МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

"До захисту допущено"

В.о. завідувача кафедри

_____ Ігор ШЕЛЕХОВ

_____червня 2023 р._____

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук

освітньо-професійної програми "Інформатика"

на тему: "Інформаційна технологія досягнення консенсусу розподілених систем на основі алгоритму Raft для мови програмування Rust"

здобувача групи ІН-94-1 Лещенка Івана Романовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Іван ЛЕЩЕНКО

(підпис)

Керівник,

старший викладач кафедри Борис КУЗІКОВ

комп'ютерних наук, к.т.н. _____

(підпис)

Суми – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

"Затверджую"

В.о. завідувача кафедри

_____ Ігор ШЕЛЕХОВ

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми "Інформатика"

здобувача групи ІН-94-1 Лещенка Івана Романовича

1. Тема роботи: "Інформаційна технологія досягнення консенсусу розподілених систем на основі алгоритму Raft для мови програмування Rust"
затверджую наказом по СумДУ від _____
2. Термін здачі здобувачем кваліфікаційної роботи до 09 червня 2023 року
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити): 1) опис проблеми. 2) порівняння існуючих імплементацій. 3) вибір інструментарію для розробки бібліотеки. 4) практична реалізація.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____
6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання “_____” _____ 20__ р.

Завдання прийняв до виконання _____ Керівник _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	Опис проблеми		
2	Порівняння існуючих імплементацій		
3	Вибір інструментарію для розробки бібліотеки		
4	Практична реалізація		

Здобувач вищої освіти _____ Керівник _____

АНОТАЦІЯ

Записка: 71 ст., 19 рис., 1 додаток, 43 літературних джерела.

Обгрунтування актуальності теми роботи - Обрана тема кваліфікаційної роботи є актуальною, оскільки в ній розглядаються методи спрощення процесу побудови розподілених додатків, які активно використовуються для вирішення практичних задач.

Об'єкт дослідження - Інформаційна технологія досягнення консенсусу розподілених систем на основі алгоритму Raft для мови програмування Rust.

Мета роботи - Розробка інформаційної технології з використанням мови програмування Rust для спрощення процесу створення розподілених додатків, складовою частиною якої є підтримка асинхронного обміну даними, можливість використання вбудованих середовищ та проста взаємодія зі скінченним автоматом протоколу розподіленого консенсусу Raft.

Методи дослідження - алгоритми побудови розподіленого консенсусу.

Результати - Було проведено комплексний аналіз існуючих рішень для створення розподілених додатків. За результатами дослідження було створено вимоги до бібліотеки, яка має на меті спростити інтеграцію протоколу розподіленого консенсусу Raft в додатки на мові програмування Rust. Кінцеве API бібліотеки надає користувачу можливість асинхронно взаємодіяти зі скінченним автоматом, при цьому підтримуючи інтеграцію з різними асинхронними середовищами. Обмін повідомленнями мережею та взаємодія зі сховищем даних імплементуються користувачем з врахуванням середовища в

якому буде виконуватися додаток, що робить бібліотеку гнучкою по відношенню до кінцевого середовища розгортання інфраструктури.

RAFT, RUST, АСИНХРОННІСТЬ, РОЗПОДІЛЕНІ ДОДАТКИ,
ХМАРНІ ОБЧИСЛЕННЯ

ЗМІСТ

ВСТУП

Актуальною проблемою на сьогоднішній день залишається створення розподіленного консенсусу між серверами. Через постійно зростаючі об’єми даних які необхідно зберігати та обробляти виникає необхідність створювати кластери, інформація в яких зберігається на декількох серверах одночасно. Завдяки такому підходу ми маємо можливість не тільки вертикального масштабування, тобто збільшення потужності вже існуючих серверів, а ще й горизонтального, який заснований на додаванні нових серверів.

Розподілення даних між декількома серверами дозволяє бути більш захищеними від надзвичайних ситуацій, оскільки у разі недоступності одного з серверів усі інші учасники кластеру продовжують обробляти запити. Відімкнення світла в датацентрі[1], пожежа[2,3], або пошкодження серверного обладнання через недбалість можуть спричинити проблеми для бізнесу, якому потрібна постійна обробка даних.

Створення географічно розпозподілених кластерів даних також надає можливість швидше обслуговувати клієнтів, оскільки затримки які виникають під час взаємодії з ресурсами зменшуються за рахунок використання найближчого до клієнта серверу.

Для вирішення проблеми розподіленого консенсусу існують різні протоколи (на кшталт Paxos[4] чи Raft[5,6]). Основне завдання цих протоколів - забезпечити цілісність даних та надати можливість безпечного редагування стану кластеру. Протоколи регламентують алгоритм, коректна імплементація якого надає змогу створювати кластери даних.

Метою проекту є створення бібліотеки яка виступає у ролі скінченного автомату протоколу Raft на мові програмування Rust[7]. Бібліотека повинна надати користувачам можливість безпечно та з мінімальними труднощами створювати власні програми з функціоналом розподілення даних.

Для досягнення мети дослідження сформульовані наступні задачі:

- Проаналізувати літературу за темою алгоритмів і методів досягнення розподіленого консенсусу.
- Проаналізувати існуючі імплементації протоколу Raft мовою програмування Rust, виділити їх переваги і недоліки.
- Реалізувати бібліотеку зі створення кластерів даних на основі алгоритму Raft для мови програмування Rust
- Розробити засоби перевірки якості реалізації розробленого рішення.

1 ІНФОРМАЦІЙНО-АНАЛІТИЧНИЙ ОГЛЯД

1.1 Формування розподіленого консенсусу

Як правило, розподілений консенсус формується за рахунок того, що учасники кластеру постійно обмінюються між собою інформацією про поточний стан сховища кожного з учасників кластеру та намагаються домовитися про використання одного й того самого стану. Правила того, як саме учасники кластеру домовляються між собою про узгодження єдиного стану відрізняються між різними протоколами розподіленого консенсусу.

Для забезпечення надійності кластеру, протоколи розподіленого консенсусу як правило зберігають поточний стан мережі між декількома учасниками кластеру, що дозволяє мережі функціонувати навіть у випадку, коли ноди стають недоступні. Оскільки подібна система є розподіленою, інколи можуть виникати конфліктні ситуації під час яких повідомлення від різних учасників кластеру мають інформацію про різні стани сховища даних. Протоколи розподіленого консенсусу повинні бути спроектованими так, щоб таких ситуацій виникало якомога менше, а у разі виникнення подібних конфліктів мати можливість їх вирішити, бажано автоматично.

1.2 Застосування протоколів дистрибутивного консенсусу

Основними споживачами подібних протоколів є хмарні сервіси та розподілені бази даних.

1.2.1 Amazon DynamoDB

DynamoDB це хмарна документна база даних розроблена усередині сервісу AWS.

Особливостями цієї бази даних є низька затримка виконання запитів та потужні можливості масштабування[8]. Функціонал масштабування є прихованим від користувача, хмарний сервіс автоматично виконує все необхідне для реплікації даних. Внутрішня архітектура DynamoDB була

продемонстрована на AWS re:Invent 2018[9], а потім опублікована на конференції USENIX ATC ’22[10].

Для розподілення бази даних DynamoDB використовує протокол Paxos, за допомогою якого побудовано кластер з трьох серверів. Один з трьох серверів обирається лідером (згідно з протоколу Paxos), який буде отримувати всі записи на мутацію сховища (рис. 1.1).

Рисунок 1.1 - Демонстрація процесу виконання запиту PutItem у DynamoDB[9]

Кожні ≈1.5 секунди лідер надсилає усім іншим серверам запит, головним завдання якого є повідомити інших учасників кластеру про те, що лідер все ще активний. Якщо такі запити не було отримано впродовж певного часу будь-який сервер отримує право стати лідером. Для переходу до ролі лідера будь-якому учаснику кластера необхідно досягти згоди з іншим сервером.

Сервери на яких зберігаються дані DynamoDB розподілені між різними зонами доступності AWS[11], що й дозволяє хмарному сервісу гарантувати високу доступність хмарного сервісу DynamoDB (рис. 1.2).

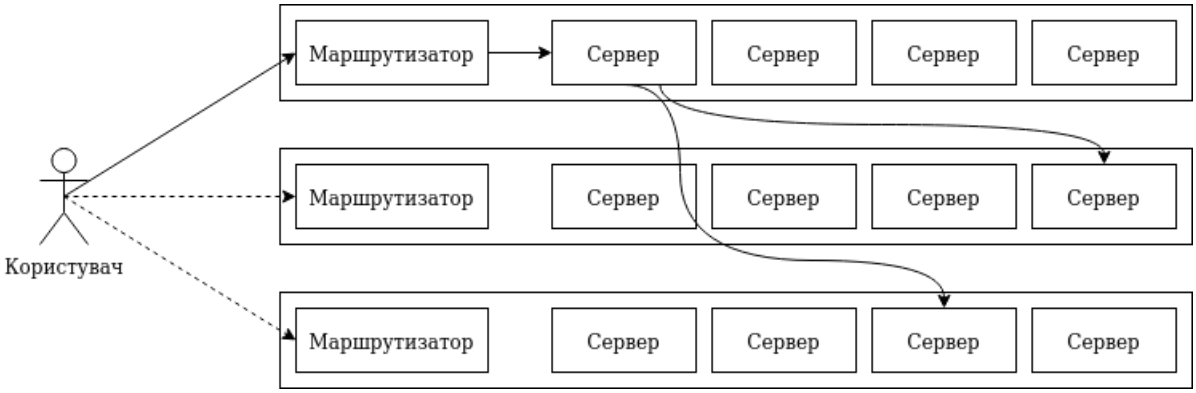


Рисунок 1.2 - Розподілення учасників кластеру DynamoDB між зонами доступності[9]

1.2.2 CockroachDB

CockroachDB є відкритою системою керування базами даних від Cockroach Labs[12]. Ця СКБД має потужний функціонал розподілення даних (рис. 1.3), що надає можливості до горизонтального масштабування та підвищеної надійності зберігання даних в цілому. CockroachDB має інтерфейс доступу до даних у вигляді діалекту мови SQL, що дозволяє вже існуючим проектам набагато швидше інтегрувати цю СКБД.

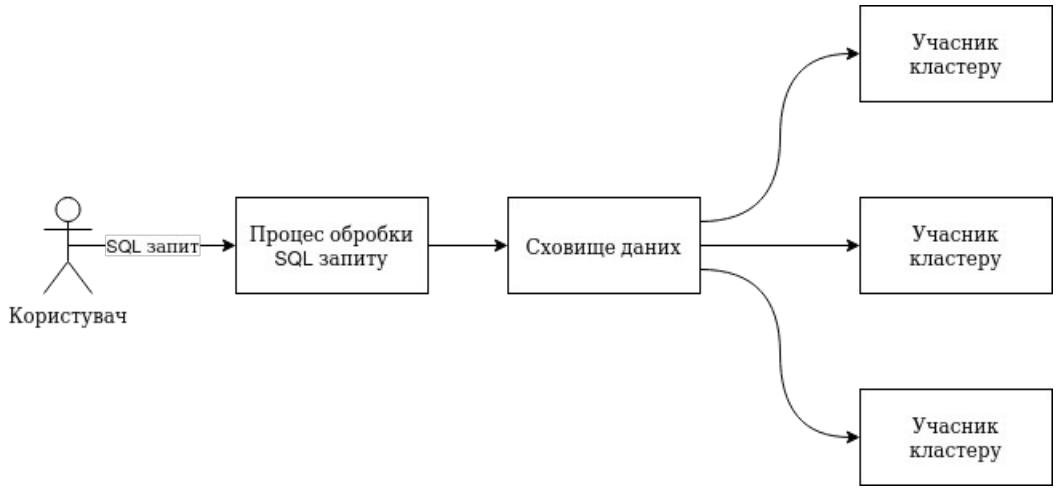


Рисунок 1.3 - Архітектура CockroachDB[13]

Розподілення даних у CockroachDB відбувається завдяки протоколу Raft[14]. Архітектурно система керування базами даних побудована пошарово[15]. Окремі шари виділені для SQL, транзакцій, сховища даних та, власне, реплікації.

Шар реплікації CockroachDB автоматично розподіляє дані між серверами усередині кластеру, забезпечуючи рівномірне навантаження на учасників.

Починаючи з версії 21.1, CockroachDB підтримує неголосуючих учасників кластеру[16]. Такі учасники не беруть участь у голосуваннях, і як результат не можуть стати лідерами в кластері чи віддати свій голос під час голосування, проте вони все ще отримують дані про записи від лідера, тим самим підвищуючи доступність даних усередині СКБД.

1.2.3 TiDB

TiDB є системою керування базами даних від компанії PingCAP[17]. Особливостями СКБД є сумісність з клієнтами MySQL, підтримка як транзакційних так і аналітичних режимів роботи та можливість горизонтального масштабування.

Підтримка горизонтального масштабування забезпечена за допомогою протоколу Raft, як і у CockroachDB (рис. 1.4). Бібліотека “raft”, яка використовується у цій СКБД та написана компанією PingCAP, є однією з перших повноцінних імплементацій протоколу Raft на мові програмування Rust.

