно, но при этом обязан обеспечивать согласованное состояние данных при работе в составе распределённой системы.

Работа разделена на два функциональных модуля:

– Разработка механизма асинхронной репликации данных между кластерами. На этом этапе необходимо спроектировать и реализовать протокол обмена данными, обеспечивающий передачу снимков состояния и последовательности событий изменений (логов), а также корректное восстановление данных на стороне принимающего кластера. Особое внимание уделяется формату данных, стратегиям передачи, гарантии идемпотентного применения, а также взаимодействию компонентов при первичном и последующих запусках репликации.
– Реализация механизма отказоустойчивости и автоматического переключения между кластерами. Данный модуль дополняет разработанный ранее протокол репликации средствами мониторинга состояния кластеров, выбора текущего «активного» кластера и безопасного переключения при

возникновении отказов. В рамках этой части требуется реализовать алгоритмы определения лидирующего кластера, процессы перехода роли, а также обеспечить непротиворечивое продолжение репликации после переключения.

ОСНОВНАЯ ЧАСТЬ

Прежде, чем приступить к разработки архитектуры георасределенной системы, необходимо описать основные механизмы, благодаря которым происходит согласованность и целостность данных в базах данных и распределенных системах. Такими основными системами являются журнал предзаписи (Write-Ahead Log) и снимки текущего состояния системы (Snapshots). Более того, с помощью этих механизмов будет построена репликация между двумя кластерами, находящих в разных зонах доступности.

1 Теоретические основы репликации

Понятие "теоретических основ репликации"может включать в себя множество систем и алгоритмов, но в данном пункте будут рассмотрены только те механизмы, которые будут применяться дальше в работе. Чтобы понимать, за счет чего достигается целостность и согласованность данных, далее рассмотрим эти механизмы.

1.1 Журнал предзаписи

Журнал предзаписи (более известен, как Write-Ahead Log или сокращенно WAL) – это широко используемая техника обеспечения атомарности и долговечности изменений в системах управления базами данных и системах хранения данных. Идея WAL проста и эффективна: прежде чем применять изменение к основным файлам данных, система записывает описание этой операции в последовательный, устойчивый (на диске) журнал. Благодаря этому, при сбое и последующем восстановлении можно проиграть записи из журнала и восстановить согласованное состояние данных. WAL позволяет отделить момент подтверждения операции для клиента от фактической записи изменений в основной структуре данных, что даёт существенные преимущества по производительности и надёжности. Например, в такой популярной базе данных, как PostgreSQL [1], пишется: "WAL гарантирует, что изменения всегда сначала фиксируются в журнале, а затем применяются к основному хранилищу. Это позволяет системе пережить сбой, восстановив последнее согласованное состояние на основе журнала".

WAL естественным образом служит источником событий для репликации: реплицирующая сторона может получать записи из WAL и последовательно применять их к своему состоянию. Такой подход гарантирует, что порядок изменений сохраняется, а при наличии механизма снапшотинга (см. далее) при первом подключении можно передать снапшот состояния, а затем докатать события из WAL, начиная с индекса снапшота. Кроме того, WAL облегчает аварийное восстановление: после краша систему можно быстро восстановить путём пролистывания последних записей журнала и повторного применения операций.

Эти принципы проявляются во многих современных СУБД и стореджах и документированы в официальных материалах и статьях по архитектуре баз данных. Вот, как используется WAL в крупных базах данных:

- **PostgreSQL.** WAL является центральным механизмом обеспечения долговечности и репликации. Все изменения логируются в сегментированные WAL-файлы. WAL используется для потоковой репликации, point-in-time recovery (PITR) и логического декодирования [1].
- **etcd.** В etcd механизм WAL хранит Raft-записи (entries) и метаданные состояний консенсуса (term/index). WAL в etcd сегментирован и обеспечивает надёжное журналирование предложений (proposals) перед подтверждением репликации. Это подробно описано в документации [2].
- **Tarantool.** Tarantool использует WAL для персистентности in-memory-данных. Каждое изменение записывается в WAL-файл (например, `.xlog`); при рестарте система воспроизводит журнал для восстановления состояния. Документация Tarantool подробно описывает формат WAL и стратегию ротации [3].

Для более глубокого понимания того, как WAL обеспечивает согласованность и правильный порядок применения изменений, рассмотрим наглядный пример. На рисунке 1 показано, как три параллельно выполняющиеся транзакции формируют общий поток записей WAL.

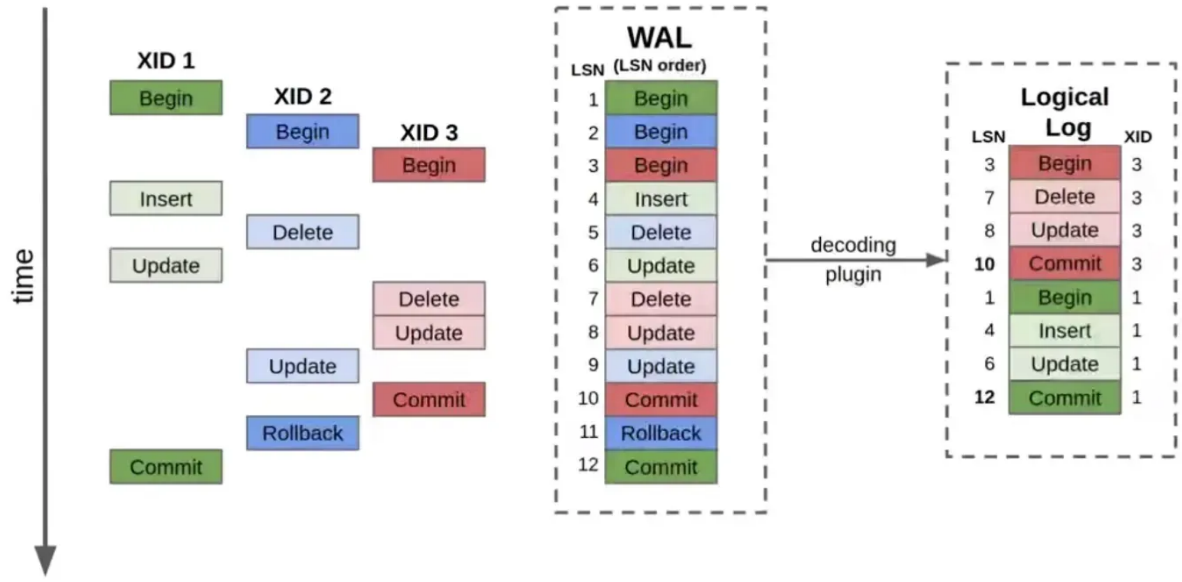

Рисунок 1 – Отношение между временем выполнения, идентификаторами транзакций (XID) и порядком записей в WAL (LSN) [4]

Этот пример демонстрирует важное свойство WAL: несмотря на асинхронность, параллельность и несовпадение порядка начала и завершения транзакций, все операции логируются в *строго линейной* последовательности (LSN), что делает возможным корректное восстановление и репликацию.

Рассмотрим основные наблюдения, иллюстрируемые диаграммой.

1. Транзакции начинаются в одном порядке, но завершаются — в другом. В примере транзакции получают идентификаторы XID по мере начала: сначала XID1 (зелёная), затем XID2 (синяя), затем XID3 (красная). Однако порядок завершения другой: первой коммитится красная транзакция, синяя откатывается (rollback), зелёная завершается последней.

Это подчёркивает важность ведения журнала: простой порядок по XID не отражает реального порядка изменений и не может использоваться напрямую для восстановления состояния.

2. WAL содержит записи даже незавершённых транзакций. Как видно из XID2 (синяя транзакция), в WAL попадают операции *до момента коммита* транзакции. Это сделано сознательно: PostgreSQL записывает изменения на диск по мере их поступления, чтобы ускорить коммит (atomic commit), избегая больших задержек.

