НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

«На правах рукопису»
УДК _____

До захисту допущено:

В. о. завідувача кафедри

_____ Олександр РОЛІК

«___»_____ 2021 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інформаційні управляючі системи та технології»
зі спеціальності 126 «Інформаційні системи та технології»
на тему: «Система підтримки прийняття консенсусу між серверами у розподілених системах»

Виконала:
студентка VI курсу, групи ІС-з01мп
Козлова Ольга Сергіївна _____

Керівник:
к.т.н., доцент
Сперкач Майя Олегівна _____

Рецензент:
Професор, д.т.н., доцент,
Клименко Ірина Анатоліївна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ

В. о. завідувача кафедри

_____ Олександр РОЛІК

«___»_____2021 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Козлова Ольга Сергіївна

1. Тема дисертації «Система підтримки прийняття консенсусу між серверами у розподілених системах», науковий керівник дисертації Сперкач Майя Олегівна, к.т.н, доцент, затверджені наказом по університету від «25» жовтня 2021 р. № 3575-с

2. Термін подання студентом дисертації _____

3. Об’єкт дослідження: *процес досягнення консенсусу між серверами у розподілених системах*

4. Вихідні дані: *пояснювальна записка, графічний матеріал*

5. Перелік завдань, які потрібно розробити: *дослідити предметне середовище; провести аналіз відомих результатів розв’язання та статей; сформулювати постановку задачі; розробити алгоритми та їх модифікації; розробити інформаційну систему; проаналізувати результати роботи алгоритмів*

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: *1. Блок-схема виборів алгоритму Raft 2. Блок-схема реплікації алгоритму Raft 3. Блок-схема обробки запитів лідера алгоритму Raft 4. Схема структурна варіантів використання 5. Діаграма класів серверної частини додатку 6. Діаграма класів клієнтської частини*

додатку 7. Діаграма компонентів системи 8. Діаграма послідовності для процесів виборів та реплікації алгоритму Raft

7. Орієнтовний перелік публікацій

8. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Аналіз результатів огляду літератури	10.10.2021	
2	Аналіз існуючих методів розв'язання задачі	16.10.2021	
3	Постановка та формалізація задачі	22.10.2021	
4	Постановка математичної моделі задачі	25.10.2021	
5	Розробка модифікацій існуючих методів розв'язання задачі	26.10.2021	
6	Розробка інформаційного та програмного забезпечення	01.11.2021	
7	Аналіз досліджень розроблених алгоритмів	15.11.2021	
8	Оформлення документації	22.11.2021	
9	Подання роботи на попередній захист	23.11.2021	
10	Подання роботи на основний захист	23.12.2021	

Студент

Науковий керівник

АНОТАЦІЯ

Магістерська дисертація: 116 с., 35 рис., 23 табл., 2 додатки, 46 джерел.

Актуальність. У даний час розподілені мікросервісні системи стали фактично промисловим стандартом. Виходячи із вимог до сучасних систем, таких як мультиплатформеність, стабільність та керованість, більшість розробників прагнуть перейти від монолітної структури до мікросервісної. Один із перших кроків, який можна зробити, це досягнути горизонтальної масштабованості системи. Тобто, перетворити монолітний застосунок в кластер, що може складатися з тих же самих монолітів, але дозволяє динамічно варіювати їх кількість, що гарантує стабільність роботи у випадку відмови будь-якого з вузлів і надає можливість безперервного функціонування системи.

При спробі досягнення горизонтальної масштабованості, постає питання синхронізації даних всередині кластера. Тоді і з'являється проблема консенсусу – як саме можна забезпечити консистентність даних в усіх вузлах? Для цього існують алгоритми прийняття консенсусу, що вирішують задачу отримання узгодженого значення групою учасників у ситуації, коли можливі відмови деяких вузлів, надання недостовірної інформації або спотворення переданих значень середовищем передачі даних.

Існують уже готові реалізації таких алгоритмів, що використовуються у базах даних чи фреймворках. Але використання готових рішень для систем великих розмірів – це обов'язково додавання додаткових точок відмови. Фреймворки можуть бути надлишкові або складні в експлуатації та вивченні, а можуть і зовсім не існувати для деякої мови програмування. До того ж, алгоритми прийняття консенсусу можуть мати різні реалізації, які не обов'язково є ефективними. Тому реалізація та покращення таких алгоритмів є доцільними.

Мета дослідження. Метою даного дослідження є покращення алгоритму прийняття консенсусу в розподілених системах.

Ціль дослідження. Розробити систему, що дозволяє адмініструвати кластер з кількох серверів, у якому працює покращений алгоритм прийняття консенсусу.

Результат. У якості результату даного дослідження очікується отримати покращений алгоритм прийняття консенсусу, що є менш енергоємним, аніж його аналоги, зберігаючи ті ж показники сприятливості відмов вузлів.

Для досягнення результату необхідно виконати наступні **завдання**:

– спостереження: виконати огляд алгоритмів Paxos і Raft для досягнення консенсусу в розподілених системах з метою порівняння показників їх ефективності;

– формулювання гіпотез: виконати порівняльний аналіз існуючих алгоритмів, їхньої складності та результатів роботи на системах із різною кількістю вузлів; покращити обраний алгоритм прийняття консенсусу в розподілених системах шляхом заміни статичного кворуму при виборі лідера динамічним;

– експериментування: обрати показники ефективності для аналізу, налаштувати тестову розподілену систему у віртуальному середовищі, що буде складатися з декількох вузлів, які мають доступ до деякого спільного сховища даних. Протестувати роботу покращеного алгоритму на різних параметрах системи;

– аналіз: проаналізувати показники ефективності розробленого алгоритму, порівняти його з відповідними показниками експериментів існуючих алгоритмів;

– формулювання висновків: на основі проведеного аналізу зробити висновки про те, чи є покращений алгоритм менш енергоємним і більш сприятливим до відмов, аніж його аналоги.

Новизна дослідження. Новизна полягає у використанні динамічного кворуму для вибору вузла–лідера у покращеному алгоритмі для досягнення консенсусу.

Публікації. Матеріали роботи опубліковані у XI міжнародній науково–практичній конференції «InfoCom Advanced Solution 2021» [1].

РОЗПОДІЛЕНІ СИСТЕМИ, АЛГОРИТМ, КОНСЕНСУС, КЛАСТЕР, ВУЗЛИ, РОЗПОДІЛЕНИЙ ЛОГ

ABSTRACT

Master's dissertation: 116 p., 35 figures, 23 tables, 2 applications, 46 sources.