Для зберігання даних TiDB використовує RocksDB - вбудовану базу даних “ключ-значення”, яка інкапсулює всі необхідні операції з підтримки сховища даних, надаючи користувачу простий інтерфейс керування записами.

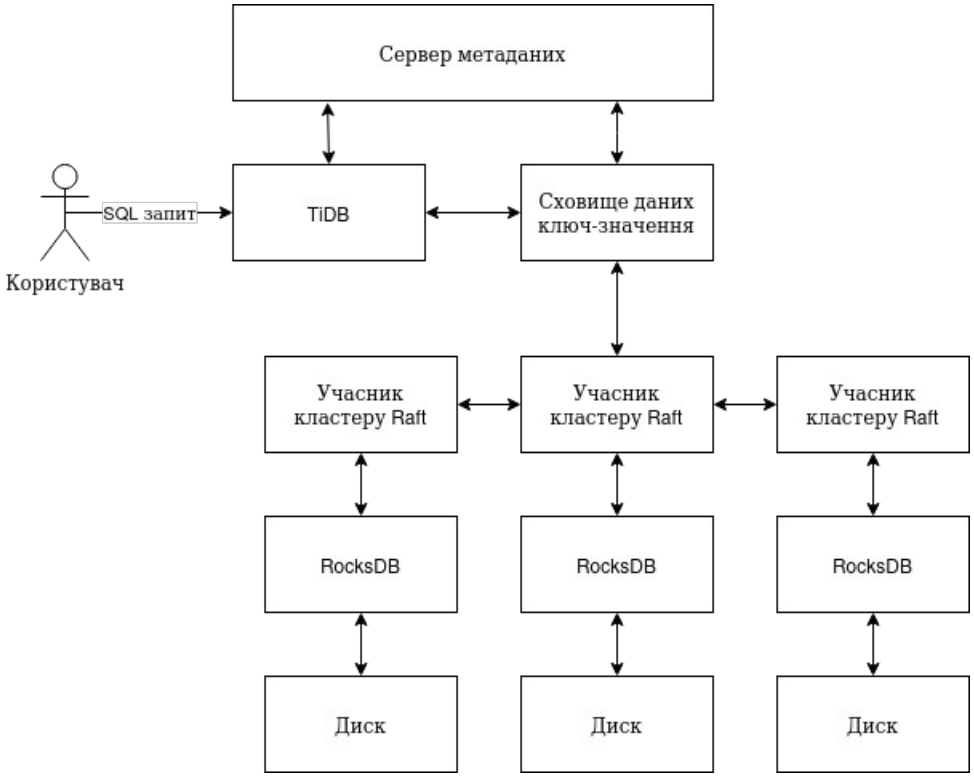


Рисунок 1.4 - Архітектура кластеру TiDB[18]

Для виконання складних аналітичних запитів TiDB має тонкий шар під назвою TiSpark, основна мета якого - активовувати Apache Spark над розподіленою OLTP базою даних, тим самим надаючи TiDB можливості виконання OLAP запитів (рис. 1.5).

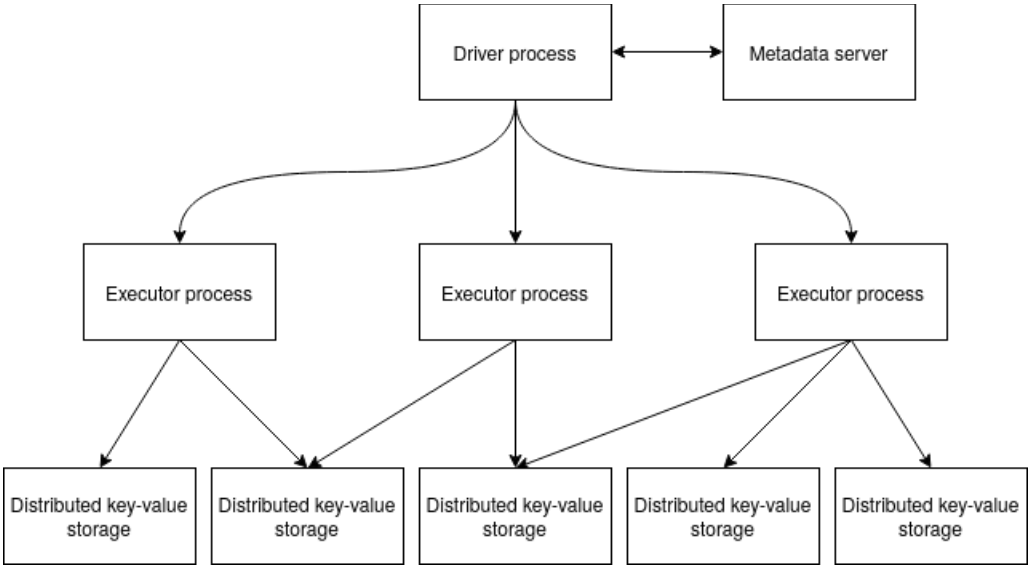


Рисунок 1.5 - Архітектура TiSpark[19]

1.2.4 etcd

etcd це key-value СКБД, яка надає користувачам простий інтерфейс з використанням протоколу gRPC, високу швидкість роботи та підтримку розподілення даних за допомогою протоколу Raft[20].

etcd написана на мові програмування Go. Бібліотека протоколу Raft, яка використовується усередині СКБД, створена спеціально для цього проекту[21].

Окремо варто виділити функціонал діагностування статусу кластеру Raft за допомогою вбудованого в СКБД інструментарію. В перелік доступних діагностик входить інформація про наявність лідера у кластері, загальна кількість записів відомих лідеру та інше[22].

1.2.5 ScyllaDB

ScyllaDB це NoSQL база даних сумісна з протоколом Apache Cassandra та DynamoDB[23]. Підтримка цих двох протоколів дозволяє вже існуючим додаткам скористатися перевагами ScyllaDB з відсутніми або мінімальними змінами вихідного коду.

Для підтримки консенсусу раніше використовувався Paxos, однак починаючи з версії 5.x ScyllaDB використовує Raft[24]. В документації зазначено, що рекомендована конфігурація кластеру складається з трьох датацентрів на один кластер, дозволяючи СКБД продовжити функціонувати у випадку недоступності одного з датацентрів[25].

1.3 Порівняння протоколів розподіленого консенсусу

Основною альтернативою протоколу Raft залишається сімейство алгоритмів Paxos.

Протокол розподіленого консенсусу Paxos використовується в розподілених системах для забезпечення узгодженості між серверами. Вперше опублікований у 1998 році, протокол Paxos був вдосконалений і модифікований у багатьох варіаціях щоб відповідати різним випадкам використання.

Незважаючи на наявність інших алгоритмів консенсусу, Paxos та його варіації залишаються актуальними і широко використовуваними і сьогодні.

Прикладами варіацій Paxos можуть слугувати наступні протоколи:

- Fast Paxos[26], який оптимізує кількість повідомлень необхідних для утворення консенсусу;
- Multi-Paxos[27], завдяки якому зменшується кількість необхідних фаз для створення консенсусу за умови наявності стабільного лідера;

1.4 Опис протоколу Raft

Протокол визначає загальну структуру системи Raft, взаємодію учасників кластеру між собою, процес обрання лідера кластеру, процес реплікації даних та вирішення конфліктів між нодами. Для цього специфікація включає в себе інформацію про ролі учасників кластеру та RPC, завдяки яким ноди спілкуються одна з одною (наприклад, для реплікації записів чи запитів на надання голосу під час виборів лідера).

Ноди з роллю лідера є основними нодами кластеру Raft. Ці ноди відповідають за ведення журналу даних, реплікацію журналу для інших нод та відстеження актуальності сховища учасників кластеру, надсилаючи дані за необхідності.

Звичайні ноди, в свою чергу, повинні обробляти запити лідера, тим самим підтримуючи актуальним стан власного сховища даних.

Роль кандидата необхідна для процесу виборів лідера усередині кластеру. Учасники кластеру з роллю кандидата переходять до ролі лідера у випадку набирання достатньої кількості голосів. Якщо достатню кількість голосів не було отримано, то кандидат повинен повернутися до ролі звичайної ноди.

1.4.1 Процес виборів лідера

У разі, якщо всередині кластеру відсутній лідер, звичайні ноди повинні з плином часу починати процес конверсії до кандидата (рис. 1.6). Якщо на протязі часу, який обирається випадково, не було отримано жодного запиту від

інших кандидатів чи лідера, то звичайна нода надає собі роль кандидата, збільшує значення поточного терміну голосування та надсилає всім іншим учасникам кластеру запит на отримання голосу.

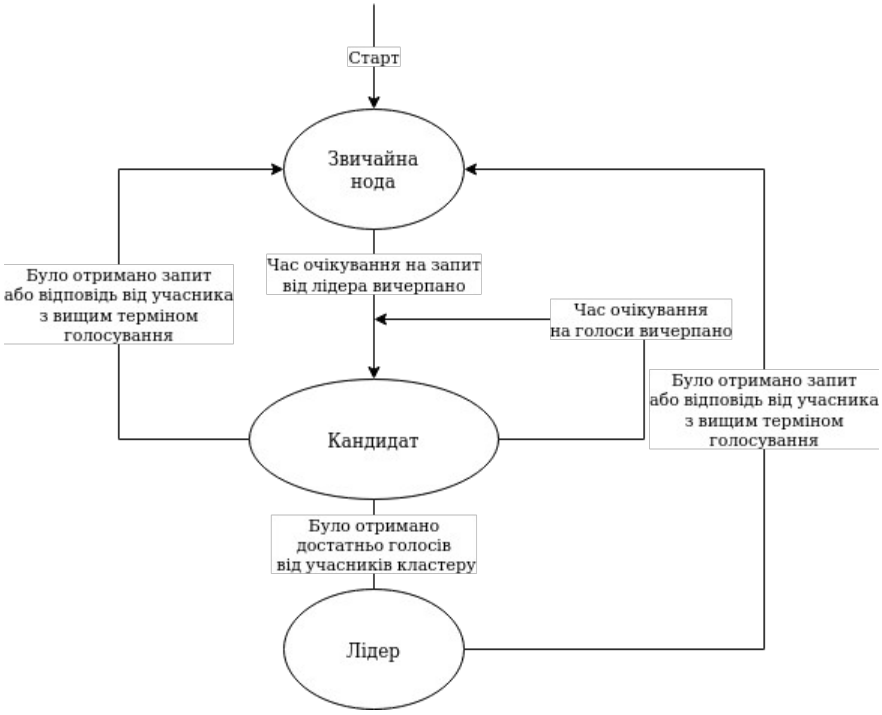


Рисунок 1.6 - Алгоритм голосування

Сервери які отримують запит від кандидата повинні вирішити чи віддавати свій голос в поточному терміні голосування за нього. Кожен сервер може голосувати лише один раз на протязі одного терміну голосування. Якщо кандидат має актуальний стан сховища даних та термін голосування вищий або еквівалентний відомому терміну голосування, то голос надається. У всіх інших випадках голос надано не буде.

Учасники кластеру не захищені від розділення мережі[28], через що може виникати ситуація коли кандидат надсилає запит на отримання голосу від лідеру. Лідери, стан сховища яких менш актуальний за стан сховища кандидатів зобов’язані перейти до ролі звичайної ноди. В усіх інших випадках такі запити повинні бути відхиленими.

Як тільки кандидат отримує голос від половини кластеру він переходить до ролі лідера, одразу надсилаючи серверам запит AppendEntries. Конкуруючі кандидати усередині кластеру зобов’язані перейти до ролі звичайної ноди як тільки вони отримують актуальний запит AppendEntries від нового лідера кластеру.

1.4.2 Реплікація сховища даних лідера

Після завершення процесу голосування новий лідер починає відправляти іншим учасникам кластеру запити AppendEntries, завдяки яким ноди оновлюють інформацію у власному сховищі даних, актуалізуючи його за наявності раніше невідомих записів (рис. 1.7).

Лідер зобов’язаний відстежувати актуальність сховища даних для кожного учасника окремо, надсилаючи у запитах AppendEntries лише ту інформацію, яка необхідна для актуалізації сховища іншого учасника кластеру, що дозволяє мінімізувати трафік між лідером та звичайними нодами.

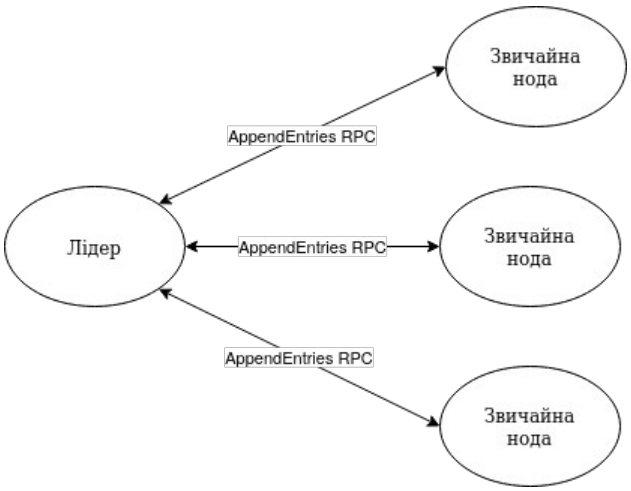


Рисунок 1.7 - Схема реплікації сховища даних лідера

Імплементація лідера повинна містити в собі таймер, інтервали якого використовуються для надсилання запитів AppendEntries іншим учасникам кластеру. Кожен запит індивідуальний та включає виключно ті записи, ідентифікатор яких старший за приблизний номер запису асоційований з сервером до якого надсилається запит AppendEntries.