Любая система, потребляющая WAL (включая движки репликации), обязана корректно обрабатывать такие случаи:

- в случае коммита — применять операции,
- в случае отката — игнорировать все изменения транзакции.

3. Порядок по LSN — глобален, но не всегда совпадает с границами транзакций. LSN присваивается каждому изменению в журнале. Однако:

- операции внутри транзакции могут быть разбросаны по WAL,
- транзакция может иметь несколько записей,
- откат (rollback) делает недействительными уже записанные изменения.

Например, операции XID2 с LSN 5 и LSN 9 в результате не должны быть применены, хотя находятся в середине WAL.

Это означает, что потребителю WAL (как и при межкластерной репликации) необходимо:

а) отслеживать границы транзакций (BEGIN / COMMIT / ROLLBACK);

б) применять только те операции, транзакции которых завершились успешно;

в) при откате — игнорировать все операции транзакции.

4. Метка времени (timestamp) не даёт гарантии порядка. Хотя каждая запись WAL имеет временную метку, Materialize подчёркивает, что она *не может использоваться* для определения корректной последовательности: в распределённых системах время ненадёжно, а WAL обеспечивает строгий порядок именно через LSN.

Таким образом, данный пример подчёркивает ключевую роль WAL как универсального и строгого источника истины при восстановлении и репликации. Для построения надёжного механизма межкластерной репликации необходимо воспроизводить логику фильтрации, порядка и атомарности транзакций, аналогичную поведение PostgreSQL и других промышленных систем.

Несмотря на широкое распространение, WAL не является универсальным решением: в ряде систем данные записываются непосредственно в долговечные структуры, используются альтернативные журнальные модели (например, MVCC или колонковые commit-logs). Кроме того, синхронная запись WAL может стать узким местом для высоконагруженных систем, поэтому на практике применяются различные компромиссы между надёжностью и производительностью.

Тем не менее WAL остаётся фундаментальным механизмом согласованности данных в распределённых системах и служит основой для построения эффективной межкластерной репликации, рассматриваемой в данной работе.

1.2 Снимки состояния

Несмотря на фундаментальную роль журнала предзаписи (WAL), только один WAL не может обеспечить эффективное и быстрое восстановление системы. Поскольку журнал имеет тенденцию расти без ограничений, его применение при рестарте или при подключении новой реплики может занять значительное время. Чтобы решить эту проблему, в большинстве систем управления данными используется механизм снимков состояния (более известный как snapshots).

Снимок представляет собой полное или частичное фиксированное состояние данных на некотором моменте времени, соответствующее определённому индексу (WAL LSN, Raft index или внутреннему checkpoint ID). Иными словами, снимок — это «контрольная точка» состояния, относительно которой WAL содержит только приращение изменений.

Механизм snapshot'ов решает сразу несколько задач:

а) **Сокращение времени восстановления.** Вместо применения потенциально гигабайтного WAL с начала времён, система восстанавливается из снимка, а затем воспроизводит только небольшую «хвостовую» часть журнала.

б) **Ограничение роста WAL.** После создания снимка и переноса состояния на стабильный носитель, часть старых сегментов журнала может быть безопасно удалена или зарезервирована в архиве. Это критически важно для систем, работающих без остановки годами.

в) **Ускорение начальной синхронизации реплики.** Новые узлы в кластере (или новые географически распределённые реплики) могут быстро загрузить снимок и затем догнаться по журналу, что значительно ускоряет подключение.

г) **Основная единица консистентного состояния.** Снимок представляет собой замороженную версию данных, согласованную по всем внутренним структурам; на нём можно строить репликацию, резервное копирование и тестовые окружения.

Механизм snapshot'ов используется практически во всех современных СУБД и системах репликации:

- В **PostgreSQL** снимки создаются через механизм checkpoint, который фиксирует состояние страниц данных и синхронизирует его с WAL. Хотя PostgreSQL не использует термин “snapshot” в контексте полного образа данных, checkpoint выполняет аналогичную роль: после него можно удалить часть старых WAL-сегментов [**pgsql-wal**].
- В **etcd** снимок является ключевым элементом реализации Raft. Он содержит сериализованное состояние FSM (Finite State Machine), а WAL уменьшается путём удаления записей, предшествующих индексу snapshot'а. Документация etcd подробно описывает формат snapshot-файлов и их восстановление [2].
- В **Tarantool** существует два вида файлов: `.snap` (снимки состояния данных) и `.xlog` (журналы транзакций). Репликация строится по принципу “snapshot + xlog tail”, что позволяет эффективно добавлять реплики и запускать кластер после аварии [3].

Таким образом, в реальных системах snapshot служит критически важным механизмом, который дополняет WAL и обеспечивает не только согласованность, но и управляемость размеров журнала.

С точки зрения теории реплицируемых состояний (replicated state machines), снимок фиксирует состояние системы после применения всех команд из журнала до некоторого индекса N. Это означает:

$$\text{Состояние после применения WAL[1..N]} \equiv \text{Snapshot}(N)$$

Тогда для восстановления состояния узла достаточно выполнить:

$$\text{Restore} = \text{LoadSnapshot}(N) \ + \ \text{Replay WAL}[N+1..\text{end}]$$

2 Архитектура георасперделенной системы

После изучения важнейших механизмов репликации необходимо перейти к продумыванию архитектуры. Высокоуровнево, задача заключается в том, чтобы реплицировать все данные из одной зоны доступности (кластера) в другую.

2.1 Выскоуровневое представление архитектуры

Говоря о межкластерной репликации, необходимо ввести несколько терминов:

- Кластер называется активным, если все его лидер-узлы осуществляют запись, а реплики (ведомые узлы) применяют их вслед за лидером.
- Обратно, кластер называется пассивным, если все его узлы отвечают только на запросы чтения. Следовательно, на пассивный кластер накладывается ограничение, которе заключается в том, что лидер-узлы не в праве принимать запросы на запись.

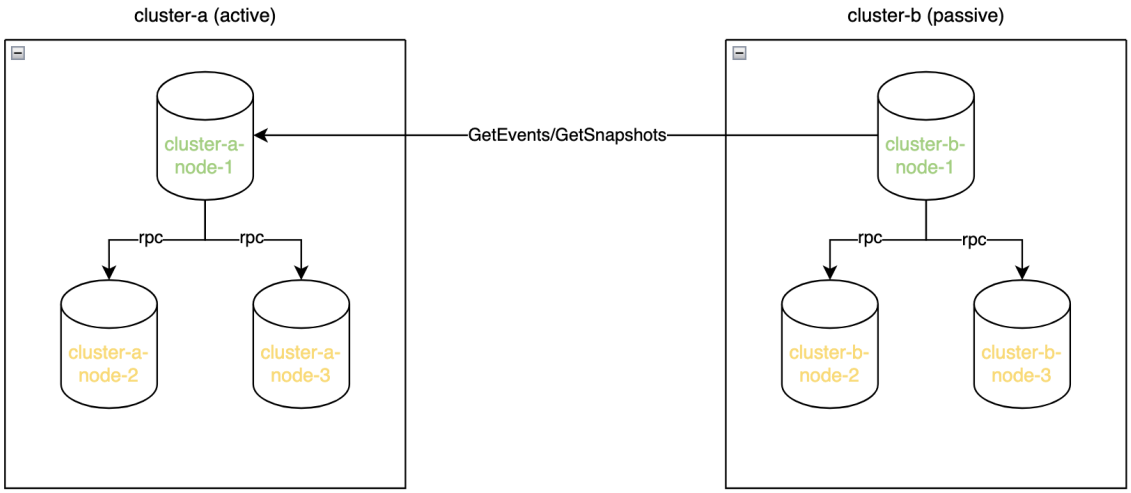

Рисунок 2 – Высокое представление георасперделенной системы

На рисунке 2 представлена схема межкластерной георасперделенной системы, состоящей из активного ($cluster\_a$) и пассивного ($cluster\_b$) кластеров. В состав активно и пассивного кластеров входят 3 узла соответственно: $cluster\_a\_node\_1$ (лидер), $cluster\_a\_node\_2$, $cluster\_a\_node\_3$ у активно-

го и $cluster\_b\_node\_1$ (лидер), $cluster-b-node-2$, $cluster-b-node-3$ у пассивного. Внутри кластеров имеются связи, подписанные rpc (remote procedure call), что показывает взаимодействие узлов по бинарному протоколу удаленного вызова.