Topicality. Currently, distributed microservice systems have become the industry standard. Based on the requirements of modern systems, such as multiplatform, stability and manageability, most developers seek to move from a monolithic structure to a microservice. One of the first steps that can be taken without refactoring and decomposition is to achieve horizontal scalability of the system. That is, to convert a monolithic application into a cluster that may consist of the same monoliths, but allows you to dynamically vary their number, which ensures the stability of any failure of any of the nodes and allows continuous operation of the system.

When trying to achieve horizontal scalability, the question of data synchronization within the cluster arises. Then there is the problem of consensus – how exactly can you ensure the consistency of data in all nodes? To do this, there are consensus acceptance algorithms that solve the problem of obtaining an agreed value by a group of participants in a situation where there may be failures of some nodes, providing inaccurate information or distortion of the transmitted values by the data transmission medium.

There are ready–made implementations of such algorithms used in databases or frameworks. But using ready–made solutions for large systems is sure to add extra points of failure. Frameworks may be redundant or difficult to operate and learn, or may not exist at all for some programming language. In addition, consensus algorithms can have different implementations that are not necessarily effective. Therefore, the implementation and improvement of such algorithms are appropriate.

Objective. The purpose of this study is to improve the algorithm for consensus acceptance in distributed systems.

Purpose. Develop a system that allows you to administer a cluster of multiple servers, which has an improved consensus acceptance algorithm.

Result. As a result of this study, it is expected to obtain an improved consensus acceptance algorithm, which has less energy footprint than its analogues, while maintaining the same rates of node failure.

To achieve the result it is necessary to perform the following **tasks**:

– observations: Review Paxos and Raft algorithms of consensus acceptance in distributed systems to compare their performance;

– formulation of hypotheses: perform a comparative analysis of existing algorithms, their complexity and the results of work on systems with different numbers of nodes; improve the chosen algorithm of consensus acceptance in distributed systems by replacing the static quorum when choosing a leader with a dynamic one;

– experimentation: choose performance metrics for analysis, configure a test distributed system in a virtual environment that will consist of several nodes that have access to some common data storage. Test the operation of the improved algorithm on various system parameters;

– analysis: analyze the performance metrics of the developed algorithm, compare it with the corresponding metrics of experiments of existing algorithms;

– formulation of conclusions: on the basis of the conducted analysis make conclusions on whether the improved algorithm is less energy–intensive and more favorable to failures than its analogues.

Scientific novelty. The novelty of this study is the use of a dynamic quorum to select a leader in an improved algorithm to reach consensus.

Publications. Materials of the study were published in the XI international scientific–practical conference "InfoCom Advanced Solution 2021" [1].

DISTRIBUTED SYSTEMS, ALGORITHM, CONSENSUS, CLUSTER, NODES, DISTRIBUTED LOG

ЗМІСТ

ВСТУП

Використання розподілених систем для вирішення складних задач є важливим етапом для розвитку сфери інформаційних технологій, оскільки кількість та складність таких задач зростає у прогресії, і єдиний комп’ютер не може впоратися з ними самостійно. Розподілені системи мають ряд переваг над традиційними обчислювальними середовищами [2]. Зокрема, розподілені системи зменшують ризики, пов'язані з наявністю єдиної точки відмови системи, підвищуючи її надійність та стійкість до відмов. Сучасні розподілені системи, як правило, розроблені так, що надають користувачеві можливість масштабувати систему в режимі реального часу, надаючи інструменти для легкого підключення додаткових обчислювальні ресурсів у момент безпосередньої роботи системи, це підвищує продуктивність та скорочує час, що витрачається на підтримку системи.

Історично розподілені обчислення були дорогими та складними у налаштуванні та управлінні. Але завдяки платформам програмного забезпечення (SaaS) [3], які пропонують розширені функціональні можливості, розподілені обчислення стали більш доступними для великих та малих підприємств. У результаті всі типи обчислювальних завдань – від управління базами даних до відеоігор – використовують розподілені обчислення. Деякі типи програмного забезпечення, такі як криптовалютні системи, системи наукового моделювання, технології блокчейн та платформи штучного інтелекту, взагалі були б неможливими без задіяння розподілених обчислень у своїй розробці.

Сьогодні існує багато моделей та архітектур розподілених систем [4]. Системи типу клієнт-сервер, що є найбільш традиційними та простими з усіх типів розподілених систем, включають в себе безліч мережевих комп’ютерів, які взаємодіють із центральним сервером для зберігання, обробки даних чи інших загальних цілей. Мережі мобільних телефонів – це розширений тип розподіленої системи, що розподіляє робоче навантаження між телефонами, системами комутації та пристроями в Інтернеті. Інші приклади архітектури розподіленої системи – це однорангові мережі, в яких робоче навантаження розподіляється між сотнями або

тисячами комп’ютерів, на яких працює одне і те ж програмне забезпечення. Найпоширенішими формами розподілених систем на підприємстві сьогодні є ті, які працюють через Інтернет, передаючи навантаження десяткам екземплярів віртуальних серверів на основі хмар, які створюються за потребою, а потім припиняються, коли завдання буде виконано.

Розподілена система, також відома як система розподілених обчислень, – це система з безліччю компонентів, розташованих на різних машинах, які обмінюються даними та координують дії для того, щоб для кінцевого користувача представляти одну цілісну систему [5]. Машинами, які є частиною розподіленої системи, можуть бути комп'ютери, фізичні сервери, віртуальні машини, контейнери або будь-який інший вузол, який може підключатися до мережі, мати локальну пам'ять і підтримувати зв'язок, передаючи повідомлення.

Виходячи із вимог до сучасних систем, таких як мультиплатформеність, стабільність та керованість, більшість розробників прагнуть перейти від монолітної структури до мікросервісної [6]. Один із перших кроків, який можна зробити, не вдаючись до рефакторингу і декомпозиції, це досягнути горизонтальної масштабованості системи. Тобто, перетворити монолітний додаток в кластер, що може складатися з тих же самих монолітів, але дозволяє динамічно варіювати їх кількість, що гарантує стабільність роботи у випадку відмови будь-якого з вузлів і надає можливість безперервного функціонування системи.

Для того, аби вирішити задачу консистентності даних в усіх вузлах горизонтально масштабованої системи, було розроблено алгоритми консенсусу [7]. Вони вирішують задачу отримання узгодженого значення групою учасників у ситуації, коли можливі відмови деяких вузлів, надання недостовірної інформації або спотворення переданих значень середовищем передачі даних.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Поняття розподілених систем

Розподілена система [8] – це група комп'ютерів (вузлів, серверів), які можуть обмінюватися повідомленнями. Кожен окремий вузол – це деяка автономна сутність. Вузол може самостійно обробляти завдання, але щоб взаємодіяти з іншими вузлами, йому потрібно посилати і приймати повідомлення.