Кожен сервер який отримав запит AppendEntries повинен проаналізувати його. Аналіз полягає у тому, що звичайна нода перевіряє еквівалентність останнього запису в своєму сховищі даних з першим записом який було передано усередині запиту AppendEntries. Якщо співпадають ідентифікатори та терміни голосування під час яких були створені записи, уся інша передана інформація з запиту AppendEntries зберігається в сховищі даних звичайної ноди, а таймер кандидата зкидується. Якщо співпадають лише ідентифікатори, то сервер який отримав запит спочатку видалити застарілі записи з власного сховища даних, перезаписавши їх використовуючи дані з запиту AppendEntries. Запит повинен бути відхилений у разі виявлення помилок під час аналізу.

Отримавши позитивну відповідь від звичайної ноди лідер повинен зберегти ідентифікатор останнього переданого запису до серверу. Цей ідентифікатор потім буде використано для розрахування підтверджених записів та для надсилання нових запитів AppendEntries.

У разі отримання негативної відповіді від звичайної ноди, лідер повинен знизити ідентифікатор останнього переданого запису до серверу на 1. З плином часу буде виконана повторна спроба відправити запит AppendEntries, але вже з оновленим переліком записів, який враховує останнє оновлення ідентифікатору переданого запису.

Цей процес повторюється постійно, доки не виникнуть будь-які проблеми з лідером (рис. 1.8). В такому випадку кластер на основі протоколу Raft почне процес виборів нового лідера.

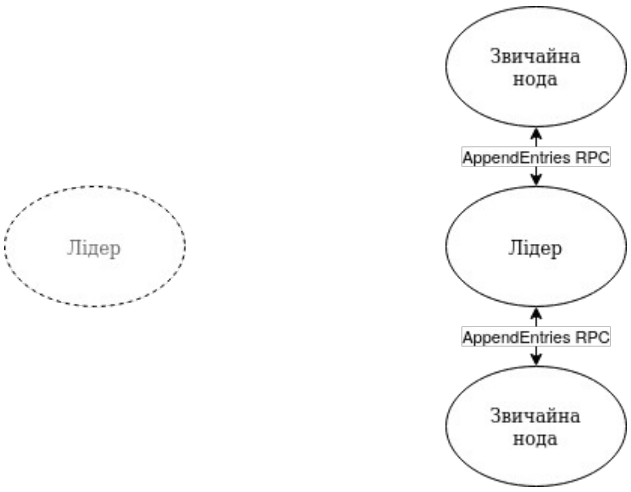


Рисунок 1.8 - Демонстрація від’єднання лідера від кластеру Raft

1.4.3 Структури RPC

Обмін даними між серверами відбувається з використанням RPC. Специфікація протоколу Raft[5] надає чіткий опис змісту запитів та відповідей на запити.

1.4.3.1 AppendEntries RPC

AppendEntries RPC використовується лідером для надання інформації про нові записи іншим учасникам кластеру Raft.

Запит AppendEntries містить у собі інформацію про:

- поточний термін голосування лідера;
- останній запис відомий серверу, який отримав запит (використовується для валідації запиту);
- нові записи, які сервер повинен додати до сховища даних у разі успішної валідації;
- ідентифікатор підтвердженого запису;

Після того як звичайна нода отримує запит AppendEntries, вона повинна перевірити актуальність наданої інформації за наступним алгоритмом:

1. Термін голосування лідера повинен бути не менший за той, про який відомо звичайній ноді;

2. Лідер та звичайна нода повинні мати однакові дані щодо останнього відомого запису;

Звичайна нода повинна відхилити запит, якщо надана лідером інформація є неактуальною, при цьому передавши лідеру поточний термін голосування.

Відповідь на AppendEntries містить в собі інформацію про:

- поточний термін голосування звичайної ноди;
- успішність виконання запиту;

1.4.3.2 RequestVote RPC

RequestVote RPC використовується кандидатом для отримання голосів від інших учасників кластеру під час виборів лідера.

Запит RequestVote містить в собі інформацію про:

- поточний термін голосування кандидата;
- останній запис відомий кандидату;

Після отримання запиту RequestVote, звичайна нода повинна перевірити актуальність наданої інформації за наступним алгоритмом:

1. Термін голосування кандидата повинен бути не менший за той, про який відомо звичайній ноді;
2. Дані про останній відомий запис кандидата повинні бути актуальними для звичайної ноди;
3. Ідентифікатор кандидата який отримав голос звичайної ноди під час останнього голосування повинен співпадати з ідентифікатором ноди, яка відправила запит, або бути пустим;

Запит повинен бути відхилений звичайною нодою у випадку, коли будь-яка з перелічених перевірок не пройшла успішно.

Відповідь на запит RequestVote містить в собі інформацію про:

- поточний термін звичайної ноди;
- успішність виконання запиту, тобто надання голосу кандидату;

1.5 Огляд імплементацій Raft

Бібліотеки наведені в цьому розділі можуть бути використані для створення власних розподілених додатків за допомогою протоколу Raft. Основною метою створення подібних бібліотек є надання користувачам можливостей для формування консенсусу, при цьому приховуючи деталі роботи протоколу за абстракціями.

Оскільки для комунікації з іншими учасниками кластеру використовуються мережеві протоколи, деякі бібліотеки протоколу Raft інтегруються зі стеком libp2p[29]. Модульний мережевий стек libp2p надає готовий набір абстракцій для комунікації використовуючи мережу, водночас надаючи кінцевому користувачу готові імплементації протоколів мережевого обміну даними та виявлення нових учасників кластеру. Автоматичний пошук нових нод відкриває можливості для автоматичного адміністрування кластером Raft.

1.5.1 HashiCorp Raft

Імплементація протоколу Raft від HashiCorp є найпопулярнішою імплементацією для мови програмування Go.

Ця бібліотека надає готові імплементації мережевого транспорту для інтеграції у розподілені додатки та тестування, що значно спрощує процес розробки.

Обробка помилок реалізована звичним для мови програмування Go способом, що дозволяє досвідченим розробникам простіше обробляти помилки які можуть виникнути під час формування консенсусу, тим самим забезпечуючи відмовостійкість додатку.

Сама бібліотека тестується в різних сценаріях за допомогою окремого фреймворку, який дозволяє простим способом повторювати різні стани кластеру Raft.

1.5.2 go-libp2p-raft

go-libp2p-raft є абстракцією над вищезазначеною бібліотекою Raft від HashiCorp. Ця бібліотека надає користувачу всі можливості бібліотеки libp2p (захищені канали обміну даними, мультиплексування протоколів, створення з'єднань з врахуванням NAT, тощо).

1.5.3 raft-rs

raft-rs є однією з перших імплементацій протоколу розподіленого консенсусу Raft для мови програмування Rust[30]. Бібліотека була розроблена для використання всередині транзакційної бази даних "ключ-значення" TiKV.

Рушієм скінченного автомату Raft виступає метод `Raft::tick`[31] який користувач повинен самотужки викликати. Це суттєво відрізняється від інших імплементацій Raft для мови програмування Rust, які зазвичай використовують асинхронні середовища для реагування на зовнішні події кластеру Raft (таймери, запити від клієнта або від інших учасників кластеру).

Сховище яке повинен надати користувач для використання бібліотеки від TiKV є блокуючим[32], через що також можуть виникнути проблеми у асинхронних додатках. Виклики блокуючих методів усередині асинхронних середовищ значно впливають на ефективність усієї системи[33].

1.5.4 async-raft

async-raft є асинхронною імплементацією Raft для мови програмування Rust[34].

Типажі які описують мережу та сховище даних мають асинхронні методи, що надає можливість конкурентно обробляти події кластеру Raft та будь-які інші події у додатку користувача (запити до серверу, читання з диску, тощо).

Для взаємодії зі скінченним автоматом бібліотека надає структуру `async_raft::raft::Raft`. Завдяки цій структурі користувач може надавати інформацію про нові запити від інших серверів усередині кластеру чи про нові

запити на зміну стану кластеру (зміна інформації про кількість серверів усередині кластеру, додавання нового запису до сховища).

Варто зазначити, що async-raft підтримує лише одне середовище виконання асинхронного коду - Tokio[35]. Це робить неможливою інтеграцію у проектах без підтримки стандартної бібліотеки Rust (мікроконтролери та нові платформи, які ще не отримали підтримки стандартної бібліотеки), або де async-std[36] є основним середовищем виконання асинхронного коду.

1.5.5 Сукупне порівняння імплементацій

Для зручності порівняння наведемо узагальнення розгляду бібліотек реалізації протоколу Raft у вигляді таблиці (табл. 1.1).

Таблиця 1.1 - Порівняння імплементацій протоколу Raft

	HashiCorp Raft	go-libp2p-raft	raft-rs	async-raft
Мова програмування	Go	Go	Rust	Rust
Інтеграція з libp2p	-	+	-	-
Підтримка async/await (для Rust)			-	+

Результат розгляду показав, що незважаючи на те, що кожна з бібліотек є готовим продуктом, всі вони мають певні вади. У розділі 1.6 розглянемо вимоги до бібліотеки-реалізації протоколу Raft мовою програмування Rust, що забезпечать її конкурентні переваги у порівнянні з наявними аналогами.

1.6 Існуючі проблеми

1.6.1 Підтримка асинхронних додатків

Скінченний автомат Raft повинен виконувати будь-які операції лише за наявності зовнішніх подій (таймери, нові запити від клієнту або інших учасників кластеру).

В традиційній моделі синхронного програмування потік блокується в очікуванні завершення операцій вводу/виводу. Це може призвести до неефективного використання системних ресурсів, оскільки потік, заблокований на вводі/виводі, не може бути використаний для інших задач (рис. 1.9).

Рисунок 1.9 - Синхронне виконання коду

Водночас, модель асинхронного програмування дозволяє виконувати інші задачі під час виконання операцій вводу/виводу, що значно підвищує ефективність додатку в цілому (рис. 1.10). Для ефективного використання асинхронних середовищ бібліотеки повинні працювати з вводом/виводом асинхронно самостійно, або ж надавати способи імплементувати асинхронний код самостійно.

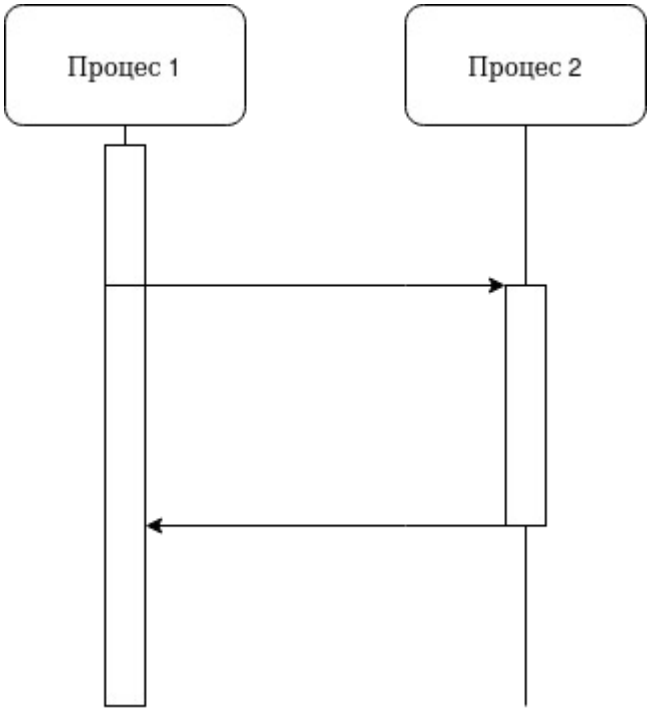


Рисунок 1.10 - Асинхронне виконання коду

1.6.2 Незалежність від асинхронного середовища

Заради максимальної гнучкості бібліотеки, вона повинна підтримувати будь-яке асинхронне середовище. Це стає можливим, оскільки бібліотека абстрагує роботу з файловою системою та мережею, що дозволяє користувачу використати імплементацію цих компонентів з власного асинхронного середовища.

1.6.3 Підтримка вбудованих проектів

Кінцева імплементація повинна підтримувати вбудовані середовища, які зазвичай не мають традиційної операційної системи.

Бібліотека повинна абстрагувати роботу з файловою системою чи мережею, тим самим дозволяючи користувачу імплементувати взаємодію з цими компонентами самостійно.

Водночас, бібліотека повинна мати доступ до динамічної алокації пам’яті, працюючи лише в додатках де наявний розподільник пам’яті.

1.6.4 Кінцевий інтерфейс

Для користувача повинен бути доступний максимально простий інтерфейс взаємодії зі скінченним автоматом Raft. Подібний інтерфейс повинен приховувати деталі імплементації від користувача, інкапсулюючи їх усередині самої бібліотеки.

1.7 Постановка задачі

Основна задача проекту - створити імплементацію протоколу дистрибутивного консенсусу Raft що відповідала б таким вимогам:

1.7.1 Сумісність з основною специфікацією

Кінцева імплементація проекту повинна відповідати специфікації Raft[5]. Це означає повну сумісність з наданими в специфікації деклараціями RPC та алгоритмами протоколу.