Чтобы получить нужные записи из активного кластера, лидеру из пассивного ($cluster\_b\_node\_1$) необходимо найти мастера из активного кластера и подписаться на его обновления. Так как лидер $cluster\_a\_node\_1$, подключение происходит к нему. Если межкластерная репликация запускается впервые, то из мастера активного кластера сперва вычитается и передается снапшот, который применятеся на лидер-узле пассивного кластера и далее по rpc протоколу на весь кластер. Если снапшот был получен ранее, то подписка происходит за счет прочтения новых записей из журнала WAL, начиная с последнего номера, называевым Log Sequence Number (LSN), который персистентно сохраняется отдельно на пассивном кластере.

Таким образом, кластеру-последователю необходимо всегда хранить последнее состояние, которое ему удалось среплицировать с активного и применить на собственном. Назовем число такого состояния $last\_applied\_lsn$ Чтобы данные не терялись при перезапуске системы, $last\_applied\_lsn$ должен также писаться на диск, аналогично остальным данным, которые пишутся пользователем.

Протокол, по которому будет идти репликационный поток между кластерами – gRPC. Выбор именно этого протокола обусловлен быстротой и эффективностью, а также возможностью создать единый поток, по которому будут ходить репликационные пакеты. Это решение эффективнее по сравнению с наивной реализацией, где на каждую запись вызывался бы новый запрос.

2.2 API межкластерного взаимодействия

Как было сказано выше, для сетевого взаимодействия между класетрами используется проткол gRPC [5].

Было выненсено два метода: *GetSnapshots* и *GetEvents*. Первый метод нужен для инициализации пассивного кластера, когда он подключатеся к активноому кластеру впервые. Как было сказано ранее, вычитывать снимок эффективнее, чем проигрывать журнал предзаписи. Конечно, его польза значима только в случае, когда активный кластер хранит достаточное количество данных, которое проигрывалось бы из WAL дольше, нежели единым снимком.

Для языка Go был написан protobuf (protocol buffer) [6] файл, представленный на листинге 1, который содержит интерфейс взаимодействия кластеров.

Листинг 1 – Описание интерфейса общения кластеров на языке Protobuf

```
syntax = "proto3";

package replica;

import "google/protobuf/empty.proto";

option go_package = "github.com/themilchenko/kv";

service Replica {
  rpc GetSnapshots (google.protobuf.Empty) returns (stream Event)
     {}
  rpc GetEvents (State) returns (stream Event) {}
}

message Event {
  uint64 lsn   = 1;
  string id    = 2;
  bytes  data  = 3;
}

message State {
  uint64 lastAppliedIndex = 1;
}
```

Метод *GetSnapshots* не содержит аргументов, так как он вычитывает снапшот-файл, и отдает поток репликационных событий. Выбор потока обусловлен улучшением производительности системы.

Репликационное событие имеет тип *Event*. Оно содержит следующие поля:

- *lsn* (тип uint64) – log sequence number текущего события;
- *id* (тип string) – ключ хранилища;
- *data* (тип string) – непосредственное значение ключа хранилища.

Метод *GetEvents* принимает в качестве аргумента *State*, который включает в себя номер (lsn) последнего примененного на пассивном кластере события. Аналогично получению снапшотов, *GetEvents* возвращает поток репликационных событий.

2.3 API взаимодействия пользователя с системой

Программный интерфейс приложения дает возможность пользователю взаимодействовать с кластерами. Как было сказано ранее, активный кластер примает запросы на запись и чтение, а пассивный только на чтение. Полный HTTP API описан в таблице 1.

Таблица 1 – HTTP API геораспределённого KV-хранилища

Метод	**URL**	**Доступный кластер**	**Описание**
GET	/key/{key}	Любой узел любого состояния кластера	Получение значения по ключу. На пассивном кластере доступно только чтение.
POST	/key/{key}	Лидер-узел активного кластера	Создание или обновление значения ключа. Доступно только на активном кластере.
DELETE	/key/{key}	Лидер-узел активного кластера	Удаление ключа. Доступно только на активном кластере.
GET	/keys?limit={n}	Любой узел любого состояния кластера	Возвращает отсортированный список всех записей в хранилище (key–value), ограниченный параметром `limit`.
GET	/status	Любой узел любого состояния кластера	Получение текущего состояния кластера: лидер, последователи, идентификатор текущего узла.
POST	/join	Внутренний вызов для присоединения узла ко кворуму кластера	Добавление узла в кластер. В теле запроса передаются `id` и `addr` нового узла.

По большей части HTTP API представляет собой CRUD операции над кластером, а также операции по получению статуса кластера и возможность присоединения узла ко кворуму внутри кластера.

2.4 Выбор библиотек

Для реализации внутренних механизмов консенсуса и межкластерной репликации использовались библиотеки, обеспечивающие формальную корректность, устойчивость к сбоям и совместимость с архитектурными требованиями проекта.

В основе работы каждого кластера лежит библиотека `hashicorp/raft`, представляющая промышленную реализацию алгоритма Raft. Её использование обусловлено следующими факторами:

- наличие полного соответствия спецификации Raft (лидерство, репликация лога, снапшоты, восстановление);
- предоставление чётко определённых интерфейсов `FSM`, `LogStore` и `SnapshotStore`;
- устойчивая эксплуатация в больших проектах (Consul, Nomad), что подтверждает её корректность и зрелость модели.

Выбор готовой реализации исключает необходимость самостоятельной разработки алгоритма консенсуса и гарантирует предсказуемое поведение кластера.

Для хранения журнала предзаписи используется `hashicorp/raft-wal`, реализующая интерфейс `raft.LogStore`. Данная библиотека обеспечивает: сегментированное хранение WAL-файлов, надёжное восстановление состояния после сбоя, последовательный доступ к записям, необходимый для реализации механизма межкластерной репликации.

Совместимость с исходной реализацией Raft позволяет использовать WAL одновременно как механизм персистентности данных внутри кластера и как источник репликационных событий.

2.5 Проектирование конфига

Для корректного развёртывания георасределённой системы требуется единый формат конфигурации, определяющий структуру кластера, параметры отдельных узлов и сведения о межкластерных связях. Конфигурационный файл описывается в формате YAML и включает как локальные параметры кластера (адреса узлов, директории хранения), так и информацию, необходимую для организации межкластерной репликации.

Конфигурация для активного и пассивного кластеров имеет одинаковую структуру, различаясь только значениями полей `cluster_status` и `follow_list`. Основные элементы конфигурации перечислены ниже.

- **data_dir** — директория для хранения данных узлов кластера.
- **bin_path** — путь к каталогу, содержащему исполняемые файлы узлов. Используется вспомогательными инструментами CLI для запуска процессов.
- **cluster** — список, содержащий описание всех узлов одного кластера.
- **cluster_name** — логическое имя кластера, позволяющее различать активный и пассивный кластеры в конфигурациях межкластерной репликации.
- **cluster_status** — режим работы кластера:
  - `active` — кластер принимает операции записи и распространяет их среди своих узлов;
  - `passive` — кластер доступен только для чтения и получает изменения от активного кластера.
- **leader** — обозначение узла, который должен быть инициирован как лидер при первичном запуске. Используется только в сценарии bootstrap'а, после чего лидерство передаётся алгоритму Raft.
- **follow_list** — список gRPC адресов серверов сосднего кластера.

Секция `cluster` содержит перечень узлов и их адресов. Каждый узел описывается следующим набором параметров:

- **alias** — уникальный идентификатор узла внутри кластера.
- **http_address** — адрес, на котором узел обслуживает HTTP API (CRUD-операции пользователя, статус кластера).
- **rpc_address** — адрес, используемый механизмом консенсуса Raft для обмена служебными сообщениями между узлами текущего кластера.