Кожна розподілена система має ряд обов'язкових атрибутів [8]:

– багатопоточність – можливість виникнення одночасних або конкурентних подій в системі. У контексті роботи з розподіленими системами вважається, що події, що відбулися на двох різних вузлах, потенційно конкурентні доти, поки немає чіткого порядку виникнення цих подій;

– відсутність глобальних годин – відсутність можливості покладатися на час у прямому його розумінні. У розподілених системах час не є абсолютним, адже навіть у надточного атомного годинника є дрифт, і можливі ситуації, коли ми не можемо сказати, яка з двох подій відбулася раніше;

– незалежна відмова вузлів системи – кожен вузол може зупинити свою роботу і це не повинно впливати на загальну працездатність системи. Вузли, як і будь-які фізичні машини, не є вічними. Може вийти з ладу жорсткий диск, перезавантажитися віртуальна система у хмарному середовищі, може відбутися мережевий збій і надіслані повідомлення загубляться;

– моделі комунікації (моделі обміну повідомленнями) між вузлами – синхронна та асинхронна моделі комунікації. Синхронна модель – модель, при якій існує деяка кінцева відома дельта часу, за яку повідомлення гарантовано доходить від одного вузла до іншого. Якщо цей час вийшов, а повідомлення не прийшло, можна констатувати, що вузол вийшов з ладу. У такій моделі ми час очікування передбачуваний. Асинхронна модель – в асинхронних моделях вважається, що час очікування кінцевий, але не існує такої дельти часу, після

якої можна гарантувати, що вузол вийшов з ладу. Тобто час очікування повідомлення від вузла може бути як завгодно довгим.

1.2 Визначення поняття консенсусу

Перш ніж формально визначити поняття консенсусу, варто розглянути механізм State Machine Replication [9] (рисунок 1.1). Усі операції, що відбуваються на репліках машини, записуються у розподілений файл, що має назву «лог». Він повинен бути консистентним і містити ідентичні дані на всіх вузлах розподіленої системи. Коли якийсь з вузлів отримує нове значення, яке він має намір записати в лог, його завданням стає запропонувати це значення всім іншим вузлам, щоб лог оновився на всіх вузлах, і система перейшла у новий консистентний стан. При цьому важливо, щоб усі вузли погодилися, що запропоноване нове значення є коректним та прийняли його, і тільки в цьому випадку усі можуть записати це значення у свій лог.

Домовленість між вузлами і згода про єдине вірне прийняте значення і є консенсусом в розподіленій системі. Формально, алгоритм досягнення консенсусу (або алгоритм консенсусу) визначається як деяка функція, яка переводить розподілену систему зі стану А в стан Б. Причому цей стан прийнятий усіма вузлами, і всі вузли можуть його підтвердити. Задача полягає у визначенні такої функції, яка додатково матиме такі властивості: узгодженість, цілісність та термінованість.

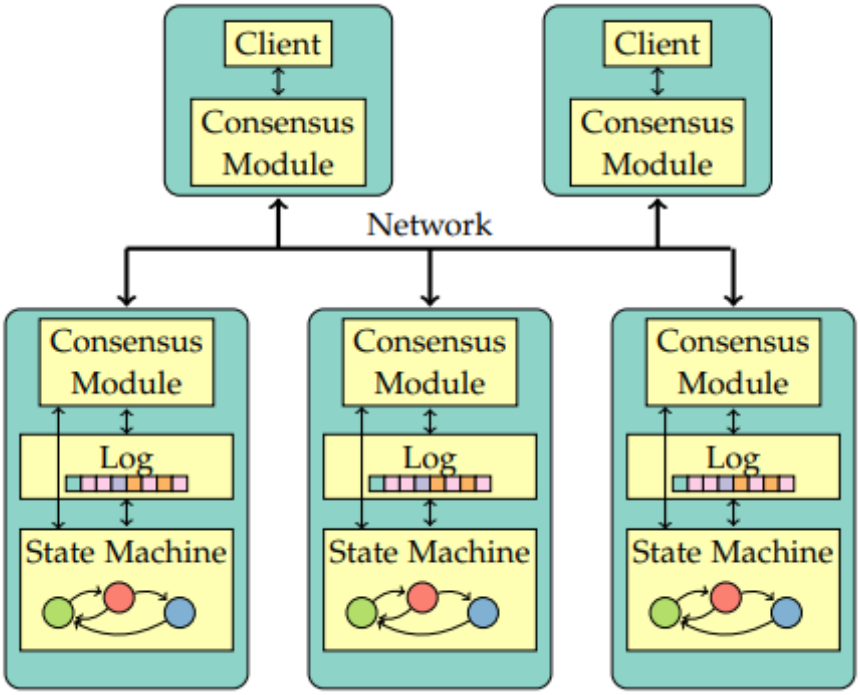


Рисунок 1.1 – Схема роботи машини станів

1.3 Опис задачі консенсусу

Виходячи з вищенаведених тверджень, задача полягає у розробці такого алгоритму досягнення консенсусу, щоб в умовах роботи частково синхронної системи можливо було отримати узгоджене значення, коли потенційно можливі відмови деяких вузлів.

Будь-який алгоритм консенсусу повинен відповідати наступним трьом умовам [10]:

– agreement (узгодженість) – усі коректно працюючі вузли повинні прийняти одне і те ж значення. Усі вузли, які функціонують (не вийшли з ладу і не втратили зв'язок з іншими) повинні прийти до згоди і прийняти деяке фінальне загальне значення;

– integrity (цілісність) – якщо усі коректно працюючі вузли пропонують одне і те ж значення v, отже кожен коректно працюючий вузол повинен прийняти це значення v;

– termination (кінцевість) – усі коректно працюючі вузли в кінцевому рахунку приймуть деяке значення, що дозволяє алгоритму мати прогрес в системі.

У 1985 році дослідники Фішер, Лінч та Паттерсон (FLP) [11] дійшли до висновку, що консенсус в асинхронній системі з несправними вузлами неможливий. Дане твердження сприяло тому, що практичні рішення цієї проблеми, які враховують результат FLP, є наразі одним із основних напрямків дослідження розподілених обчислень. Адже теоретичне “консенсус у асинхронній системі неможливий” на практиці означає, що “консенсус у асинхронній системі не завжди можливий” і існують такі практичні рішення, які дозволяють системі працювати і досягати консенсусу – мається на увазі частково синхронна система, у якій можливі лише відмови одного або декількох вузлів.