Відповідність протоколу повинна перевірятися автоматичним та ручним тестуванням. Тестування повинно враховувати ймовірність нестандартних ситуацій які можуть виникати у кластері Raft, перевіряючи їх за рахунок точного контролю плину часу та обміну даними між нодами.

1.7.2 Підтримка асинхронного виконання коду

Для створення мережевих додатків все більш популярною стає мова програмування Rust, яка завдяки потужним перевіркам під час компіляції коду та нативній підтримці async/await дає змогу розробникам писати безпечний та продуктивний код.

Асинхронне програмування дуже добре підходить для реалізації протоколу розподіленого консенсусу Raft, оскільки скінченний автомат в першу чергу керується подіями вводу/виводу (мережеві запити, таймери).

Екосистема асинхронних бібліотек для мови програмування Rust є доволі розвинутою та має усе необхідне для створення власних асинхронних додатків. Однак, нажаль, ця підтримка присутня не без проблем. Наприклад, досить часто бібліотеки замикаються на підтримці лише одного асинхронного середовища

(наприклад, Tokio), через що використання будь-якого іншого рантайму є проблематичним якщо взагалі можливим[37].

1.7.3 Незалежність від середовища виконання асинхронного коду

Для надання користувачу максимальної гнучкості, асинхронні бібліотеки мови програмування Rust повинні підтримувати роботу в будь-якому асинхронному середовищі. На даний момент не існує стандартного рішення проблеми того, що різні асинхронні середовища надають різні способи взаємодії з файлами або мережею, тому різні бібліотеки по різному вирішують цю проблему.

Наприклад, це може бути реалізовано завдяки перемиканню підтримуваних функцій під час компіляції або завдяки окремо виділеному типажу, який описує усі необхідні асинхронні операції.

Найпопулярніші бібліотеки Raft для Rust зав’язані на використанні асинхронного середовища Tokio.

1.7.4 Підтримка роботи без стандартної бібліотеки

Мова програмування Rust дозволяє створювати додатки для платформ, які не мають операційної системи. Подібні додатки працюють без використання стандартної бібліотеки, через що звична взаємодія з файловою системою чи мережею стає недоступною.

Водночас, підтримка async/await в Rust не залежить від стандартної бібліотеки, і є доступною навіть для вбудованих систем за наявності асинхронного середовища (наприклад, Embassy[38]).

Існуючі бібліотеки протоколу Raft для Rust зазвичай підтримують лише додатки зі стандартною бібліотекою, оскільки це значно спрощує процес розробки самої бібліотеки.

1.7.5 Публічне API

В мові програмування Rust є підтримка асинхронних потоків даних завдяки бібліотеці `futures`. Асинхронні потоки використовуються в цьому

проекті, оскільки вони надають користувачу зручний інтерфейс для обміну даними про мережеві події зі скінченним автоматом Raft.

Для використання сховища даних та таймерів користувачу буде запропоновано окремий типаж, який визначає асинхронні операції з запису даних у сховище та створення таймерів. Завдяки такому інтерфейсу з’являється можливість використовувати будь-які асинхронні середовища та значно простіше тестувати додаток, оскільки будь-яка з зазначених операцій може бути замінена на її тестову імплементацію.

1.7.6 Скінченний автомат

Згідно зі специфікацією протоколу розподіленого консенсусу Raft кожен учасник кластеру може мати лише одну роль одночасно, що можна виразити за допомогою перечислень мови програмування Rust.

Обмін даними з іншими учасниками кластеру повинен відбуватися також за рахунок перечислень Rust, оскільки таким чином можна виразити всі можливі мережеві події у системі типів, тим самим спрощуючи процес розробки та зменшуючи ризик некоректно використати бібліотеку для кінцевого користувача.

2 ПРОЕКТУВАННЯ БІБЛІОТЕКИ

2.1 Загальна структура проекту

Головною точкою входу проекту для кінцевого користувача є типаж `RaftStreamExt` (рис. 2.1). За допомогою цього типажу користувач може перетворити асинхронний потік (`futures::stream::Stream`[39]) вхідних подій Raft (рис. 2.2), імплементацію типажу `PersistentState` та унікальний ідентифікатор серверу у скінченний автомат протоколу Raft.

```
pub trait RaftStreamExt<NodeId, RequestId, PS: PersistentState<NodeId>>: FusedStream<Item = Incomir
    type IntoRaftEvents: FusedStream<Item = Result<OutgoingRaftNetworkEvent<NodeId, RequestId, PS::

    fn into_raft_events(
        self,
        node_id: NodeId,
        persistent_state: PS
    ) -> Self::IntoRaftEvents;
}
```

[–] Conversion of a `IncomingRaftNetworkEvent` stream into a Raft state machine.

Required Associated Types

```
type IntoRaftEvents: FusedStream<Item = Result<OutgoingRaftNetworkEvent<NodeId, RequestId,     source
PS::Contents>, PS::Error>>
```

Required Methods

```
fn into_raft_events(                                                                          source
    self,
    node_id: NodeId,
    persistent_state: PS
) -> Self::IntoRaftEvents
```

Convert a `FusedStream` of `IncomingRaftNetworkEvent` into a Raft state machine.

Params

`node_id` specifies a unique node identifier for the current node.

`persistent_state` specifies your custom `PersistentState` implementation.

Рисунок 2.1 - Опис типажу RaftStreamExt

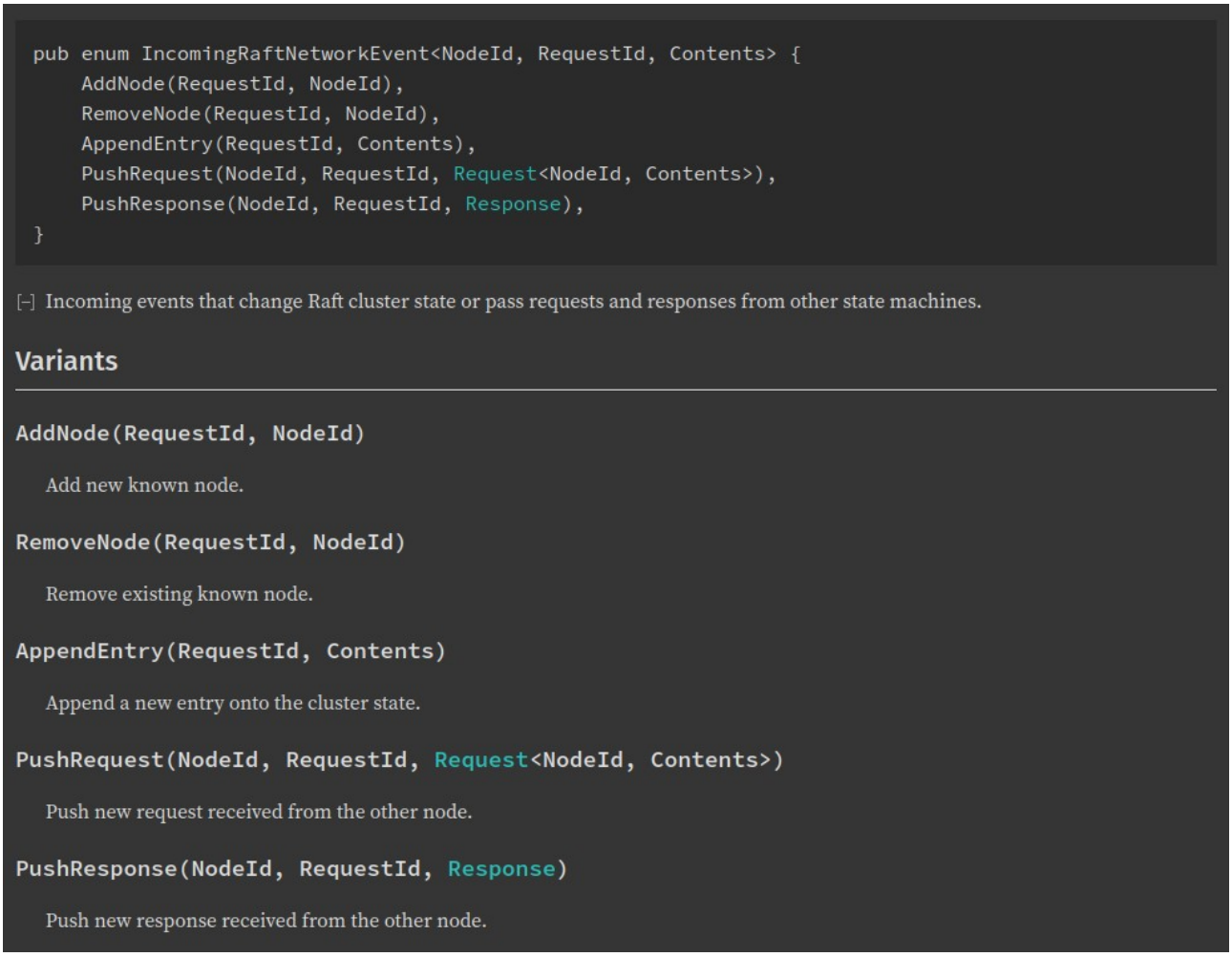
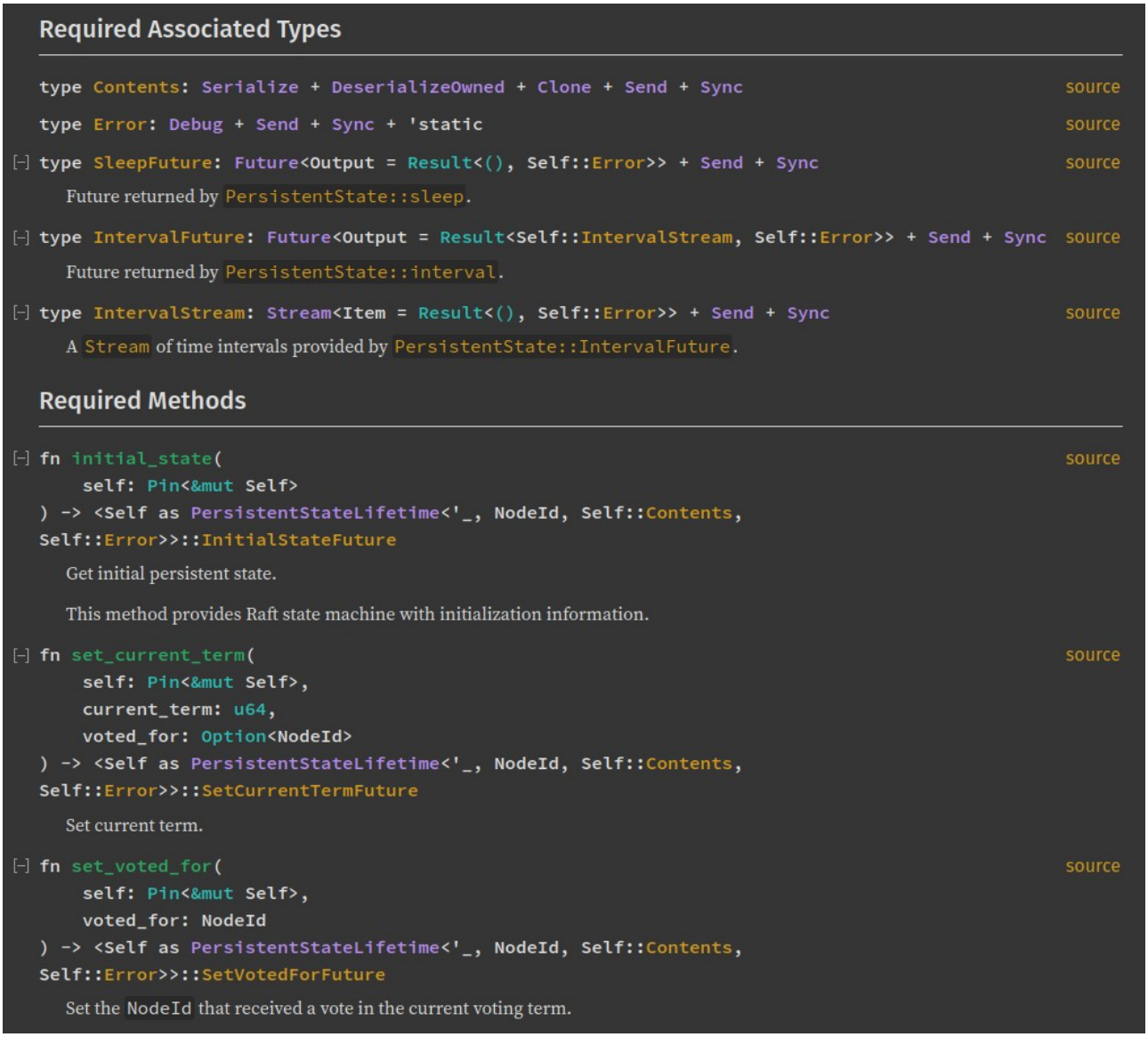


Рисунок 2.2 - Перелік вхідних подій

Це значно спрощує ініціалізацію бібліотеки зі сторони користувача, оскільки за допомогою системи типів мови програмування Rust повністю виражені усі вимоги щодо того, як саме скінченний автомат отримує зовнішні події та продукує власні.