- **grpc_address** — адрес, по которому узел предоставляет gRPC API для межкластерной репликации. Именно по этому адресу пассивный кластер подключается к активному.

Помимо описания локального кластера, конфигурация содержит два параметра, относящихся к межкластерному взаимодействию:

- **follow_list** — список gRPC-адресов всех узлов удалённого (противоположного) кластера. Пассивный кластер использует эти адреса для выбора лидера активного кластера по алгоритму round-robin и установления потока репликации. На активном кластере `follow_list` обычно указывает на пассивный, но используется только вспомогательными инструментами.
- **cluster_status**:
  - в активной конфигурации имеет значение `active`,
  - в пассивной конфигурации — `passive`, что означает обработку только операций чтения и обязательную репликацию из активного.

3 Демонстрация механизма асинхронной репликации между кластерами

3.1 Описание конфигурации активного и пассивного кластеров

Для демонстрации механизма межкластерной асинхронной репликации были развёрнуты два независимых кластера, каждый из которых содержит по пять узлов. Оба кластера имеют одинаковую внутреннюю топологию и различаются только ролью (*active* / *passive*) и адресным пространством.

Внутреннее устройство конфигурационных файлов полностью соответствует формату, описанному ранее, однако в рамках демонстрации их параметры имеют отдельное назначение.

На листинге 2 представлена конфигурация активного кластера. Отличающими опциями являются поле $cluster\_status$, которое задает стартовое состояние. Не смотря на то, что кластер активный, у него есть список $follow\_list$. Это сделано с расчетом на будущее проектирование механизма отказоустойчивости: если текущий кластер поменяет статус на противоположный, то он знал, к каким адресам ему подключаться для репликации.

Листинг 2 – Конфигурация активного кластера

```
data_dir: var
bin_path: bin
cluster:
  - alias: node1
    http_address: "127.0.0.1:8080"
    rpc_address: "127.0.0.1:9000"
    grpc_address: "127.0.0.1:9090"
  - alias: node2
    http_address: "127.0.0.1:8081"
    rpc_address: "127.0.0.1:9001"
    grpc_address: "127.0.0.1:9091"
  - alias: node3
    http_address: "127.0.0.1:8082"
    rpc_address: "127.0.0.1:9002"
    grpc_address: "127.0.0.1:9092"
  - alias: node4
    http_address: "127.0.0.1:8083"
    rpc_address: "127.0.0.1:9003"
```

```
    grpc_address: "127.0.0.1:9093"
  - alias: node5
    http_address: "127.0.0.1:8084"
    rpc_address: "127.0.0.1:9004"
    grpc_address: "127.0.0.1:9094"
leader: node1
cluster_status: active
cluster_name: "cluster1"
follow_list:
  - "127.0.0.1:9190"
  - "127.0.0.1:9191"
  - "127.0.0.1:9192"
  - "127.0.0.1:9193"
  - "127.0.0.1:9194"
```

Аналогично, на листинге 3 представлена конфигурация пассивного кластера. Отличия, как говорилось ранее, в указании адресов для прослушивания, а также в имене и его изначальном статусе.

Листинг 3 – Конфигурация пассивного кластера

```
data_dir: var
bin_path: bin
cluster:
  - alias: node1
    http_address: "127.0.0.1:18080"
    rpc_address: "127.0.0.1:19000"
    grpc_address: "127.0.0.1:19090"
  - alias: node2
    http_address: "127.0.0.1:18081"
    rpc_address: "127.0.0.1:19001"
    grpc_address: "127.0.0.1:19091"
  - alias: node3
    http_address: "127.0.0.1:18082"
    rpc_address: "127.0.0.1:19002"
    grpc_address: "127.0.0.1:19092"
  - alias: node4
    http_address: "127.0.0.1:18083"
    rpc_address: "127.0.0.1:19003"
    grpc_address: "127.0.0.1:19093"
  - alias: node5
    http_address: "127.0.0.1:18084"
    rpc_address: "127.0.0.1:19004"
```

```
    grpc_address: "127.0.0.1:19094"
leader: node1
cluster_status: passive
cluster_name: "cluster2"
follow_list:
  - "127.0.0.1:9090"
  - "127.0.0.1:9091"
  - "127.0.0.1:9092"
  - "127.0.0.1:9093"
  - "127.0.0.1:9094"
```

3.2 Старт кластеров

На листинге 4 показан процесс запуска двух кластеров с использованием утилиты clusterctl. Каждый кластер поднимается на основе собственного конфигурационного файла, после чего все узлы инициализируют локальные каталоги, Raft-компоненты и сетевые интерфейсы. Вывод демонстрирует корректный старт всех узлов и формирование рабочей топологии.

Листинг 4 – Старт кластеров с помощью утилиты clusterctl

```
$ ./bin/clusterctl -c config/cluster1.yml start

2025/11/19 22:09:14 Starting leader "node1"...
2025/11/19 22:09:14 Started "node1" (pid 41617)
2025/11/19 22:09:16 Starting follower "node2"...
2025/11/19 22:09:16 Started "node2" (pid 41626)
2025/11/19 22:09:16 Starting follower "node3"...
2025/11/19 22:09:16 Started "node3" (pid 41627)
2025/11/19 22:09:16 Starting follower "node4"...
2025/11/19 22:09:16 Started "node4" (pid 41628)
2025/11/19 22:09:16 Starting follower "node5"...
2025/11/19 22:09:16 Started "node5" (pid 41629)

$ ./bin/clusterctl -c config/cluster2.yml start

2025/11/19 22:09:21 Starting leader "node1"...
2025/11/19 22:09:21 Started "node1" (pid 41665)
2025/11/19 22:09:23 Starting follower "node2"...
2025/11/19 22:09:23 Started "node2" (pid 41672)
2025/11/19 22:09:23 Starting follower "node3"...
2025/11/19 22:09:23 Started "node3" (pid 41673)
```

```
2025/11/19 22:09:23 Starting follower "node4"...
2025/11/19 22:09:23 Started "node4" (pid 41674)
2025/11/19 22:09:23 Starting follower "node5"...
2025/11/19 22:09:23 Started "node5" (pid 41675)
```

На листинге 5 представлена структура файлов работы кластеров.

Листинг 5 – Структура системной директории кластеров

```
$ tree var

var/
  cluster1/
    node1/
      node.log
      node.pid
      snapshots/
      wal/
        00000000000000000001-0000000000000000.wal
        wal-meta.db
    node2/
      node.log
      node.pid
      snapshots/
      wal/
        00000000000000000001-0000000000000000.wal
        wal-meta.db
    node3/
      node.log
      node.pid
      snapshots/
      wal/
        00000000000000000001-0000000000000000.wal
        wal-meta.db
    node4/
      node.log
      node.pid
      snapshots/
      wal/
        00000000000000000001-0000000000000000.wal
        wal-meta.db
    node5/
      node.log
```

```
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db

cluster2/
  node1/
    node.log
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db
  node2/
    node.log
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db
  node3/
    node.log
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db
  node4/
    node.log
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db
  node5/
    node.log
    node.pid
    snapshots/
    wal/
      00000000000000000001-0000000000000000.wal
      wal-meta.db
```

Для каждого кластера формируется отдельная директория внутри var/, содержащая подпапки для узлов. Каждый узел имеет:

- журнал WAL;
- каталог снапшотов;
- PID-файл;
- лог работы.

Структура подтверждает корректную инициализацию подсистем WAL и snapshot для всех узлов обоих кластеров.

На листинге 10 представлен вывод команды *clusterctlstatus* для активного и пассивного кластеров. Листинг показывает лидера каждого кластера и список последовательных узлов (followers), что подтверждает стабильную работу алгоритма Raft и корректное формирование кворума в обоих кластерах.