1.4 Огляд існуючих рішень для прийняття консенсусу у розподілених системах

Консенсус є центральним концептом для сучасних розподілених систем, це яскраво демонструє офіційне порівняння між реальними його реалізаціями у праці Consensus in the Cloud: Paxos Systems Demystified [12], що представлено на рисунку 1.2:

		Usage Patterns								
Project	**Consensus System**	**SR**	**LR**	**SS**	**BO**	**SD**	**GM**	**LE**	**MM**	**Q**
GFS	Chubby			✓				✓	✓	
Borg	Chubby/Paxos	✓				✓		✓		
Kubernetes	etcd						✓		✓	
Megastore	Paxos		✓							
Spanner	Paxos	✓								
Bigtable	Chubby						✓	✓	✓	
Hadoop/HDFS	ZooKeeper	✓						✓		
HBase	ZooKeeper	✓		✓			✓		✓	
Hive	ZooKeeper			✓					✓	
Configerator	Zeus								✓	
Cassandra	ZooKeeper					✓		✓	✓	
Accumulo	ZooKeeper		✓	✓					✓	
BookKeeper	ZooKeeper						✓		✓	
Hedwig	ZooKeeper						✓		✓	
Kafka	ZooKeeper						✓	✓	✓	
Solr	ZooKeeper							✓	✓	✓
Giraph	ZooKeeper		✓		✓				✓	
Hama	ZooKeeper				✓					
Mesos	ZooKeeper							✓		
CoreOS	etcd					✓				
OpenStack	ZooKeeper					✓				
Neo4j	ZooKeeper			✓				✓		

Рисунок 1.2 – Порівняльна характеристика алгоритмів консенсусу

Незважаючи на те, що станом на 2021 рік представлена велика кількість алгоритмів прийняття консенсусу, лише два з них домінують у кількості реалізацій у робочих системах – це алгоритм Paxos і алгоритм Raft. Частина відомих алгоритмів, таких як алгоритм прийняття консенсусу Чандри-Хоег, вважаються застарілими і більше не досліджуються. Також широко розповсюджені алгоритми прийняття консенсусу у блокчейні (PoW, PoS, DBFT) [13], проте вони не знаходять застосування у кластерах реальних систем, оскільки обробляють лише часткові випадки і мають вузьку спеціалізацію.

Лампорт запропонував алгоритм Paxos [14] для гарантування узгодженості розподілених систем, що є загальним протоколом консенсусу для асинхронних середовищ і, як було доведено, забезпечує теоретичний розподілений консенсус. Paxos не має основного вузла, тому кожен запит є новим екземпляром Paxos, і він

повинен пройти два етапи, щоб бути прийнятим більшістю вузлів. Для вирішення проблеми, через яку Paxos не може обслуговувати послідовні запити, що часто вимагається на практиці, багато робіт присвячено покращенню алгоритму Paxos. Multi-Paxos [15] вилучає фазу пропозиції, обираючи лідера, який може постійно обслуговувати запити. Як результат, Multi-Paxos широко використовується в системах розподілених баз даних, таких як Google Spanner [16] та MegaStore [17], Tencent's Paxos Store [18] та IBM Spinnaker [19].

Незважаючи на це, Paxos досі важко зрозуміти та правильно реалізувати. Raft [20] – це консенсусний протокол, розроблений у першу чергу із метою бути більш зрозумілим. Він розділяє протокол на вибори лідера та реплікацію логів. Вузол–лідер відповідальний за обробку запитів та тиражування результатів послідовниками. Raft вводить обмеження щодо того, що логи повинні оброблятися послідовно. Це обмеження полегшує розуміння Raft, але також обмежує продуктивність одночасної реплікації. Через простоту реалізації багато систем, таких як etcd [21], TiDB [22] та PolarFS [23], використовують Raft для забезпечення високої доступності.

Сувора серіалізація алгоритму Raft обмежує багатопоточність систем. Багато дослідників працюють над цим, щоб покращити ефективність роботи Raft. PloarFS [24] запропонував ParallelRaft, який підтримує паралельну реплікацію логів. Але він призначений для файлових систем, тому реплікація у невизначеному порядку реалізується шляхом запису адреси блоку попередніх N записів у кожному записі логу. Перевірки конфліктів записів недостатньо для семантики проведення транзакції, а підтримка набору записів із N записів у кожному лозі є надто дорогою операцією для пам'яті та мережі. TiDB оптимізує Raft по-іншому, він ділить дані на розділи, і кожен розділ є єдиною групою Raft для реплікації. Використовуючи багато груповий Raft, він врівноважує навантаження на декількох серверах, що покращує паралельність усієї системи. Вайбхав та інші [25] у своїй роботі обговорили проблему послідовного читання фоловерів, щоб фоловери могли поділити навантаження лідера, аби уникнути того, щоб лідер став вузьким місцем у роботі всієї системи. DARE [26] дав інший спосіб прискорити реплікацію за допомогою апаратного забезпечення. Він

використовує RDMA, щоб переробити протокол, подібний до Raft, для досягнення високої продуктивності.

Висновок до розділу

У даному розділі було розглянуто поняття розподілених систем та визначено основні їх атрибути, що є обов'язковими. Наведено визначення поняття консенсусу, визначено постановку задачі знаходження консенсусу між вузлами у розподіленій системі та властивості алгоритмів, що таку задачу вирішують. Було проаналізовано дослідження, що порівнює існуючі на даний момент алгоритми досягнення консенсусу та їх реалізації, зроблено висновки про ефективність роботи та простоту реалізації таких алгоритмів, їхні переваги та недоліки.

2 МОДЕЛІ ТА МЕТОДИ ДОСЯГНЕННЯ КОНСЕНСУСУ У РОЗПОДІЛЕНИХ СИСТЕМАХ

2.1 Задача візантійських генералів

У 1982 Леслі Лампорт написав працю [27] про задачу візантійських генералів. У ній він абстрактно описав проблему консенсусу, уникаючи технічних термінів, аби зобразити її максимально простою і зрозумілою.

Нехай маємо дві армії – одна позначена рудим кольором і друга, що позначена білим кольором. Білі війська базуються у зайнятому ними ворожому місті. Руді війська на чолі з генералами A1 і A2 розташовуються по двох сторонах від цього міста. Завдання рудих – напасти на біле місто і перемогти. Однак військо кожного рудого генерала окремо менше війська білих (рис 2.1).

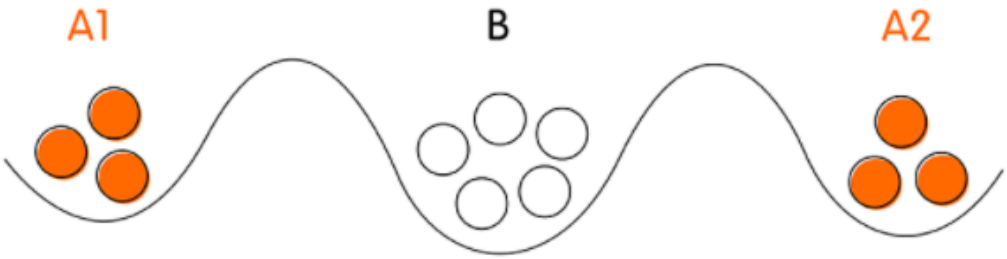


Рисунок 2.1 – Ілюстрація до задачі візантійських генералів

Умови перемоги для рудих: обидва генерала повинні напасти одночасно, щоб мати чисельну перевагу над білими. Для цього генералам A1 та A2 необхідно домовитися один з одним про час спільного нападу. Якщо кожен буде нападати окремо, руді програють.