Типаж `PersistentState` описує взаємодію скінченного автомату Raft з зовнішнім середовищем за допомогою асинхронних методів, які необхідно імплементувати користувачу (рис. 2.3). Під час імплементації типажу користувачу необхідно надати для скінченного автомату сховище даних, генератор псевдовипадкових чисел та таймери.



Required Associated Types

```
type Contents: Serialize + DeserializeOwned + Clone + Send + Sync                    source
type Error: Debug + Send + Sync + 'static                                            source
type SleepFuture: Future<Output = Result<(), Self::Error>> + Send + Sync             source
```

Future returned by `PersistentState::sleep`.

```
type IntervalFuture: Future<Output = Result<Self::IntervalStream, Self::Error>> + Send + Sync  source
```

Future returned by `PersistentState::interval`.

```
type IntervalStream: Stream<Item = Result<(), Self::Error>> + Send + Sync            source
```

A `Stream` of time intervals provided by `PersistentState::IntervalFuture`.

Required Methods

```
fn initial_state(                                                                    source
    self: Pin<&mut Self>
) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::InitialStateFuture
```

Get initial persistent state.

This method provides Raft state machine with initialization information.

```
fn set_current_term(                                                                 source
    self: Pin<&mut Self>,
    current_term: u64,
    voted_for: Option<NodeId>
) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::SetCurrentTermFuture
```

Set current term.

```
fn set_voted_for(                                                                    source
    self: Pin<&mut Self>,
    voted_for: NodeId
) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::SetVotedForFuture
```

Set the `NodeId` that received a vote in the current voting term.

Рисунок 2.3 - Опис типажу PersistentState

В цілому, для користувача бібліотека має інтерфейс у вигляді комбінатора для `futures::stream::Stream`. Подібні комбінатори зазвичай імплементовані як частина `futures::stream::StreamExt` та інших “*Ext” типажів. Для реалізації подібного функціоналу існують бібліотеки на кшталт `async_stream`[40] або `futures`[41], однак вони не підтримують роботу без стандартної бібліотеки Rust або мають занадто складний інтерфейс для кінцевого користувача.

Для цього проекту було розроблено окрему бібліотеку `enstream`[42], яка дозволяє перетворювати асинхронні блоки коду на асинхронні потоки, при цьому підтримуючи проекти без стандартної бібліотеки:

```
#![feature(type_alias_impl_trait)]

use std::future::Future;

use enstream::{HandlerFn, HandlerFnLifetime, Yielder, enstream};
use futures_util::{future::FutureExt, pin_mut, stream::StreamExt};

struct StreamState<'a> {
    val: &'a str
}

type Fut<'yielder, 'a: 'yielder> = impl Future<Output = ()>;

impl<'yielder, 'a> HandlerFnLifetime<'yielder, &'a str> for
StreamState<'a> {
    type Fut = Fut<'yielder, 'a>;
}

impl<'a> HandlerFn<&'a str> for StreamState<'a> {
    fn call<'yielder>(
        self,
        mut yielder: Yielder<'yielder, &'a str>,
    ) -> <Self as HandlerFnLifetime<'yielder, &'a str>>::Fut {
        async move {
            yielder.yield_item(self.val).await;
        }
    }
}
```

```
let owned = String::from("test");
let stream = enstream(StreamState {
    val: &owned
});
pin_mut!(stream);

assert_eq!(stream.next().now_or_never().flatten(), Some("test"));
```

Після виклику `into_raft_events` користувач отримує абстракний асинхронний потік вихідних подій (рис. 2.4), які необхідно переслати до інших учасників кластеру.

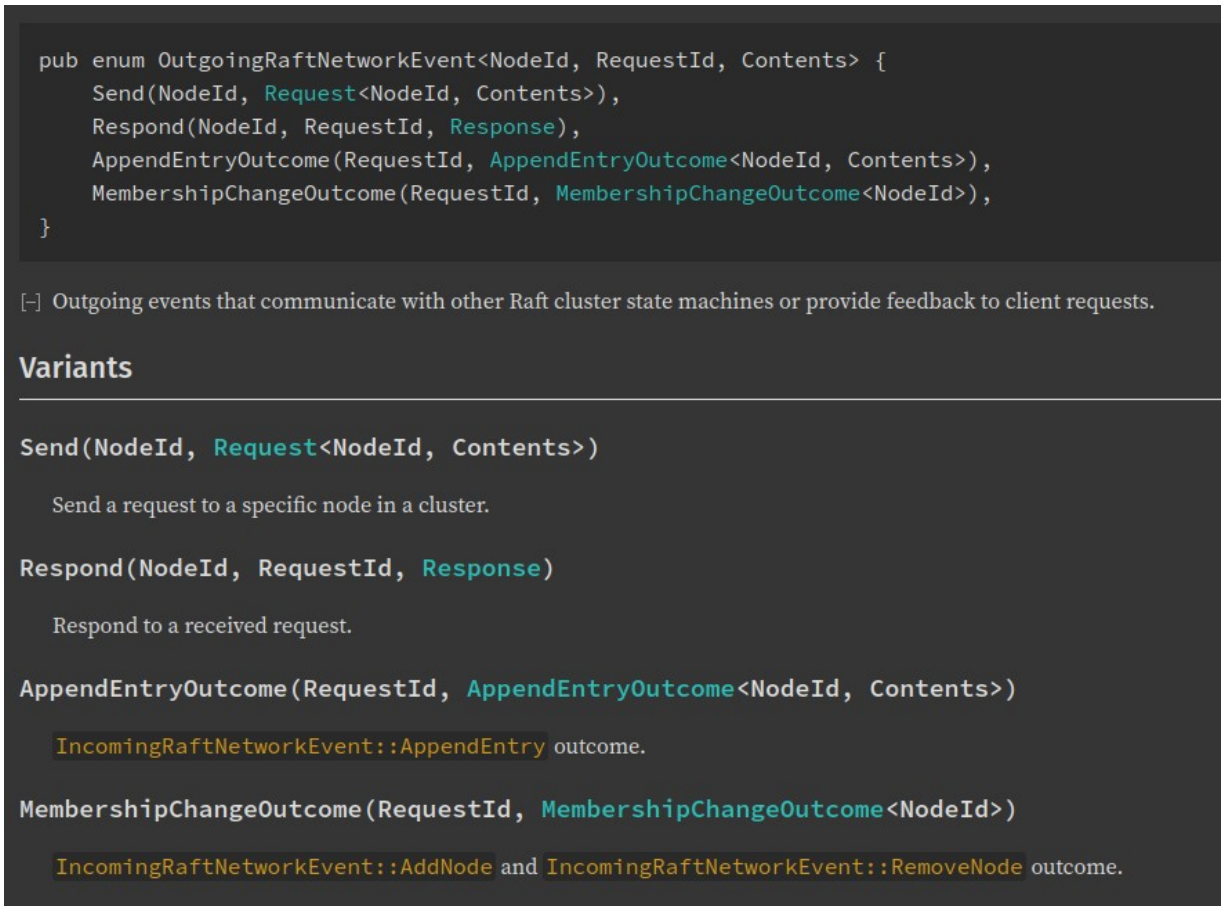


Рисунок 2.4 - Перелік вихідних подій

Користувач сам обирає яким чином надіслати події до інших учасників кластеру. Наприклад, для тестування можуть підходити асинхронні канали, які надаються разом з бібліотекою `tokio`, в той час як для відправки через мережу користувач може використати стек `libp2p`.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ БІБЛІОТЕКИ

3.1 Використання бібліотеки

Встановлення бібліотеки відбувається через створення нової залежності в файлі Cargo.toml, який декларує інформацію про проект для пакетного менеджера мови програмування Rust - Cargo:

```
aquaraft_raft = { path = "PATH_TO_LIB" }
```

За необхідності, користувач може ввімкнути функціонал діагностування роботи кластеру:

```
aquaraft_raft = { path = "PATH_TO_LIB", features = ["diagnostics"] }
```

Після цього варто додати до списку залежностей проекту асинхронне середовище виконання коду (наприклад, Tokio), пакети для роботи з каналами та потоками.

Середовище в якому буде змінюватися стан скінченного автомату Raft обирає користувач, створення скінченного автомату відбувається викликом `RaftStreamExt::into_raft_events` на типі асинхронного потоку. Під час створення скінченного автомату необхідно надати інформацію про ідентифікатор поточного серверу та сховище даних:

```
Box::pin(receiver.into_stream().into_raft_events(
    node_id.clone(),
    ShimmedState::from_node_configuration(vec![node_id]),
))
```

Одразу після цього користувач може використовувати вихідний потік який йому було надано викликом `RaftStreamExt::into_raft_events`. Для прикладу створимо два методи абстрактної структури яка зберігає скінченний автомат - одну для передачі нових подій до скінченного автомату, другу для отримання нових подій зі скінченного автомату:

```
fn send(&self, event: IncomingRaftNetworkEvent<NodeId, RequestId,
Contents>) {
    self.sender.send(event).unwrap();
```

```
}

async fn recv(&mut self) -> Option<OutgoingRaftNetworkEvent<NodeId,
RequestId, Contents>> {
    self.receiver.next().await.transpose().unwrap()
}
```

Для цього прикладу було використано бібліотеку `flume`, яка дозволяє створювати асинхронні канали. Обидва методи можуть використовуватися усередині конструкції на кшталт `tokio::select`[43]:

```
tokio::select! {
    Ok(incoming_request) = self.request_receiver.next() {
        // відправка події використовуючи метод send
    },
    Some(outgoing_event) = self.recv() {
        // обробка події, яка була отримана від методу recv
    }
}
```

Обмін подіями між учасниками кластеру користувач може імплементувати будь-яким зручним для себе методом, врахувавши при цьому особливості додатка в якому використовується бібліотека.

Якщо користувач ввімкнув підтримку діагностування кластеру, то використовуючи запит `IncomingRaftNetworkEvent::AskForDiagnostics` можна отримати інформацію про учасника кластеру, до якого був надісланий запит. Відповідь на цей запит включає в себе:

- поточну роль у кластері
- ідентифікатори відомих учасників кластеру
- поточний термін голосування
- ідентифікатор учасника за якого було віддано голос під час останнього голосування
- метадані останнього запису в сховищі даних

- кількість записів у сховищі даних
- ідентифікатор останнього підтвердженого запису

3.2 Автоматизація тестування бібліотеки

Бібліотека тестувалася за допомогою автоматичних та ручних тестів.

Для автоматичного тестування використовується спеціальний DSL, створений для простої ініціалізації кластеру Raft, обміну даними між нодами та контролю плину часу для тестування роботи таймерів. В якості асинхронного середовища під час автоматичного тестування використовується Tokio.

Приклад тесту написанного з використанням DSL написаного за допомогою інструментів метапрограмування мови Rust:

```
#[tokio::test(start_paused = true)]
async fn single_node_leader() {
    seq_test! {
        // Створюємо конфігурацію кластеру, яка складається з однієї
ноди
        node_configuration main_config, ["first"];

        // Ініціалізуємо одну ноду з використанням конфігурації вище
        node first, first_sender, main_config;

        // Намагаємося отримати подію від скінченного автомату
        recv (None) from first;

        skip 10005;

        // Надаємо ноді роль кандидата
        convert first to candidate;

        // Оскільки це єдина нода у кластері, перетворюємо її на лідера
без голосування
        convert first to leader;
```

```
    };
}
```

Для ретельного тестування ситуацій які можуть виникнути під час використання бібліотеки було описано різноманітні тест-кейси використовуючи вищезазначений DSL (рис. 3.1).

```
[ivan770@wolfram:~/Documents/Rust/aquaraft]$ cargo test --all-features
   Compiling aquaraft_raft v0.1.0 (/home/ivan770/Documents/Rust/aquaraft/crates/raft)
    Finished test [unoptimized + debuginfo] target(s) in 2.09s
     Running unittests src/lib.rs (target/debug/deps/aquaraft_raft-aff7fcfba42ccce2)

running 28 tests
test node::candidate::tests::start_elections ... ok
test node::candidate::tests::reject_append_entry ... ok
test node::candidate::tests::step_down_on_append_entries ... ok
test node::candidate::tests::single_node_leader ... ok
test node::candidate::tests::win_elections ... ok
test node::follower::tests::candidate_conversion ... ok
test node::follower::tests::accept_membership_changes ... ok
test node::follower::tests::accept_new_entries ... ok
test node::follower::tests::grant_vote ... ok
test node::follower::tests::max_possible_commit_index ... ok
test node::follower::tests::outdated_log_vote ... ok
test node::follower::tests::entry_replace_shutdown ... ok
test node::follower::tests::redirect_append_entry ... ok
test node::follower::tests::reject_out_of_cluster_request_vote ... ok
test node::follower::tests::reject_append_entry ... ok
test node::leader::tests::add_node ... ok
test node::follower::tests::shutdown_on_membership_change ... ok
test node::leader::tests::commit_index ... ok
test node::leader::tests::heartbeats_with_payloads ... ok
test node::leader::tests::empty_heartbeats ... ok
test node::leader::tests::leader_outage_in_membership_change ... ok
test node::leader::tests::reject_out_of_cluster_request_vote ... ok
test node::leader::tests::min_log_entry_id ... ok
test node::leader::tests::remove_node ... ok
test node::leader::tests::shutdown_on_remove_node ... ok
test node::leader::tests::reject_membership_change_when_unapplied ... ok
test node::leader::tests::single_node_commit_index ... ok
test node::tests::stream_termination ... ok
```

Рисунок 3.1 - Демонстрація покриття автоматичними тестами

Ручне тестування відбувається за допомогою окремого інструменту, який локально створює кластер Raft та контролює обмін подіями між учасниками кластеру з використанням команд (рис. 3.2).