Листинг 6 – Статус активного и пассивного кластеров соответственно

```
$ ./bin/clusterctl -c config/cluster1.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)

$ ./bin/clusterctl -c config/cluster2.yml status

Leader:   node1 (127.0.0.1:19000)
Followers:
  - node2 (127.0.0.1:19001)
  - node4 (127.0.0.1:19003)
  - node3 (127.0.0.1:19002)
  - node5 (127.0.0.1:19004)
```

На листинге 7 демонстрирует практическую работу репликации. Сначала в активный кластер вставляются десять пар ключ–значение с помощью HTTP-запросов. Затем выполняются запросы /keys как к активному, так и к пассивному кластеру. Оба ответа идентичны, что подтверждает успешный механизм межкластерной репликации: все изменения, произведённые в активном кластере, были корректно доставлены и применены в пассивном.

Листинг 7 – Заполнение активного кластера и демонстрация репликации на пассивный кластер

```
$ for i in $(seq 1 10); do
  curl -X POST localhost:8080/key -d "{\"key$i\":\"value$i\"}"
done

$ curl 'localhost:8081/keys?limit=10' | jq
  % Total    % Received % Xferd  Average Speed   Time    Time
     Time  Current
                                 Dload  Upload   Total   Spent
                                    Left  Speed
100   164  100   164    0     0  87560      0 --:--:-- --:--:--
   --:--:--  160k
{
  "key1": "value1",
  "key10": "value10",
  "key2": "value2",
  "key3": "value3",
  "key4": "value4",
  "key5": "value5",
  "key6": "value6",
  "key7": "value7",
  "key8": "value8",
  "key9": "value9"
}

$ curl 'localhost:18084/keys?limit=10' | jq
  % Total    % Received % Xferd  Average Speed   Time    Time
     Time  Current
                                 Dload  Upload   Total   Spent
                                    Left  Speed
100   164  100   164    0     0   177k      0 --:--:-- --:--:--
   --:--:--  160k
{
  "key1": "value1",
  "key10": "value10",
  "key2": "value2",
  "key3": "value3",
  "key4": "value4",
  "key5": "value5",
  "key6": "value6",
```

```
  "key7": "value7",
  "key8": "value8",
  "key9": "value9"
}
```

Листинг 8 содержит фрагмент внутреннего журнала пассивного лидера. В логе фиксируются: получение снапшота (если это первый запуск), подключение к gRPC-потоку, последовательное получение WAL-записей, применение каждого события к локальному хранилищу, обновление счётчика *last_applied*.

Листинг 8 – Лог репликации мастера пассивного кластера

```
2025-11-19 22:47:30.098628 I | [replica-pool] trying 127.0.0.1:9091
2025-11-19 22:47:30.100131 I | [replica-pool] 127.0.0.1:9091 not
   master, switching...
2025-11-19 22:47:30.100163 I | [replica-pool] trying 127.0.0.1:9092
2025-11-19 22:47:30.102160 I | [replica-pool] 127.0.0.1:9092 not
   master, switching...
2025-11-19 22:47:30.102191 I | [replica-pool] trying 127.0.0.1:9093
2025-11-19 22:47:30.104218 I | [replica-pool] 127.0.0.1:9093 not
   master, switching...
2025-11-19 22:47:30.104240 I | [replica-pool] trying 127.0.0.1:9094
2025-11-19 22:47:30.106313 I | [replica-pool] 127.0.0.1:9094 not
   master, switching...
2025-11-19 22:47:30.106338 I | [replica-pool] trying 127.0.0.1:9090
2025-11-19 22:53:52.911278 I | [replica] event 7 (42 bytes)
2025-11-19 22:53:53.411039 I | [replica] event 8 (42 bytes)
2025-11-19 22:53:53.420613 I | [replica] event 9 (42 bytes)
2025-11-19 22:53:53.439543 I | [replica] event 10 (42 bytes)
2025-11-19 22:53:53.455685 I | [replica] event 11 (42 bytes)
2025-11-19 22:53:53.480662 I | [replica] event 12 (42 bytes)
2025-11-19 22:53:53.497175 I | [replica] event 13 (42 bytes)
2025-11-19 22:53:53.513441 I | [replica] event 14 (42 bytes)
2025-11-19 22:53:53.533515 I | [replica] event 15 (42 bytes)
2025-11-19 22:53:53.552106 I | [replica] event 16 (44 bytes)
```

Этот лог подтверждает, что механизм репликации работает в потоковом режиме и не теряет события даже при многократных обновлениях.

4 Обеспечение отказоустойчивости

При размещении двух Raft-кластеров в разных зонах доступности требуется наделить систему свойством отказоустойчивости. Под этим понимается способность системы автоматически продолжать обработку запросов на чтение и запись при частичном или полном отказе одного из кластеров, без ручного вмешательства администратора, а также с минимальными потерями данных, обусловленными асинхронной природой межкластерной репликации.

4.1 Архитектура отказоустойчивости

В разрабатываемой архитектуре отказоустойчивость реализуется на уровне кластеров целиком (active/passive), а не отдельных узлов. В каждый момент времени ровно один кластер находится в активном состоянии и принимает операции записи, тогда как второй кластер функционирует в пассивном режиме и получает изменения посредством асинхронной межкластерной репликации (снапшоты и поток WAL-записей). Основной задачей механизма отказоустойчивости является корректное и безопасное переключение ролей кластеров при деградации или недоступности активного кластера.

Общая архитектура механизма отказоустойчивости представлена на рис. 3. На схеме показаны два Raft-кластера, развёрнутые в различных зонах доступности, а также внешний координатор состояния, используемый для предотвращения сценариев *split-brain*.