Щоб домовитися, генерали A1 і A2 можуть посилати один до одного гінців через територію білого міста. Гінець може успішно дістатися до союзного генерала або може бути перехоплений білим противником. Питання полягає у тому, чи існує така послідовність комунікацій між рудими генералами (послідовність відправки гінців від A1 до A2 і навпаки від A2 до A1), при якій вони гарантовано домовляться

про напад на годину X. Під гарантіями розуміється, що обидва генерали будуть мати однозначне підтвердження, що союзник (інший генерал) точно атакує в призначений час X.

Припустимо, що A1 посилає гінця до A2 з посланням: «Давай нападемо сьогодні опівночі!». Генерал A1 не може напасти без підтвердження від генерала A2. Якщо гінець від A1 дійшов, то генерал A2 посилає підтвердження з повідомленням: «Так, згода!». Але тепер генерал A2 не знає, дійшов його гінець чи ні, у нього немає гарантій, що напад буде одночасним. Тепер вже генералу A2 знову потрібно підтвердження.

Якщо розписувати їх комунікацію далі, з'ясується наступне: скільки б не було циклів обміну повідомленнями, немає способу гарантовано повідомити обох генералів про те, що їх повідомлення отримано (за умови, що будь-який з гінців може бути перехоплений).

Завдання двох генералів – це відмінна ілюстрація дуже простої розподіленої системи, де є два вузла з ненадійною комунікацією, де немає однозначної гарантії того, що вони синхронізуються. Виходячи з даної задачі, можна прийти до висновку, що для того, щоб на практиці система вважалася толерантною до падінь, вона мусить мати як мінімум ⅔ надійних вузлів (більшість).

2.2 Простий приклад роботи алгоритму консенсусу у синхронній моделі

Дали розглянуто приклад роботи найпростішого алгоритму консенсусу у системі з синхронною моделлю обміну повідомленнями, у якій всі вузли працюють коректно, повідомлення не губляться та не спотворюються:

1) першою операцією у алгоритмі консенсусу є пропозиція (Propose) деякого значення. Припустимо, що до вузла під назвою «Вузол 1» підключився клієнт і почав транзакцію, передавши вузлу нове значення – O. З цього моменту «Вузол 1» будемо називати proposer. Як proposer «Вузол 1» тепер повинен сповістити всю систему про те, що у нього є нові актуальні дані, і він розсилає всім іншим вузлам повідомлення виду: «Отримав значення «O», хочу його записати. Прошу підтвердити, що ви теж запишете «O» в свій лог»;

2) наступна стадія – голосування за запропоноване значення (Voting). Може так статися, що іншим вузлам прийшла більш свіжа інформація, і у них є дані по цій же транзакції. Тому голосування необхідне. Коли вузол «Вузол 1» посилає свою пропозицію, інші вузли перевіряють в своїх логах дані по цій події. Якщо ніяких протиріч немає, вузли оголошують: «Так, у мене немає інших даних по цій події. Значення «О» – це найактуальніша інформація». У будь-якому іншому випадку, вузли можуть відповісти «Вузлу 1»: «У мене є більш актуальні дані по цій транзакції, не приймаю запропоноване значення «О»». На стадії голосування вузли приходять до висновку: або усі приймають одне значення, або хтось із них голосує проти, позначивши, що у нього є більш актуальні дані;

3) якщо стадія голосування пройшла успішно, і всі були «за», то система переходить в нову стадію – прийняття значення (Accept). «Вузол 1» збирає всі відповіді інших вузлів і повідомляє: «Усі погодилися зі значенням «О». Тепер це офіційно найбільш актуальне значення, можете записати його у свій лог»;

4) решта вузлів надсилають підтвердження, що вони записали у свої логи значення «О», нічого нового за цей час надійти не встигло (свого роду двофазний коміт). Після цього можна вважати, що розподілена транзакція виконалася.

Таким чином алгоритм консенсусу в простому випадку складається з чотирьох кроків: пропозиція (propose), голосування (voting), прийняття (accept), підтвердження прийняття (accepted).

Якщо на якомусь етапі вузли не змогли досягти згоди, то алгоритм запускається заново, з урахуванням тієї інформації, яку нададуть вузли, які відмовилися підтверджувати запропоноване значення.

2.3 Алгоритми Paxos та Multi–Paxos для досягнення консенсусу

Paxos – це сімейство алгоритмів, які вирішують проблему консенсусу для частково синхронних систем, за умови що деякі вузли можуть виходити з ладу. Автором цього підходу є Leslie Lamport, що у 1989 році запропонував формальний

доказ існування і коректності алгоритму. Цей алгоритм є важливим у контексті розгляду проблеми консенсусу, адже усі наступні розроблені алгоритми були створені на його основі, а безліч популярних прикладних програм, таких як Apache Zookeeper [28], Apache Hadoop & Kafka [29], PhxPaxos [30], використовують його як базовий алгоритм консенсусу.

В алгоритмі Paxos кожен з вузлів обов'язково має деяку роль, або ж в залежності від ситуації – декілька ролей. Дали розглянуто три основні (існують також модифікації алгоритму з додатковими ролями):

– proposers – ініціатори (також можуть зустрічатися терміни: лідери або координатори). Це вузли беруть на себе роль лідера, коли отримують нове значення від користувача. Їх завдання запустити стадію пропозиції нового значення і координувати подальші дії вузлів. При цьому Paxos допускає наявність декількох лідерів в певних ситуаціях;

– acceptors (Voters) – виборці. Це вузли, які голосують за прийняття або неприйняття того чи іншого значення. Саме від них залежить в який стан перейде (або не перейде) система після чергового етапу алгоритму консенсусу;

– learners. Вузли, які отримують та записують нове прийняте значення, коли стан системи змінився. Вони не приймають рішення, просто отримують дані і можуть віддати їх кінцевому користувачу.

Передбачається, що для прийняття рішень вузли використовують обмін повідомленнями як метод комунікації. Paxos передбачає використання асинхронної моделі обміну повідомленнями, але без наявності “візантійських помилок”, тобто мається на увазі частково синхронна система. Це означає, що:

– вузли працюють з довільною швидкістю, можуть припинити свою роботу або перезапустится. Оскільки вузол може аварійно призупинитися після участі у прийнятті рішення по деякому значенню, але до того, як він, власне, дізнається про прийняте рішення, вузол не зможе дізнатися значення прийнятого рішення, якщо не буде способу персистентно зберегти його стан на період між завершенням голосування та перезапуском вузла;

– повідомлення можуть доставлятися з довільною швидкістю, можуть дублюватися або ж не бути доставленими, але їх зміст не може бути пошкодженим.