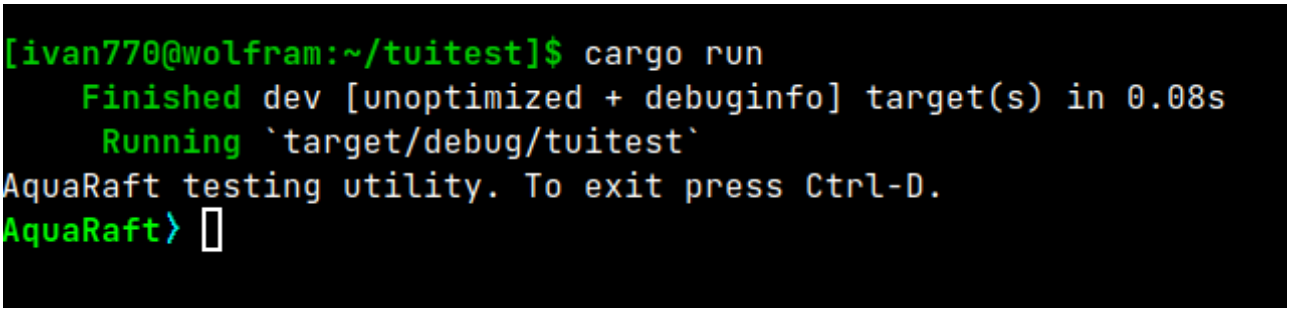


Рисунок 3.2 - Поле вводу команд

Для перегляду доступних операцій над кластером доступна окрема команда `help` (рис. 3.3).

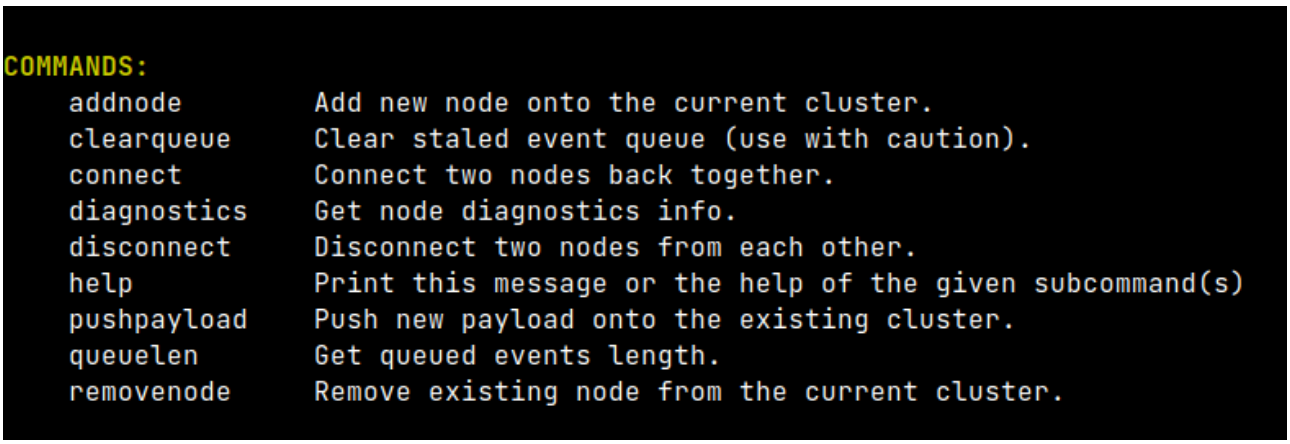


Рисунок 3.3 - Доступні операції

Наприклад, для тестування кластеру з трьох нод можна використати команду `addnode`. Для внесення нових записів в цей кластер можна викликати команду `pushpayload` (рис. 3.4).

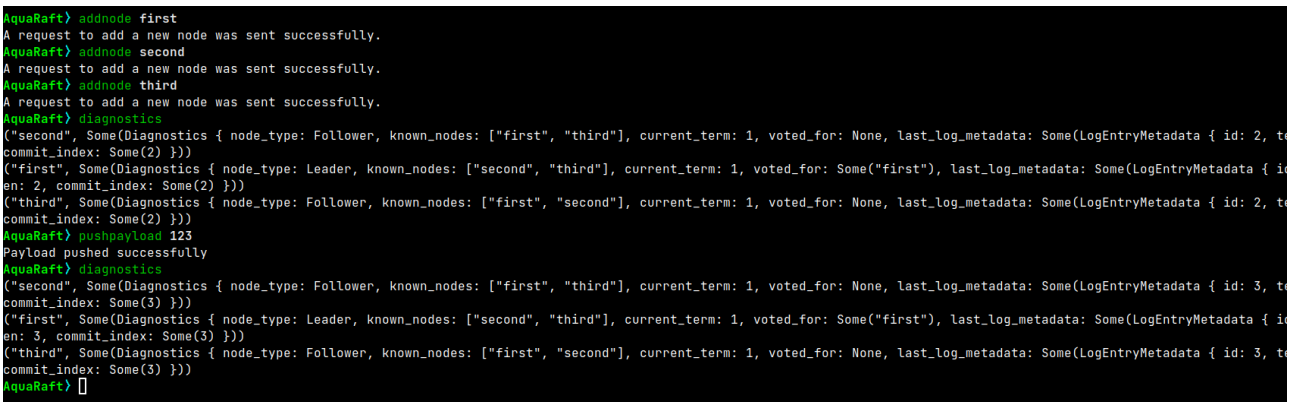


Рисунок 3.4 - Створення кластеру з трьох учасників

Тестування ситуацій, коли між учасниками кластеру відбувається фрагментація мережі відбувається за допомогою команд `connect` та `disconnect`. За замовчуванням, усі учасники кластеру можуть обмінюватися даними між собою без будь-яких перешкод.

Для прикладу, в кластері з трьох учасників від’єднано першу ноду, через що дві інші ноди роблять повторне голосування за лідера між собою (рис. 3.5).

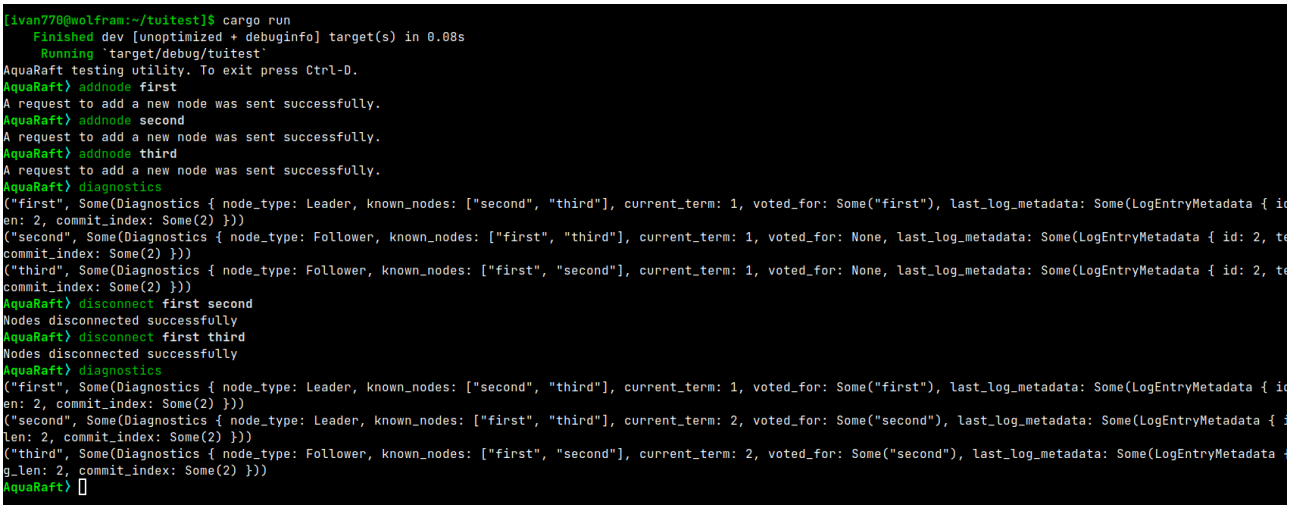


Рисунок 3.5 - Від’єднання першого учасника кластеру від мережі

Функціонал цієї утиліти дозволяє інтерактивно тестувати різні ситуації кластера Raft, за необхідності повторюючи їх потім використовуючи DSL для автоматичних тестів.

ВИСНОВКИ

В рамках цього проекту було проведено дослідження протоколу розподіленого консенсусу Raft та його реалізацій на різних мовах програмування.

Під час дослідження було проведено детальний аналіз специфікації як самого протоколу Raft, так і інших протоколів метою яких є вирішення проблеми розподіленого консенсусу, описано використання різних протоколів розподіленого консенсусу в існуючих проектах.

У ході виконання робіт розглянуті існуючі реалізації алгоритмів розподіленого консенсусу, а саме Paxos та Raft. Результатом розгляду став вибір мови програмування Rust та алгоритму Raft для досягнення мети роботи. Проаналізовано існуючі імплементації, а саме HashiCorp Raft, raft-rs та async-raft. Результат розгляду узагальнено в таблиці 1.1.

На основі зібраних вимог було спроектовано інтерфейс бібліотеки, завдяки якому користувач може асинхронно взаємодіяти зі скінченним автоматом Raft.

Для тестування бібліотеки було створено окремий DSL, який дозволяє впливати на весь процес обміну даними між учасниками кластеру.

СПИСОК ВИКОРИСТАННИХ ДЖЕРЕЛ

1. Google, Oracle Data Centers Knocked Offline by London Heat [Електронний ресурс] // Bloomberg. Режим доступу до ресурсу: https://web.archive.org/web/20230406205334/https://www.bloomberg.com/news/articles/2022-07-19/google-oracle-data-centers-knocked-offline-by-london-heat.

2. Millions of websites offline after fire at French cloud services firm [Електронний ресурс] // Reuters. Режим доступу до ресурсу: https://web.archive.org/web/20230508103925/https://www.reuters.com/article/idUSKBN2B20NU.

3. Google Cloud Service Health [Електронний ресурс] // Google Cloud. Режим доступу до ресурсу: https://web.archive.org/web/20230604185708/https://status.cloud.google.com/incidents/dS9ps52MUnxQfyDGPfkY.

4. Lamport L. The part-time parliament // Concurrency: the Works of Leslie Lamport. 2019. С. 277–317.

5. Ongaro D., Ousterhout J. In search of an understandable consensus algorithm // 2014 USENIX Annual Technical Conference (Usenix ATC 14). 2014. С. 305–319.

6. Ongaro D. Consensus: Bridging theory and practice. Stanford University, 2014.

7. Rust Programming Language [Електронний ресурс]. Режим доступу до ресурсу: https://web.archive.org/web/20230508082623/https://www.rust-lang.org/.

8. Amazon DynamoDB - Amazon Web Services [Електронний ресурс] // aws.amazon.com. Режим доступу до ресурсу: https://web.archive.org/web/20230504120433/https://aws.amazon.com/dynamodb/.

9. AWS re:Invent 2018: Amazon DynamoDB Under the Hood: How We Built a Hyper-Scale Database (DAT321) [Електронний ресурс] // youtube.com. Режим доступу до ресурсу: https://www.youtube.com/watch?v=yvBR71D0nAQ.

10. Elhemali M. et al. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service // 2022 USENIX Annual Technical Conference (USENIX ATC 22). Carlsbad, CA: USENIX Association, 2022. С. 1037–1048.

11. Regions and Zones [Електронний ресурс] // docs.aws.amazon.com. Режим доступу до ресурсу: https://web.archive.org/web/20230501204106/https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

12. Cockroach Labs, the company building CockroachDB [Електронний ресурс] // cockroachlabs.com. Режим доступу до ресурсу: https://web.archive.org/web/20230414011550/https://www.cockroachlabs.com/.

13. The new stack: Meet CockroachDB, the resilient SQL database [Електронний ресурс] // cockroachlabs.com. Режим доступу до ресурсу: https://web.archive.org/web/20230531092016/https://www.cockroachlabs.com/blog/the-new-stack-meet-cockroachdb-the-resilient-sql-database/.

14. Architecture Overview [Електронний ресурс] // cockroachlabs.com. Режим доступу до ресурсу: https://web.archive.org/web/20230506070208/https://www.cockroachlabs.com/docs/stable/architecture/overview.html#database-terms.

15. Architecture Overview [Електронний ресурс] // cockroachlabs.com. Режим доступу до ресурсу: https://web.archive.org/web/20230506070208/https://www.cockroachlabs.com/docs/stable/architecture/overview.html#layers.

16. Replication Layer [Електронний ресурс] // cockroachlabs.com. Режим доступу до ресурсу: https://web.archive.org/web/20230508102251/https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.