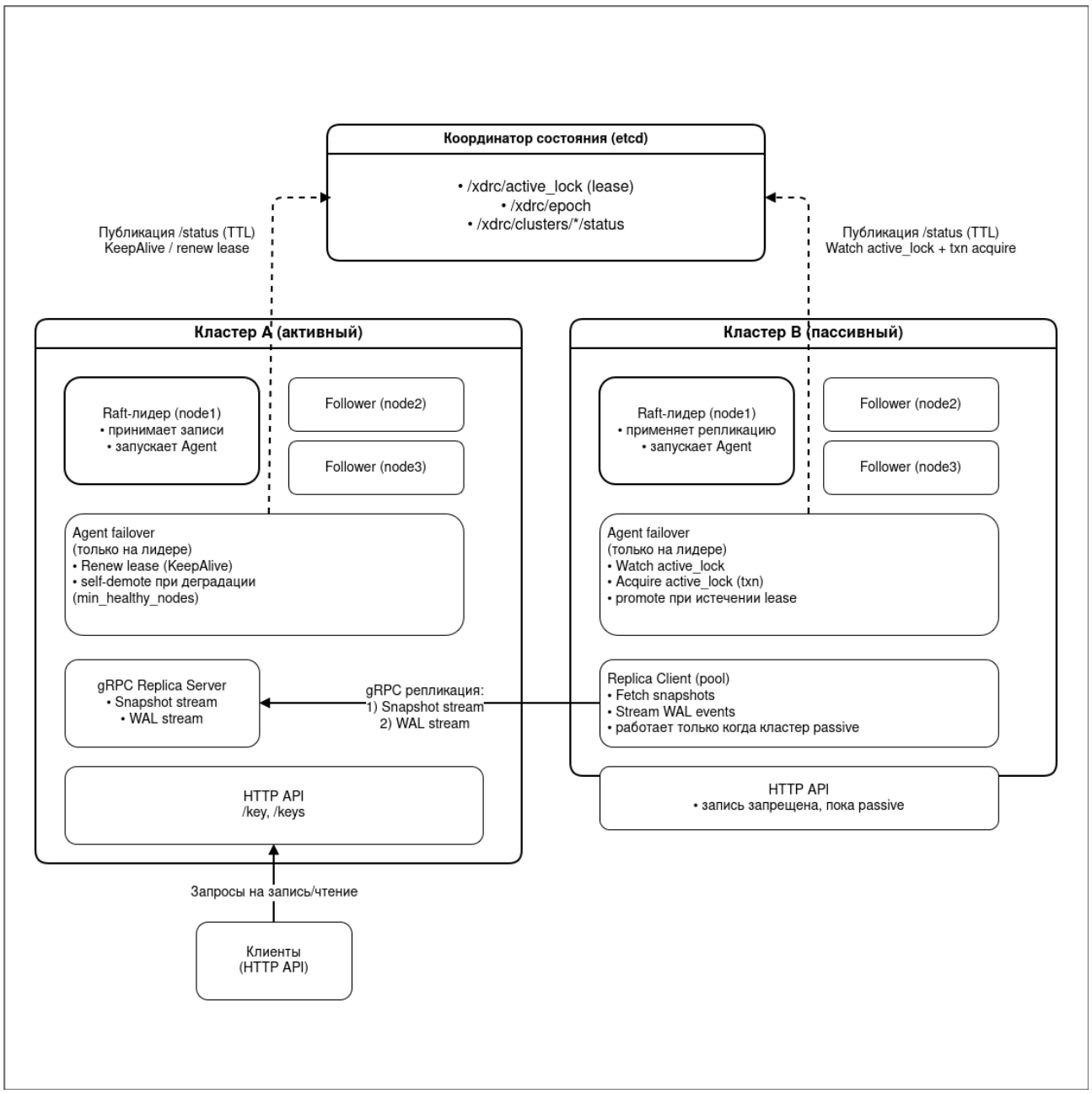

Рисунок 3 – Архитектура отказоустойчивости XDCR

Ключевой проблемой при проектировании подобной архитектуры является предотвращение сценариев *split-brain*, при которых оба кластера одновременно считают себя активными и принимают операции записи. Для исключения таких ситуаций в систему вводится внешний координатор состояния, реализованный на базе распределённого ключ-значение хранилища. В качестве координатора используется etcd, предоставляющий примитивы аренд (lease) и атомарных транзакций.

В координаторе хранится информация о текущем активном кластере в виде ключа `active_lock`, связанного с арендой с ограниченным временем жизни (`TTL`). Наличие данного ключа означает, что кластер, имя которого записано в значении, обладает правом принимать операции записи. При истечении аренды ключ удаляется автоматически, что используется в качестве триггера для переключения ролей кластеров.

Поддержание аренды активного состояния возлагается на лидера Raft внутри активного кластера. Как показано на рис. 3, на лидере запускается специализированный агент отказоустойчивости, который периодически выполняет продление аренды (`KeepAlive`) в координаторе. Тем самым лидер подтверждает свою работоспособность и доступность связности с другими узлами кластера (наличие кворума).

4.1.0.1 Деградация активного кластера.

В системе реализована политика, при которой активный кластер добровольно отказывается от своей роли (active $\rightarrow$ passive) при нарушении условий доступности. В частности, задаётся порог минимального количества «здоровых» узлов (`min_healthy_nodes`), при снижении ниже которого активный кластер считается деградировавшим. Агент отказоустойчивости, работающий на лидере, периодически оценивает доступность кластера на основе информации о подключённых Raft-пирах. При обнаружении деградации агент намеренно прекращает продление аренды `active_lock`. После истечения аренды ключ автоматически удаляется из координатора, и право на запись считается освобождённым.

Такой подход соответствует принципу *самодемотирования* активного кластера и существенно снижает риск ложных переключений, в том числе при сетевых разделениях, поскольку решение о потере статуса активного принимается самим кластером на основе локально наблюдаемого состояния, а не внешних эвристик.

4.1.0.2 Повышение роли пассивного кластера.

Пассивный кластер непрерывно отслеживает состояние ключа `active_lock` в координаторе. При обнаружении отсутствия ключа (истечение аренды) агент пассивного кластера инициирует процедуру повышения роли, пытаясь атомарно установить новый `active_lock` с помощью транзакции

etcd (операция «создать, если не существует»). В случае успеха кластер становится новым активным и начинает принимать операции записи. Одновременно направление межкластерной репликации изменяется: новый активный кластер становится источником изменений, а ранее активный кластер переходит в режим пассивного получателя данных.

4.1.0.3 Контроль согласованности при промоуте.

Поскольку межкластерная репликация носит асинхронный характер, при аварийном переключении возможна ситуация, когда пассивный кластер отстаёт от активного по объёму применённых изменений. Для ограничения возможных потерь данных предусмотрена опциональная политика `max_replication_lag`, ограничивающая допустимое отставание пассивного кластера на момент промоута. В случае превышения данного порога повышение роли не выполняется, и кластер продолжает ожидание, что используется в экспериментальной части работы.

4.1.0.4 Версионирование состояния (epoch).

Для обеспечения корректности при возврате ранее отказавшего активного кластера используется механизм версионирования состояния (epoch). При каждом успешном захвате `active_lock` значение epoch увеличивается и фиксируется в координаторе. Эпоха публикуется вместе со статусом кластера и служит маркером актуальности права на запись. Это гарантирует, что кластер, временно потерявший доступ к координатору, не сможет самопроизвольно возобновить приём записей после восстановления без явного подтверждения своей актуальной роли.

4.2 Ограничения и допущения

Следует отметить, что предложенный механизм обеспечивает отказоустойчивость на уровне кластеров, но не реализует прозрачную балансировку запросов записи внутри кластера. Операции записи корректно обрабатываются только Raft-лидером активного кластера; узлы-последователи возвращают отказ на запись (*not leader*) и требуют перенаправления запроса к лидеру. Данное поведение соответствует стандартной модели использования Raft и упрощает обеспечение согласованности при переключении ролей.

4.3 Конфигурация отказоустойчивости

Для управления поведением механизма отказоустойчивости в разработанной системе используется декларативная конфигурация, задаваемая в YAML-файле при запуске кластера. Такой подход позволяет гибко настраивать политику переключения ролей кластеров без изменения программного кода и адаптировать систему под различные эксплуатационные требования.

Конфигурация механизма отказоустойчивости логически выделена в отдельный раздел `failover` и применяется симметрично к каждому из Raft-кластеров. Пример фрагмента конфигурации приведён ниже:

Листинг 9 – Блок конфигурации отказоустойчивости системы

```
failover:
  enabled: true
  coordinator: etcd
  etcd_endpoints: ["127.0.0.1:2379"]
  lease_ttl_seconds: 10
  renew_interval_seconds: 3
  min_healthy_nodes: 3
  max_replication_lag: 1000
  promote_when_coordinator_only: true
```

Назначение параметров конфигурации следующее:

- `enabled` — включает или отключает механизм отказоустойчивости для данного кластера. При отключённом значении кластер функционирует в автономном режиме без взаимодействия с внешним координатором, что может использоваться для локального тестирования и отладки.
- `coordinator` — задаёт тип внешнего координатора состояния, используемого для предотвращения сценариев *split-brain*. В рамках данной работы используется значение `etcd`, отражающее применение распределённого ключ-значение хранилища с поддержкой аренд и атомарных операций.
- `etcd_endpoints` — список сетевых адресов узлов координатора состояния. Использование нескольких адресов повышает отказоустойчивость взаимодействия с координатором и снижает вероятность ложных переключений, вызванных временной недоступностью отдельного узла.