2.3.1 Процедура прийняття рішення

Одним із найпростіших способів прийняття рішення у поставлених умовах було б деяким чином визначитися на рахунок того вузла, який би завжди виступав у ролі виборця. Тоді б усі інші вузли могли б посилати йому свої значення і отримувати рішення – були вони прийняті чи ні. Виборець у свою чергу обирав би те значення, яке отримував би першим. Дане рішення хоч і просте, але на жаль неможливе на практиці, адже у разі виходу з ладу вузла-виборця, жодне рішення у системі більше не може бути прийнятим.

Іншим способом прийняття рішення було б замість одного виборця використовувати декількох. Тоді значення буде прийнятим, якщо більшість виборців погодилися прийняти його. Виборець в свою чергу не може прийняти більше двох значень, оскільки може виникнути ситуація, коли більшість голосів буде належати декільком пропозиціям одночасно.

Коли виборець отримує першу пропозицію, він не може гарантувати, що інші пропозиції не надійдуть. Особливо ця проблема є актуальною у середовищі, де повідомлення можуть бути загублені. Рішення також повинне бути прийнятим, коли є тільки один ініціатор. Звідси слідує наступне твердження:

T1: Виборець приймає першу пропозицію, яку він отримує.

Однак, може виникнути проблема, коли декілька значень пропонуються різними ініціаторами приблизно у той самий момент часу. Це може призвести до ситуації, коли кожен виборець прийняв рішення, але всі вони різні, і жодне з рішень у системі не набрало більшості голосів (рис 2.2).

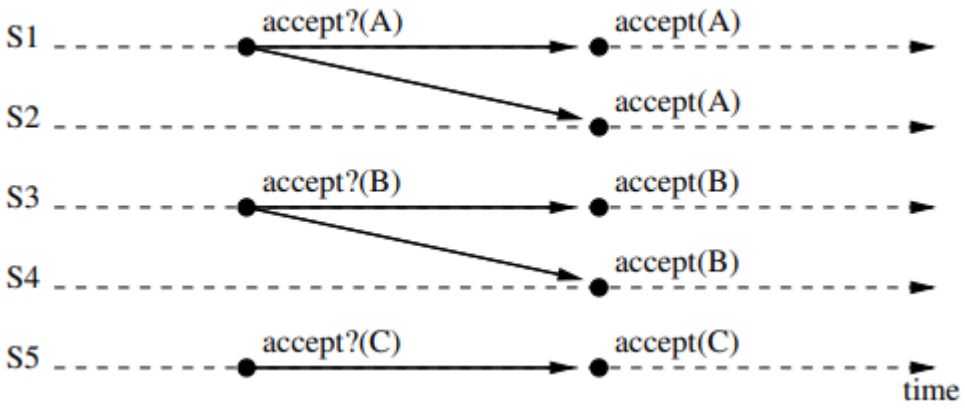


Рисунок 2.2 – Різні виборці прийняли різні значення, рішення не може бути прийнято

Умова T1 разом із твердженням, що будь-яке значення може бути прийнятим у системі тоді і тільки тоді, коли за нього проголосувала більшість виборців, призводить до того, що виборцям дозволяється приймати більше, ніж одне значення. Різні пропозиції можуть бути ідентифіковані виборцем з допомогою деякого числового ідентифікатора. Таким чином будь-яка пропозиція повинна мати вигляд (значення, номер пропозиції), а різні пропозиції повинні мати різні номери.

Як саме визначати номер пропозиції залежить від конкретної реалізації. Один із можливих варіантів – кожен вузол має унікальний ідентифікатор і зберігає у локальній змінній *max_round* максимальний номер раунду, який йому відомий. Для того, що запропонувати системі деяке значення, вузол інкрементує цю змінну на 1 і задає номер пропозиції як конкатенацію значень змінної та унікального ідентифікатора вузла.

Таким чином, у системі можуть бути прийняті декілька пропозицій, але тільки тоді, коли вони відрізняються лише номером, але містять одне і те ж значення (рис 2.3). Звідси слідує наступне твердження:

T2: Якщо пропозиція *(v, M)* (прийняти значення *v* з номером пропозиції *M*) прийнята, то будь-яка інша пропозиція з номером, більшим за *M*, прийнята цим же виборцем, повинна також містити значення *v*.

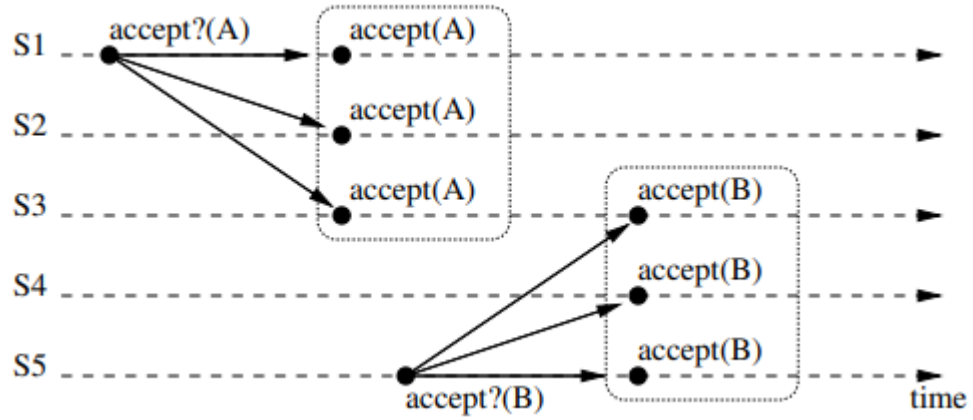


Рисунок 2.3 – Декілька значень у системі прийняті одночасно

Так як ідентифікатори пропозицій зростають, умова T2 гарантує, що обрано лише одне значення. З іншого боку, для того, щоб вважатися прийнятою, пропозиція мусить бути прийнятою принаймні одним виборцем. Звідси умова T2 може виконуватись, лише якщо:

T2': Якщо пропозицію *(v, M)* прийнято, то будь-яка інша пропозиція, що прийнята будь-яким виборцем і має номер, більший за *M*, має також пропонувати значення *v*.