17. Scalable HTAP Database | TiDB [Електронний ресурс] // en.pingcap.com. Режим доступу до ресурсу: https://web.archive.org/web/20230505071427/https://www.pingcap.com/tidb/.

18. TiDB Storage [Електронний ресурс] // docs.pingcap.com. Режим доступу до ресурсу: https://web.archive.org/web/20230531091456/https://docs.pingcap.com/tidb/stable/tidb-storage.

19. TiSpark User Guide [Електронний ресурс] // docs.pingcap.com. Режим доступу до ресурсу: https://web.archive.org/web/20230531091135/https://docs.pingcap.com/tidb/dev/tispark-overview.

20. etcd [Електронний ресурс] // etcd.io. Режим доступу до ресурсу: https://web.archive.org/web/20230418091619/https://etcd.io/.

21. etcd-io/raft [Електронний ресурс] // github.com. Режим доступу до ресурсу: https://web.archive.org/web/20230414020704/https://github.com/etcd-io/raft.

22. Metrics | etcd [Електронний ресурс] // etcd.io. Режим доступу до ресурсу: https://web.archive.org/web/20230128232656/https://etcd.io/docs/v3.5/metrics/#server.

23. ScyllaDB [Електронний ресурс] // ScyllaDB. Режим доступу до ресурсу: https://web.archive.org/web/20230506055136/https://www.scylladb.com/.

24. Raft Consensus Algorithm in ScyllaDB [Електронний ресурс] // ScyllaDB. Режим доступу до ресурсу: https://web.archive.org/web/20230508102201/https://docs.scylladb.com/stable/architecture/raft.html.

25. Upgrade Considerations for ScyllaDB 5.0 and Later [Електронний ресурс] // ScyllaDB. Режим доступу до ресурсу: https://web.archive.org/web/20230508102639/https://docs.scylladb.com/branch-5.1/architecture/raft.html#upgrade-considerations-for-sylladb-5-0-and-later.

26. Lamport L. Fast paxos // Distributed Computing. Springer, 2006. Вип. 19, № 2. С. 79–103.

27. Chandra T.D., Griesemer R., Redstone J. Paxos made live: an engineering perspective // Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. 2007. С. 398–407.

28. Network partition - Wikipedia [Електронний ресурс] // en.wikipedia.org. Режим доступу до ресурсу: https://web.archive.org/web/20230327203148/https://en.wikipedia.org/wiki/Network_partition.

29. libp2p [Електронний ресурс] // libp2p.io. Режим доступу до ресурсу: https://web.archive.org/web/20230507005950/https://libp2p.io/.

30. raft-rs [Електронний ресурс] // Github. Режим доступу до ресурсу: https://web.archive.org/web/20220531152541/https://github.com/tikv/raft-rs.

31. raft::Raft::tick [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230508100614/https://docs.rs/raft/latest/raft/prelude/struct.Raft.html#method.tick.

32. raft::storage::Storage [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230508100916/https://docs.rs/raft/latest/raft/storage/trait.Storage.html.

33. tokio - Rust [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230425161902/https://docs.rs/tokio/latest/tokio/index.html#cpu-bound-tasks-and-blocking-code.

34. async-raft [Електронний ресурс] // Github. Режим доступу до ресурсу: https://web.archive.org/web/20230131181445/https://github.com/async-raft/async-raft.

35. Tokio - an asynchronous Rust runtime [Електронний ресурс] // tokio.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230507213628/https://tokio.rs/.

36. async-std [Електронний ресурс] // async.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230507213932/https://async.rs/.

37. wg-async [Електронний ресурс] // rust-lang.github.io. Режим доступу до ресурсу: https://web.archive.org/web/20230215093312/https://rust-lang.github.io/wg-async/vision/submitted_stories/status_quo/barbara_writes_a_runtime_agnostic_lib.html.

38. Modern embedded framework, using Rust and async. [Електронний ресурс] // github.com. Режим доступу до ресурсу: https://web.archive.org/web/20230324154311/https://github.com/embassy-rs/embassy.

39. futures::stream::Stream [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230406143402/https://docs.rs/futures/latest/futures/stream/trait.Stream.html.

40. tokio-rs/async-stream [Електронний ресурс] // github.com. Режим доступу до ресурсу: https://web.archive.org/web/20230508103006/https://github.com/tokio-rs/async-stream.

41. unfold in futures::stream [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230317013633/https://docs.rs/futures/latest/futures/stream/fn.unfold.html.

42. ivan770/enstream [Електронний ресурс] // github.com. Режим доступу до ресурсу: https://github.com/ivan770/enstream.

43. select in tokio - Rust [Електронний ресурс] // docs.rs. Режим доступу до ресурсу: https://web.archive.org/web/20230508102801/https://docs.rs/tokio/latest/tokio/macro.select.html.

ДОДАТКИ

Додаток А. Лістинг програмного коду

main.rs

```
#![feature(type_alias_impl_trait, drain_filter)]
#![allow(clippy::type_complexity)]

mod mini_swarm;
mod state;

use std::{
    collections::HashMap,
    str::FromStr,
    sync::{Arc, Mutex},
};

use aquaraft_raft::Diagnostics;
use flume::Sender;
use itertools::Itertools;
use mini_swarm::{MiniSwarm, MiniSwarmRequest};
use reedline_repl_rs::{
    clap::{value_parser, Arg, ArgMatches, Command},
    Repl,
};
use tracing_subscriber::{layer::SubscriberExt, util::SubscriberInitExt,
EnvFilter};

#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let logfile = tracing_appender::rolling::hourly("./", "aquaraft-
logs");

    tracing_subscriber::registry()
```

```
        .with(EnvFilter::from_str("aquaraft_raft=trace").unwrap())
        .with(
            tracing_subscriber::fmt::layer()
                .with_ansi(false)
                .with_writer(logfile),
        )
        .init();

    let (mini_swarm_sender, diagnostics_info) = MiniSwarm::spawn();

    tokio::task::spawn_blocking(move || {
        Repl::new((mini_swarm_sender, diagnostics_info))
            .with_name("AquaRaft")
            .with_banner("AquaRaft testing utility. To exit press Ctrl-
D.")
            .with_command(
                Command::new("addnode")
                    .arg(
                        Arg::new("name")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .about("Add new node onto the current cluster."),
                addnode,
            )
            .with_command(
                Command::new("removenode")
                    .arg(
                        Arg::new("name")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .about("Remove existing node from the current
cluster."),
                removenode,
```

```
            )
            .with_command(
                Command::new("diagnostics")
                    .arg(Arg::new("node").value_parser(value_parser!
(String)))
                    .about("Get node diagnostics info."),
                diagnostics,
            )
            .with_command(
                Command::new("pushpayload")
                    .arg(
                        Arg::new("payload")
                            .required(true)
                            .value_parser(value_parser!(usize)),
                    )
                    .about("Push new payload onto the existing
cluster."),
                pushpayload,
            )
            .with_command(
                Command::new("connect")
                    .arg(
                        Arg::new("one")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .arg(
                        Arg::new("two")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .about("Connect two nodes back together."),
                connect,
            )
            .with_command(
```

```
                Command::new("disconnect")
                    .arg(
                        Arg::new("one")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .arg(
                        Arg::new("two")
                            .required(true)
                            .value_parser(value_parser!(String)),
                    )
                    .about("Disconnect two nodes from each other."),
                disconnect,
            )
            .with_command(
                Command::new("queuelen").about("Get queued events
length."),
                queuelen,
            )
            .with_command(
                Command::new("clearqueue").about("Clear staled event
queue (use with caution)."),
                clearqueue,
            )
            .run()
    })
    .await??;

    Ok(())
}

fn addnode(
    args: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
```

```
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    sender
        .send(MiniSwarmRequest::AddNode(Arc::from(
            &**args.get_one::<String>("name").unwrap(),
        )))
        .unwrap();

    Ok(Some(String::from(
        "A request to add a new node was sent successfully.",
    )))
}

fn removenode(
    args: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    sender
        .send(MiniSwarmRequest::RemoveNode(Arc::from(
            &**args.get_one::<String>("name").unwrap(),
        )))
        .unwrap();

    Ok(Some(String::from(
        "A request to remove a node was sent successfully.",
    )))
}
```

```
fn diagnostics(
    args: ArgMatches,
    (_, diagnostics_info): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    let diagnostics = &diagnostics_info.lock().unwrap();

    let requested_value = if let Some(node_id) =
args.get_one::<String>("node") {
        format!("{:?}", diagnostics.1.get(&**node_id))
    } else {
        format!(
            "{}",
            diagnostics
                .1
                .iter()
                .format_with("\n", |val, f| f(&format_args!("{:?}",
val)))
        )
    };

    Ok(Some(requested_value))
}

fn pushpayload(
    args: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
```

```
) -> Result<Option<String>, reedline_repl_rs::Error> {
    let payload = args.get_one::<usize>("payload").unwrap();

    sender
        .send(MiniSwarmRequest::PushPayload(*payload))
        .unwrap();

    Ok(Some(String::from("Payload pushed successfully")))
}

fn connect(
    args: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    let one = args.get_one::<String>("one").unwrap();
    let two = args.get_one::<String>("two").unwrap();

    if one != two {
        sender
            .send(MiniSwarmRequest::Activate(
                Arc::from(&**one),
                Arc::from(&**two),
            ))
            .unwrap();
    }

    Ok(Some(String::from("Nodes connected successfully")))
}

fn disconnect(
```

```
    args: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    let one = args.get_one::<String>("one").unwrap();
    let two = args.get_one::<String>("two").unwrap();

    if one != two {
        sender
            .send(MiniSwarmRequest::Deactivate(
                Arc::from(&**one),
                Arc::from(&**two),
            ))
            .unwrap();
    }

    Ok(Some(String::from("Nodes disconnected successfully")))
}

fn queuelen(
    _: ArgMatches,
    (_, diagnostics_info): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    let diagnostics = &diagnostics_info.lock().unwrap();

    Ok(Some(format!("Event queue length: {}", diagnostics.0)))
}
```

```
fn clearqueue(
    _: ArgMatches,
    (sender, _): &mut (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ),
) -> Result<Option<String>, reedline_repl_rs::Error> {
    sender.send(MiniSwarmRequest::ClearQueuedEvents).unwrap();

    Ok(Some(String::from("Event queue cleared successfully")))
}
```