- `lease_ttl_seconds` — время жизни аренды `active_lock` в координаторе. Если активный кластер не продлевает аренду в течение данного интервала, ключ автоматически удаляется, что инициирует процедуру повышения роли пассивного кластера.
- `renew_interval_seconds` — периодичность продления аренды активным кластером. Значение параметра выбирается таким образом, чтобы в пределах одного TTL выполнялось несколько попыток продления, что повышает устойчивость системы к кратковременным задержкам и перегрузкам.
- `min_healthy_nodes` — минимально допустимое количество «здоровых» узлов в активном кластере. Под здоровыми узлами понимаются узлы, поддерживающие связность с Raft-лидером. При снижении фактического числа доступных узлов ниже заданного порога активный кластер считается деградировавшим и добровольно прекращает продление аренды `active_lock`, что приводит к автоматическому освобождению права на запись.
- `max_replication_lag` — максимально допустимое отставание пассивного кластера от активного по количеству применённых записей журнала. Параметр используется в качестве защитной политики при повышении роли: если отставание превышает заданное значение, промоут кластера не выполняется, что позволяет ограничить потенциальную потерю данных при аварийных переключениях.
- `promote_when_coordinator_only` — определяет, допускается ли повышение роли пассивного кластера в ситуации, когда координатор состояния временно недоступен или ключ `active_lock` отсутствует. Включение данного параметра повышает доступность системы, однако требует корректной настройки остальных параметров политики отказоустойчивости.

5 Экспериментальная проверка механизма отказоустойчивости

Целью экспериментальной части является практическая проверка корректности работы механизма отказоустойчивости, описанного в предыдущих разделах. В рамках экспериментов исследуется поведение системы при различных типах отказов активного кластера и подтверждается отсутствие сценариев *split-brain*.

В качестве базовой конфигурации используются два Raft-кластера, развёрнутые в различных зонах доступности, с включённым механизмом отказоустойчивости и внешним координатором состояния etcd. Параметры конфигурации выбираются одинаковыми для обоих кластеров, за исключением начального предпочтения роли.

5.1 Полный отказ активного кластера

В начальный момент времени оба кластера находятся в штатном состоянии. Для активного и пассивного кластеров проверяется состав Raft и наличие лидера. На листинге 10 показано, что кластер `cluster1` находится в активном состоянии и имеет одного лидера и двух последователей, аналогично для `cluster2`.

Листинг 10 – Проверка состояния Raft-кластеров

```
$ ./bin/clusterctl -c config/cluster1.yml status
Leader:   node1 (127.0.0.1:19000)
Followers:
  - node3 (127.0.0.1:19002)
  - node2 (127.0.0.1:19001)

$ ./bin/clusterctl -c config/cluster2.yml status
Leader:   node1 (127.0.0.1:29000)
Followers:
  - node3 (127.0.0.1:29002)
  - node2 (127.0.0.1:29001)
```

Далее проверяется состояние внешнего координатора. На листинге 11 видно, что ключ `active_lock` указывает на `cluster1`, а статусы кластеров соответствуют ролям active и passive. Таким образом, в системе корректно зафиксировано глобальное состояние отказоустойчивости.

Листинг 11 – Состояние координатора etcd до отказа

```
$ etcdctl get --prefix /xdrc
/xdrc/active_lock
cluster1
/xdrc/clusters/cluster1/status
{"cluster_name":"cluster1","role":"active","leader_id":"node1","
   last_applied_lsn":0,"epoch":1765748550953572409,"updated_at
   ":"2025-12-14T21:46:12.954705333Z"}
/xdrc/clusters/cluster2/status
{"cluster_name":"cluster2","role":"passive","leader_id":"node1","
   last_applied_lsn":5,"epoch":0,"updated_at":"2025-12-14T21
   :46:22.323099179Z"}
/xdrc/epoch
1765748550953572409
```

После этого выполняется тестовая операция записи в активный кластер. Запрос на запись успешно обрабатывается лидером активного кластера, а последующий запрос на чтение подтверждает сохранение данных. Репликация обеспечивает доставку изменений в пассивный кластер, что подтверждается результатами чтения на обоих кластерах (листинг 12).

Листинг 12 – Запись и проверка репликации данных

```
$ curl -X POST localhost:18080/key -d '{"1":"1"}'
$ curl localhost:18080/keys
{"1":"1"}
$ curl localhost:28080/keys
{"1":"1"}
```

Далее, на листинге 13 инициируется полный отказ активного кластера путём остановки всех его узлов. В результате Raft-лидер перестаёт функционировать и, как следствие, агент отказоустойчивости более не продлевает аренду ключа `active_lock` в координаторе.

Листинг 13 – Полный отказ активного кластера

```
$ ./bin/clusterctl -c config/cluster1.yml stop
```

После истечения времени жизни аренды координатор автоматически удаляет ключ `active_lock`. Пассивный кластер обнаруживает его отсутствие и инициирует процедуру повышения роли. На листинге 14 видно, что `cluster2` захватывает активное состояние, значение epoch увеличивается, и кластер начинает принимать операции записи.

Листинг 14 – Состояние координатора после failover

```
$ etcdctl get --prefix /xdrc
/xdrc/active_lock
cluster2
/xdrc/clusters/cluster2/status
{"cluster_name":"cluster2","role":"active","leader_id":"node1","
   last_applied_lsn":5,"epoch":1765748785324795018,"updated_at
   ":"2025-12-14T21:46:25.32630548Z"}
/xdrc/epoch
1765748785324795018
```

После переключения ролей ранее активный кластер запускается повторно. Несмотря на восстановление его узлов, он корректно стартует в пассивном режиме и не пытается самопроизвольно принять роль активного, поскольку ключ `active_lock` принадлежит другому кластеру с более актуальной эпохой. Перезапуск демонстрируется на листинге 15.

Листинг 15 – Перезапуск ранее активного кластера

```
$ ./bin/clusterctl -c config/cluster1.yml start
```

Запись данных в новый активный кластер выполняется успешно, что подтверждается результатом чтения. При этом данные автоматически реплицируются в пассивный кластер, включая ранее отказавший и восстановленный `cluster1` (листинг 16).

Листинг 16 – Запись после failover и проверка репликации

```
$ curl -X POST localhost:28080/key -d '{"2":"2"}'
$ curl localhost:28080/keys
{"1":"1","2":"2"}
$ curl localhost:18080/keys
{"1":"1","2":"2"}
```

Попытка выполнить операцию записи в пассивный кластер завершается отказом, что подтверждает корректное соблюдение модели active/passive и отсутствие сценария *split-brain* даже после восстановления ранее активного кластера (листинг 17).

Листинг 17 – Отказ записи в пассивный кластер

```
$ curl -X POST localhost:18080/key -d '{"3":"1"}'
cluster not active (not leader)
```

5.2 Деградация активного кластера при нарушении порога доступности

В начальный момент времени оба кластера функционируют штатно. Кластер `cluster1` является активным и состоит из трёх узлов, один из которых является лидером Raft. Пассивный кластер `cluster2` также полностью доступен. Это подтверждается выводом команд на листинге 18.

Листинг 18 – Начальное состояние кластеров

```
$ ./bin/clusterctl -c config/cluster1.yml status
Leader:   node1 (127.0.0.1:19000)
Followers:
  - node2 (127.0.0.1:19001)
  - node3 (127.0.0.1:19002)

$ ./bin/clusterctl -c config/cluster2.yml status
Leader:   node1 (127.0.0.1:29000)
Followers:
  - node3 (127.0.0.1:29002)
  - node2 (127.0.0.1:29001)
```

Состояние внешнего координатора подтверждает, что активным кластером является `cluster1`, а ключ `active_lock` корректно привязан к нему. Значение epoch соответствует текущему активному состоянию системы (листинг 19).

Листинг 19 – Состояние координатора перед деградацией

```
$ etcdctl get --prefix /xdrc
/xdrc/active_lock
cluster1
/xdrc/clusters/cluster1/status
{"cluster_name":"cluster1","role":"passive","leader_id":"node2","
   last_applied_lsn":0,"epoch":0,"updated_at":"2025-12-14T22
   :09:50.467697079Z"}
/xdrc/clusters/cluster2/status
{"cluster_name":"cluster2","role":"passive","leader_id":"node1","
   last_applied_lsn":0,"epoch":0,"updated_at":"2025-12-14T22
   :09:48.406810727Z"}
/xdrc/epoch
1765750126953280043
```

Выполняется тестовая операция записи в активный кластер. Запрос успешно обрабатывается, а данные реплицируются в пассивный кластер, что подтверждается результатами чтения (листинг 20).