Нехай пропозицію *(v, M)* було прийнято, тобто у системі існує така множина *C*, що складається із більшості вузлів, кожен з яких прийняв пропозицію *(v, M)*. Із індукційного припущення слідує, що усі пропозиції з номерами від *M* до *N–1* мають значення *v(*)*. Необхідно довести, що пропозиція з номером *N* також матиме значення *v*. За методом математичної індукції Лампорт доводить це і пропонує наступне твердження:

T3: Для будь-яких *v* і *N*, якщо у системі зроблено пропозицію *(v, N)*, існує така множина *S*, що складається із більшості виборців, і що вірне одне з двох тверджень:

– жоден виборець із множини *S* не прийняв жодної пропозиції з номером менше *N*;

– *v* – це значення пропозиції з найбільшим номером, меншим за *N* серед усіх пропозицій прийнятих виборцями множини *S*.

Для того, аби забезпечити існування вищевказаної множини *S*, ініціатор, що планує зробити пропозицію з номером *N*, повинен дізнатися у більшості виборців

номер і значення останньої пропозиції, яку вони приймуть до моменту N. Однак існує шанс, що після того, як виборець відправить свої дані, він прийме ще одне рішення, і відправлені дані уже не будуть актуальними. Щоб запобігти цьому, Лампорт пропонує виборцю не приймати ніяких пропозицій, з номером, меншим за N. Як тільки множину S буде сформовано, ініціатор робить пропозицію (v, N).

2.3.2 Фази алгоритму Paxos

Алгоритм Paxos передбачає дві великі фази, кожна з яких в свою чергу поділена на два кроки. Разом ці фази формують увесь процес роботи алгоритму, приклади сценаріїв якого зображені на малюнку 2.4.

– phase 1a: Prepare. На етапі підготовки лідер (proposer) повідомляє усі вузли про те, що починається новий етап голосування. Номер цього раунду – N. Поки він просто повідомляє про початок нового циклу, але не повідомляє нове значення. Завданням цього етапу є ініціювати новий раунд і повідомити всім його унікальний номер. Номером раунду повинно бути значення більше, ніж всі попередні номери голосувань від всіх попередніх лідерів. Саме завдяки номеру раунду інші вузли в системі будуть розуміти, наскільки свіжі дані у лідера.

– phase 1b: Promise. Коли вузли–виборці отримали номер нового етапу голосування, можливі два результати:

a. номер N поточного голосування більший, ніж номер будь-якого з попередніх голосувань, в якому брав участь виборець. Тоді виборець відправляє лідеру Promise (обіцянка, підтвердження), що не буде брати участь ні в яких голосуваннях з меншим номером, ніж N. Якщо виборець вже встиг за щось проголосувати (тобто він вже в другій фазі прийняв якесь значення), то до своєї обіцянки він прикладає прийняте значення і номер голосування, в якому він брав участь;

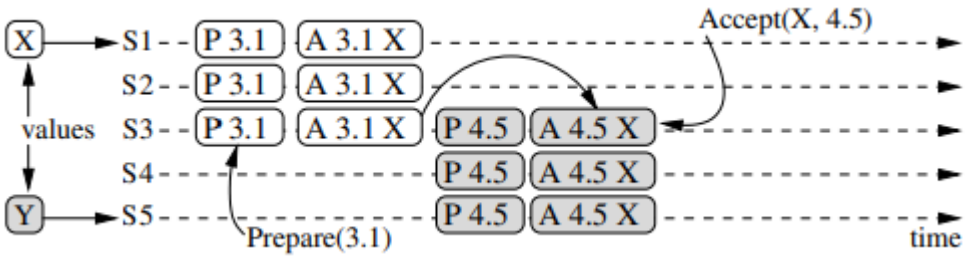
b. в іншому випадку, якщо виборець вже знає про голосування з номером, більшим за номер поточного голосування, він може просто проігнорувати даний етап і не відправляти лідеру жодної відповіді.

– phase 2a: Accept. Лідеру необхідно дочекатися відповіді від кворуму Q (більшості вузлів в системі) і, якщо потрібне число відповідей отримано, то існує два подальші варіанти розвитку подій:

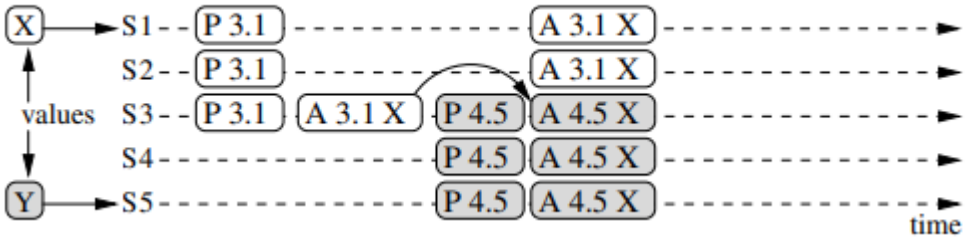
a. деякі з вузлів-виборців прислали значення, за які вони вже голосували. У цьому випадку лідер обирає значення з того голосування, яке має максимальний номер раунду. Нехай це значення x, і розсилає всім вузлам повідомлення виду: «Accept (x, N)»;

b. якщо жоден з вузлів–виборців не надіслав жодних значень, а лідер отримав лише обіцянки голосувати в цьому раунді, лідер може запропонувати їм проголосувати за своє значення, яке він отримав від користувача. Нехай це значення рівне y. Він розсилає всім вузлам повідомлення виду: «Accept (y, N)», за аналогією з попереднім пунктом.

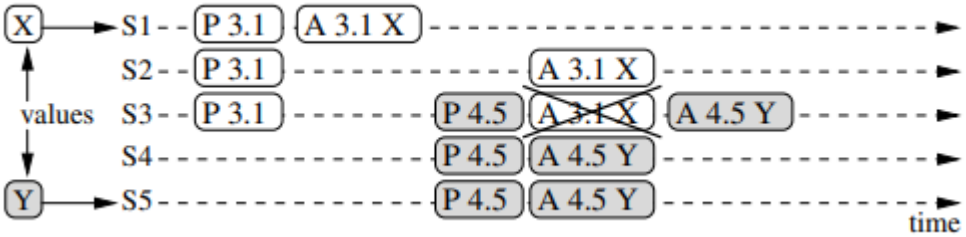
– phase 2b: Accepted. На наступному етапі вузли–виборці, при отриманні повідомлення «Accept (...)» від лідера, погоджуються з ним (розсилають всім вузлам підтвердження, що вони згодні з новим значенням) тільки в тому випадку, якщо вони не обіцяли якомусь (іншому) лідеру брати участь в голосуваннях з номером раунду $N' > N$, в іншому випадку вони ігнорують запит на підтвердження. Якщо лідеру відповіла більшість вузлів, і всі вони підтвердили нове значення, то нове значення вважається прийнятим. Якщо ж більшість голосів не набрано або є вузли, які відмовилися приймати нове значення, то увесь цикл прийняття консенсусу запускається з початку.