mini_swarm.rs

```
use std::{
    collections::{HashMap, HashSet},
    convert::Infallible,
    future::Future,
    pin::Pin,
    sync::{Arc, Mutex},
    task::Poll,
    time::Duration,
};

use aquaraft_raft::{
    Diagnostics, DiagnosticsNodeType, IncomingRaftNetworkEvent,
OutgoingRaftNetworkEvent,
    RaftStreamExt,
};
use flume::{Receiver, Sender};
use futures_util::{future::poll_fn, pin_mut, stream::FusedStream,
StreamExt};
use rand::{prelude::IteratorRandom, thread_rng};
use tokio::time::interval;
```

```
use crate::state::ShimmedState;

pub enum MiniSwarmRequest {
    AddNode(Arc<str>),
    RemoveNode(Arc<str>),
    Activate(Arc<str>, Arc<str>),
    Deactivate(Arc<str>, Arc<str>),
    ClearQueuedEvents,
    PushPayload(usize),
}

type RaftStream = impl FusedStream<Item =
Result<OutgoingRaftNetworkEvent<Arc<str>, (), usize>, Infallible>>
    + Send
    + Sync;

struct MiniSwarmMember {
    sender: Sender<IncomingRaftNetworkEvent<Arc<str>, (), usize>>,
    receiver: Pin<Box<RaftStream>>,
}

impl MiniSwarmMember {
    fn new(node_id: Arc<str>) -> Self {
        let (sender, receiver) = flume::unbounded();

        MiniSwarmMember {
            sender,
            receiver: Box::pin(receiver.into_stream().into_raft_events(
                node_id.clone(),
                ShimmedState::from_node_configuration(vec![node_id]),
            )),
        }
    }
```

```
    fn send(&self, event: IncomingRaftNetworkEvent<Arc<str>, (), usize>)
{
        self.sender.send(event).unwrap();
    }

    async fn recv(&mut self) ->
Option<OutgoingRaftNetworkEvent<Arc<str>, (), usize>> {
        self.receiver.next().await.transpose().unwrap()
    }
}

pub struct MiniSwarm {
    active_connections: HashSet<(Arc<str>, Arc<str>)>,
    diagnostics: Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    members: HashMap<Arc<str>, MiniSwarmMember>,
    queued_events: Vec<(
        Arc<str>,
        Arc<str>,
        IncomingRaftNetworkEvent<Arc<str>, (), usize>,
    )>,
    request_receiver: Receiver<MiniSwarmRequest>,
}

impl MiniSwarm {
    pub fn spawn() -> (
        Sender<MiniSwarmRequest>,
        Arc<Mutex<(usize, HashMap<Arc<str>,
Option<Diagnostics<Arc<str>>>>)>>,
    ) {
        let (request_sender, request_receiver) = flume::unbounded();

        let diagnostics = Arc::new(Mutex::new((0, HashMap::default())));
```

```
        tokio::spawn(
            MiniSwarm {
                active_connections: HashSet::default(),
                diagnostics: diagnostics.clone(),
                members: HashMap::default(),
                queued_events: Vec::new(),
                request_receiver,
            }
            .drive(),
        );

        (request_sender, diagnostics)
    }

    pub async fn drive(mut self) {
        let mut diagnostics_interval =
interval(Duration::from_millis(100));

        loop {
            let suitable_events =
                self.queued_events
                    .drain_filter(|(sender_idx, receiver_idx, _)| {
                        self.active_connections
                            .contains(&(sender_idx.clone(),
receiver_idx.clone()))
                    });

            for (_, receiver_idx, event) in suitable_events {
                if let Some(receiver) = self.members.get(&receiver_idx)
{
                    receiver.send(event);
                }
            }
```

```
            let selector = poll_fn(|cx| {
                let member = self
                    .members
                    .iter_mut()
                    .map(|(idx, member)| (idx.clone(), member))
                    .choose(&mut thread_rng());

                match member {
                    Some((idx, member)) => {
                        let fut = member.recv();
                        pin_mut!(fut);

                        fut.poll(cx).map(|val| (idx, val))
                    }
                    None => Poll::Pending,
                }
            });

            tokio::select! {
                Ok(incoming_request) =
self.request_receiver.recv_async() => {
                    match incoming_request {
                        MiniSwarmRequest::AddNode(node_id) => {
                            let mut locked_diagnostics =
self.diagnostics
                                .lock()
                                .unwrap();

                            if self.members.is_empty() {
                                self.members.insert(node_id.clone(),
MiniSwarmMember::new(node_id.clone()));

                                locked_diagnostics.1.insert(node_id,
```

```
None);
                            } else {
                                let approximate_leader =
locked_diagnostics.1
                                    .iter()
                                    .find_map(|(idx, diagnostics)| match
diagnostics {
                                        Some(diagnostics) if
diagnostics.node_type == DiagnosticsNodeType::Leader =>
self.members.get(idx),
                                        _ => None
                                    });

                                if let Some(approximate_leader) =
approximate_leader {
                                    for member in self.members.keys() {

self.active_connections.insert((member.clone(), node_id.clone()));

self.active_connections.insert((node_id.clone(), member.clone()));
                                    }



approximate_leader.send(IncomingRaftNetworkEvent::AddNode((),
node_id.clone()));

                                    self.members.insert(node_id.clone(),
MiniSwarmMember::new(node_id.clone()));

                                    locked_diagnostics.1.insert(node_id,
None);
                                }
                            }
                        },
```

```
                        MiniSwarmRequest::RemoveNode(node_id) => {
                            let locked_diagnostics = self.diagnostics
                                .lock()
                                .unwrap();

                            let approximate_leader =
locked_diagnostics.1
                                .iter()
                                .find_map(|(idx, diagnostics)| match
diagnostics {
                                    Some(diagnostics) if
diagnostics.node_type == DiagnosticsNodeType::Leader =>
self.members.get(idx),
                                    _ => None
                                });

                            if let Some(approximate_leader) =
approximate_leader {

approximate_leader.send(IncomingRaftNetworkEvent::RemoveNode((),
node_id.clone()));
                            }
                        },
                        MiniSwarmRequest::Activate(one, two) => {
                            self.active_connections.insert((one.clone(),
two.clone()));
                            self.active_connections.insert((two, one));
                        },
                        MiniSwarmRequest::Deactivate(one, two) => {

self.active_connections.remove(&(one.clone(), two.clone()));
                            self.active_connections.remove(&(two, one));
                        },
                        MiniSwarmRequest::ClearQueuedEvents => {
```

```
                            self.queued_events.clear()
                        }
                        MiniSwarmRequest::PushPayload(payload) => {
                            let approximate_leader = self.diagnostics
                                .lock()
                                .unwrap()
                                .1
                                .iter()
                                .find_map(|(idx, diagnostics)| match
diagnostics {
                                    Some(diagnostics) if
diagnostics.node_type == DiagnosticsNodeType::Leader =>
self.members.get(idx),
                                    _ => None
                                });

                            if let Some(approximate_leader) =
approximate_leader {

approximate_leader.send(IncomingRaftNetworkEvent::AppendEntry((),
payload));
                            }
                        },
                    }
                },
                (sender_idx, network_event) = selector => {
                    match network_event {

Some(OutgoingRaftNetworkEvent::Send(receiver_idx, request)) => {
                            if let Some(receiver) =
self.members.get(&receiver_idx) &&
self.active_connections.contains(&(sender_idx.clone(),
receiver_idx.clone())) {
```

```
receiver.send(IncomingRaftNetworkEvent::PushRequest(sender_idx.clone(),
(), request));
                            } else {

self.queued_events.push((sender_idx.clone(), receiver_idx.clone(),
IncomingRaftNetworkEvent::PushRequest(sender_idx.clone(), (),
request)));
                            }
                        },

Some(OutgoingRaftNetworkEvent::Respond(receiver_idx, request_id,
response)) => {
                            if let Some(receiver) =
self.members.get(&receiver_idx) &&
self.active_connections.contains(&(sender_idx.clone(),
receiver_idx.clone())) {

receiver.send(IncomingRaftNetworkEvent::PushResponse(sender_idx.clone(),
request_id, response));
                            } else {

self.queued_events.push((sender_idx.clone(), receiver_idx.clone(),
IncomingRaftNetworkEvent::PushResponse(sender_idx.clone(), request_id,
response)));
                            }
                        }

Some(OutgoingRaftNetworkEvent::Diagnostics(diagnostics)) => {
                            let mut locked_diagnostics =
self.diagnostics.lock().unwrap();
                            locked_diagnostics.0 =
self.queued_events.len();

locked_diagnostics.1.insert(sender_idx.clone(), Some(diagnostics));
```

```
                        },

Some(OutgoingRaftNetworkEvent::AppendEntryOutcome(_, _)) => {},

Some(OutgoingRaftNetworkEvent::MembershipChangeOutcome(_, _)) => {},
                        None => {
                            let mut locked_diagnostics =
self.diagnostics
                                .lock()
                                .unwrap();

                            for member in self.members.keys() {

self.active_connections.remove(&(member.clone(), sender_idx.clone()));

self.active_connections.remove(&(sender_idx.clone(), member.clone()));
                            }

                            self.members.remove(&sender_idx);

                            locked_diagnostics.1.remove(&sender_idx);
                        }
                    }
                },
                _ = diagnostics_interval.tick() => {
                    for member in self.members.values() {

member.send(IncomingRaftNetworkEvent::AskForDiagnostics);
                    }
                }
            }
        }
```

```
    }
}
```

state.rs

```
use std::{convert::Infallible, future, num::NonZeroU64, pin::Pin,
time::Duration};

use aquaraft_raft::state::{InitialState, LogEntry, PersistentState,
PersistentStateLifetime};
use futures_util::{stream, Future, FutureExt, Stream};
use rand::{rngs::ThreadRng, thread_rng};
use serde::{de::DeserializeOwned, Serialize};
use tokio::time::{interval, sleep};

pub struct ShimmedState<NodeId, Contents> {
    current_term: u64,
    log: Vec<LogEntry<NodeId, Contents>>,
    applied_log: Vec<LogEntry<NodeId, Contents>>,
    node_configuration: Vec<NodeId>,
    voted_for: Option<NodeId>,
}

impl<NodeId, Contents> Default for ShimmedState<NodeId, Contents> {
    fn default() -> Self {
        Self {
            current_term: Default::default(),
            log: Default::default(),
            applied_log: Default::default(),
            node_configuration: Default::default(),
            voted_for: Default::default(),
        }
    }
}

impl<NodeId, Contents> ShimmedState<NodeId, Contents> {
```

```
    pub fn from_node_configuration(node_configuration: Vec<NodeId>) ->
Self {
        Self {
            node_configuration,
            ..Default::default()
        }
    }
}

type InitialStateFuture<'a, NodeId> =
    impl Future<Output = Result<InitialState<NodeId>, Infallible>>;
type SetCurrentTermFuture<'a> = impl Future<Output = Result<(),
Infallible>>;
type SetVotedForFuture<'a> = impl Future<Output = Result<(),
Infallible>>;
type LogEntriesFuture<'a, NodeId, Contents> =
    impl Future<Output = Result<Vec<LogEntry<NodeId, Contents>>,
Infallible>>;
type AppendLogEntryFuture<'a> = impl Future<Output = Result<(),
Infallible>>;
type RemoveLogEntriesFuture<'a> = impl Future<Output = Result<(),
Infallible>>;
type ApplyLogEntriesFuture<'a> = impl Future<Output = Result<(),
Infallible>>;
type SleepFuture = impl Future<Output = Result<(), Infallible>>;
type IntervalFuture<NodeId, Contents>
where
    NodeId: Clone + Send + Sync + Unpin,
    Contents: Serialize + DeserializeOwned + Clone + Send + Sync +
Unpin,
= impl Future<Output = Result<IntervalStream<NodeId, Contents>,
Infallible>>;
type IntervalStream<NodeId, Contents>
where
    NodeId: Clone + Send + Sync + Unpin,
```

```
    Contents: Serialize + DeserializeOwned + Clone + Send + Sync +
Unpin,
= impl Stream<Item = Result<(), Infallible>>;

impl<'a, NodeId, Contents> PersistentStateLifetime<'a, NodeId, Contents,
Infallible>
    for ShimmedState<NodeId, Contents>
where
    NodeId: Clone + Send + Sync + Unpin,
    Contents: Serialize + DeserializeOwned + Clone + Send + Sync +
Unpin,
{
    type Rng = ThreadRng;

    type InitialStateFuture = InitialStateFuture<'a, NodeId>;

    type SetCurrentTermFuture = SetCurrentTermFuture<'a>;

    type SetVotedForFuture = SetVotedForFuture<'a>;

    type LogEntriesFuture = LogEntriesFuture<'a, NodeId, Contents>;

    type AppendLogEntryFuture = AppendLogEntryFuture<'a>;

    type RemoveLogEntriesFuture = RemoveLogEntriesFuture<'a>;

    type ApplyLogEntriesFuture = ApplyLogEntriesFuture<'a>;
}

impl<NodeId, Contents> PersistentState<NodeId> for ShimmedState<NodeId,
Contents>
where
    NodeId: Clone + Send + Sync + Unpin,
```

```
    Contents: Serialize + DeserializeOwned + Clone + Send + Sync +
Unpin,
{
    type Contents = Contents;

    type Error = Infallible;

    type SleepFuture = SleepFuture;

    type IntervalFuture = IntervalFuture<NodeId, Self::Contents>;

    type IntervalStream = IntervalStream<NodeId, Self::Contents>;

    fn initial_state(
        self: Pin<&mut Self>
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::InitialStateFuture {
        future::ready(Ok(InitialState {
            current_term: self.current_term,
            last_committed_entry_id: self.applied_log.last().map(|entry|
entry.metadata.id),
            last_log_entry: self.log.last().map(|entry| entry.metadata),
            node_configuration: self.node_configuration.clone(),
            voted_for: self.voted_for.clone(),
        }))
    }

    fn set_current_term(
        mut self: Pin<&mut Self>,
        current_term: u64,
        voted_for: Option<NodeId>
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::SetCurrentTermFuture {
        self.current_term = current_term;
```

```
        self.voted_for = voted_for;
        future::ready(Ok(()))
    }

    fn set_voted_for(
        mut self: Pin<&mut Self>,
        voted_for: NodeId,
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::SetVotedForFuture {
        self.voted_for = Some(voted_for);
        future::ready(Ok(()))
    }

    fn log_entries(
        self: Pin<&mut Self>,
        start: NonZeroU64,
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::LogEntriesFuture {
        future::ready(Ok(self
            .log
            .get((start.get() - 1) as usize..)
            .unwrap_or_default()
            .to_vec()))
    }

    fn append_log_entry(
        mut self: Pin<&mut Self>,
        log_entry: LogEntry<NodeId, Self::Contents>,
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::AppendLogEntryFuture {
        self.log.push(log_entry);
        future::ready(Ok(()))
    }
```

```
    fn remove_log_entries(
        mut self: Pin<&mut Self>,
        start: NonZeroU64,
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::RemoveLogEntriesFuture {
        self.log.truncate((start.get() - 1) as usize);
        future::ready(Ok(()))
    }

    fn apply_log_entries(
        mut self: Pin<&mut Self>,
        end: NonZeroU64
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::ApplyLogEntriesFuture {
        let start = self.applied_log.len();
        let log = self.log[start..end.get() as usize].to_vec();
        self.applied_log.extend_from_slice(&log);
        future::ready(Ok(()))
    }

    fn sleep(self: Pin<&mut Self>, duration: Duration) ->
Self::SleepFuture {
        sleep(duration).map(Ok)
    }

    fn interval(self: Pin<&mut Self>, duration: Duration) ->
Self::IntervalFuture {
        future::ready(Ok(stream::unfold(
            interval(duration),
            |mut state| async move {
                state.tick().await;
                Some((Ok(()), state))
            },
        )))
```

```
    }

    fn rng(
        self: Pin<&mut Self>,
    ) -> <Self as PersistentStateLifetime<'_, NodeId, Self::Contents,
Self::Error>>::Rng {
        thread_rng()
    }
}
```