Листинг 20 – Проверка записи и репликации перед деградацией

```
$ curl -X POST localhost:18080/key -d '{"1":"1"}'
$ curl localhost:18080/keys
{"1":"1"}
$ curl localhost:28080/keys
{"1":"1"}
```

Далее инициируется частичный отказ активного кластера путём принудительного завершения процесса Raft-лидера. Важно отметить, что кластер не теряет полностью работоспособность: оставшиеся узлы продолжают функционировать, и происходит переизбрание нового лидера. Тем не менее, число доступных узлов снижается ниже заданного порога `min_healthy_nodes = 3`, что иллюстрируется командами на листинге 21.

Листинг 21 – Завершение процесса лидера активного кластера

```
$ ps -aux | grep node1
milchen+ 451070 1.3 0.1 1740140 31584 pts/2 Sl 00:57 0:01 bin/node
   -id node1 -haddr 127.0.0.1:18080 -raddr 127.0.0.1:19000 -gaddr
   127.0.0.1:19091 -follow-nodes
   127.0.0.1:29091,127.0.0.1:29092,127.0.0.1:29093 -cluster-name
   cluster1 -active -failover-enabled -etcd 127.0.0.1:2379 -lease-
   ttl-seconds 10 -renew-interval-seconds 3 -min-healthy-nodes 3 -
   max-replication-lag 1000 -promote-when-coordinator-only var/
   cluster1/node1
$ kill 451070
```

После завершения процесса лидера кластер `cluster1` продолжает функционировать в составе двух узлов и успешно переизбирает нового лидера, что подтверждается выводом команды состояния (листинг 22). Таким образом, отказ носит частичный характер и не приводит к полной недоступности кластера.

Листинг 22 – Состояние активного кластера после потери узла

```
$ ./bin/clusterctl -c config/cluster1.yml status
Leader:   node2 (127.0.0.1:19001)
Followers:
  - node3 (127.0.0.1:19002)
```

Несмотря на сохранение Raft-лидерства, агент отказоустойчивости, работающий на лидере, фиксирует нарушение политики доступности: число «здоровых» узлов оказывается ниже заданного порога. В результате агент намеренно прекращает продление аренды ключа `active_lock` в координаторе. После истечения TTL ключ освобождается, и пассивный кластер инициирует процедуру повышения роли.

Состояние координатора после завершения переключения ролей показано на листинге 23. Активным становится кластер `cluster2`, значение epoch увеличивается, а ранее активный `cluster1` переходит в пассивное состояние, несмотря на свою частичную работоспособность.

Листинг 23 – Состояние координатора после деградации

```
$ etcdctl get --prefix /xdrc
/xdrc/active_lock
cluster2
/xdrc/clusters/cluster1/status
{"cluster_name":"cluster1","role":"passive","leader_id":"node2","
   last_applied_lsn":0,"epoch":0,"updated_at":"2025-12-14T22
   :09:53.467290881Z"}
/xdrc/clusters/cluster2/status
{"cluster_name":"cluster2","role":"active","leader_id":"node1","
   last_applied_lsn":0,"epoch":1765750191408043198,"updated_at
   ":"2025-12-14T22:09:51.410753678Z"}
/xdrc/epoch
1765750191408043198
```

После переключения ролей операции записи успешно принимаются новым активным кластером. Данные реплицируются в пассивный кластер, включая ранее деградировавший `cluster1`. Результаты чтения подтверждают согласованность данных в обоих кластерах (листинг 24).

Листинг 24 – Запись после деградации активного кластера

```
$ curl -X POST localhost:28080/key -d '{"2":"2"}'
$ curl localhost:28080/keys
{"1":"1","2":"2"}
$ curl localhost:18080/keys
{"1":"1","2":"2"}
```

Данный сценарий демонстрирует важное свойство предложенного механизма: кластер может добровольно отказаться от роли активного не только при полном отказе, но и при частичной деградации, когда формально остаётся работоспособным. Такое поведение позволяет предотвратить работу активного кластера в условиях недостаточного кворума и снижает риск нарушения согласованности данных, обеспечивая более предсказуемое и безопасное переключение ролей.

ЗАКЛЮЧЕНИЕ

В рамках данной научно-исследовательской работы была спроектирована и реализована георасределённая система межкластерной репликации данных (xdrc), ориентированная на использование Raft-кластеров и обеспечивающая отказоустойчивость на уровне кластеров целиком. В основе системы лежит модель active/passive, при которой в каждый момент времени операции записи допускаются ровно в одном кластере, а второй кластер получает данные посредством асинхронной межкластерной репликации.

Ключевым результатом работы является разработка механизма автоматического переключения ролей кластеров без ручного вмешательства администратора и без возникновения сценариев *split-brain*. Для этого был введён внешний координатор состояния, реализованный на базе распределённого ключ-значение хранилища etcd. Использование примитивов аренды (lease), атомарных транзакций и дополнительного механизма версионирования состояния (epoch) позволило гарантировать, что право на выполнение операций записи в системе всегда принадлежит не более чем одному кластеру.

В ходе работы была реализована и экспериментально подтверждена корректность работы механизма отказоустойчивости в двух принципиально различных сценариях: при полном отказе активного кластера и при частичной деградации, выражающейся в снижении числа доступных узлов ниже заданного порога. Особое внимание было уделено второму сценарию, в котором активный кластер остаётся формально работоспособным, однако добровольно отказывается от роли активного на основании локально наблюдаемой деградации. Такой подход снижает вероятность некорректной работы в условиях потери кворума и повышает предсказуемость поведения системы при отказах.

Реализация межкластерной репликации основана на передаче снапшотов состояния и потока WAL-записей по gRPC и обеспечивает согласованность данных между кластерами с учётом асинхронной модели доставки. Проведённые эксперименты показали, что после переключения ролей система сохраняет целостность данных, корректно обрабатывает возврат ранее отказавшего кластера и не допускает возобновления приёма записей кластером, утратившим актуальную эпоху.

Следует отметить ряд ограничений предложенного решения. Во-первых, система обеспечивает отказоустойчивость на уровне кластеров, но не реализует прозрачную балансировку клиентских запросов внутри кластера и между кластерами. Во-вторых, межкластерная репликация носит асинхронный характер, что допускает ограниченное отставание пассивного кластера и потенциальную потерю части последних записей при аварийном переключении. Эти ограничения осознанны и соответствуют выбранной архитектурной модели.

В качестве направлений дальнейшего развития системы можно выделить расширение политик принятия решений при промоуте кластеров (учёт задержек репликации, метрик нагрузки и сетевой доступности), интеграцию механизмов автоматического перенаправления клиентских запросов, а также формализацию модели отказов и анализ её свойств с использованием методов теории распределённых систем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Group P. G. D. Write-Ahead Logging. — 2024. — URL: https://www.postgresql.org/docs/current/wal-intro.html ; [Электронный ресурс]. Дата обращения: 16.11.2025.

2. Authors etcd. etcd: Write Ahead Log. — 2024. — URL: https://etcd.io/docs/v3.5/learning/wal/ ; [Электронный ресурс]. Дата обращения: 16.11.2025.

3. Team T. Tarantool WAL Subsystem. — 2024. — URL: https://www.tarantool.io/en/doc/latest/reference/reference_lua/box_ctl/wal/ ; [Электронный ресурс]. Дата обращения: 16.11.2025.

4. Materialize I. Connecting Materialize Directly to PostgreSQL via the Replication Stream. — 2023. — URL: https://materialize.com/blog/connecting-materialize-directly-to-postgresql-via-the-replication-stream/ ; [Электронный ресурс]. Дата обращения: 16.11.2025.

5. Google, Inc. gRPC Documentation. — 2024. — URL: https://grpc.io/ ; [Электронный ресурс]. Дата обращения: 16.11.2025.

6. Google, Inc. Protocol Buffers Developer Documentation. — 2024. — URL: https://protobuf.dev/ ; [Электронный ресурс]. Дата обращения: 16.11.2025.