(а) Значення уже прийнято, новий ініціатор дізнається про нього і змінює своє пропоноване значення



(б) Значення ще не прийнято, але ініціатор дізнається про нього у запиті Prepare. Новий ініціатор користується значенням, що було запропоноване раніше. Обидва ініціатори успішно завершують голосування.



(в) Попереднє значення не вибрано і не відомо поточному ініціатору. Новий ініціатор використовує власне значення, голосування за попередньою пропозицією заблоковано.

Рисунок 2.4 – Приклади сценаріїв роботи Paxos

2.3.3 Перехід до Multi-Paxos

Алгоритм Paxos, що також називають Single-Paxos, застосовується для прийняття поодинокого рішення. Для розв'язання задачі реплікації розподіленого логу використовують алгоритм Multi-Paxos [31]. Він додає до алгоритму ще одного учасника – клієнта, від якого приходять деякі значення, за які необхідно голосувати.

Клієнтів може бути декілька, вони мусять мати можливість перевіряти роботу кластера, а також можуть в будь-який момент відключатися. Основною проблемою даного алгоритму є збереження унікального значення у реплікованому журналі, а також необхідність фонової реплікації для вузлів, що не відповідають.

Якщо Single-Paxos вирішує задачі того, що деякі учасники повинні домовитись про деяке єдине значення, то в Multi–Paxos виникає ще один учасник – клієнт, який надсилає ці значення, що пізніше висуваються на голосування та приймаються у системі. Також клієнт повинен мати можливість дізнатися, чи було прийняте його значення кластером. Клієнтський вузол також може відмовити, що може призвести до ситуації, коли він не може дізнатися, чи було прийняте значення, запропоноване ним раніше.

Більше того, клієнтів може бути декілька, що вимагає використання реплікованого журналу задля усунення можливих протиріч – якби для різних клієнтів просто проводились паралельні незалежні голосування, це могло б призвести до того, що порядок результатів голосувань був би неузгодженим для всього кластера. Інакше кажучи, для частини серверів спочатку було б прийняте значення A, а потім B, а для іншої частини серверів – навпаки.

Сценарій роботи реплікованого кінцевого автомату виглядає наступним чином:

1) клієнт посилає команду для виконання на сервер;
2) використовуючи Paxos, сервер реплікує запит клієнта на усі машини, усі команди реплікуються у фіксованому для всіх серверів порядку;
3) сервер очікує прийняття рішення по всім попереднім записах в журналі, після чого застосовує їх, виконуючи попередні команди клієнтів;
4) сервер застосовує вибрану команду і повертає результат клієнту.

Для того, щоб зберігати команди в фіксованому глобальному порядку, кожна пропозиція повинна мати нову змінну – індекс в реплікованому журналі. Однією із задач алгоритму є забезпечити її унікальність.

Для цього припустимо, що у нас є ініціатор A. Із локального журналу йому відомо по яких пропозиціях уже були прийняті рішення, а отже, номери їх позицій точно не можуть бути використані у якості індексу. Також ініціатору варто

виключити ті номери, за якими він уже запустив затвердження пропозицій, але вони ще не є остаточно прийнятими. Таким чином варто знайти перший вільний індекс і спробувати затвердити його – виконати *Prepare* пропозицію з цим індексом як одним із параметрів (позицією в журналі).

У відповідь на даний запит ініціатор може отримати інформацію про те, що запис з таким номером уже використовується якимось вузлом, отже треба повторити пропозицію, але збільшити її номер на 1. Якщо ж ініціатор отримує у відповідь інформацію про те, що даний запис не використовується, він може затвердити її і почати новий цикл обробки запитів клієнта.

Таким чином, алгоритм вибору індекса пропозиції виглядає наступним чином:

1) знайти вільний індекс у журналі;
2) виконати *Prepare* запит Paxos–алгоритму з цим індексом;
3) перевірити результат *Prepare* запиту:
   a) якщо у відповідь сервер повернув інше значення, переключитися на нього, записати це значення у свій журнал, після чого повернутися на крок 1;
   b) якщо повертається запропоноване значення, виконати Accept-запит, отримати нову команду клієнта і повернутися на крок 1.

2.4 Алгоритм Raft для досягнення консенсусу

Raft – алгоритм прийняття консенсусу для керування розподіленим логом. Він був розроблений на основі Multi-Paxos, але має іншу архітектуру та є відносно легким для застосування на практиці.

2.4.1 Передумови роботи алгоритму

При розгляді роботи алгоритму Raft досягнення консенсусу припускається, що виконуються наступні положення:

1) мережевий зв’язок між вузлами є ненадійним, включаючи затримки мережі, розділи та втрату пакетів, дублювання та зміну порядку повідомлень;

2) вузли мають доступ до нескінченної постійної пам’яті, яку неможливо пошкодити, і будь-який запис у постійну пам’ять буде завершено до збою;

3) вузли працюють в асинхронному середовищі з довільною швидкістю, верхньої мети затримки повідомлень не існує;

4) візантійські невдачі не можуть відбуватися;

5) вузли статично налаштовані так, що знають про всі інші вузли кластера, членство в кластері не може змінюватись динамічно;

6) протокол використовує нескінченно великі монотонно зростаючі значення;

7) клієнти програми повинні спілкуватися з кластером через поточного лідера. Клієнти мають статичну конфігурацію зі знанням усіх вузлів;

8) машини стану, що працюють на кожному вузлі, запускаються в одному стані і детерміновано реагують на операції клієнта.

2.4.2 Основні поняття алгоритму Raft

Основні елементи та умови роботи алгоритму Raft схожі з алгоритмом Paxos, проте існують деякі відмінності у його термінології та визначеннях.

1) Стан серверів (*SS*)

У кластері кожен з серверів в будь-який момент часу знаходиться в одному з трьох станів:

– leader (лідер) – обробляє усі клієнтські запити, є джерелом правди усіх даних в логах фоловерів;

– follower (фоловер) – пасивний сервер, який тільки приймає нові записи в лог від лідера і перенаправляє усі вхідні запити від клієнтів на лідера;

– candidate (кандидат) – спеціальний стан сервера, можливий тільки під час вибору нового лідера.

Під час нормальної роботи в кластері тільки один сервер є лідером, усі решта – його фоловери. Події у системі призводять до того, що вузли змінюють свій стан [32], як показано на рисунку 2.5.

2) Кворум (*Q*)

Нехай маємо кластер, що складається із N вузлів. Із них максимум F вузлів можуть вийти з ладу. Якщо F вузлів виходять з ладу, це означає, що в кластері має бути мінімум $2F + 1$ фоловерів. Це потрібно для того, щоб коректно працюючі вузли завжди мали більшість, звідси $Q = F + 1$.

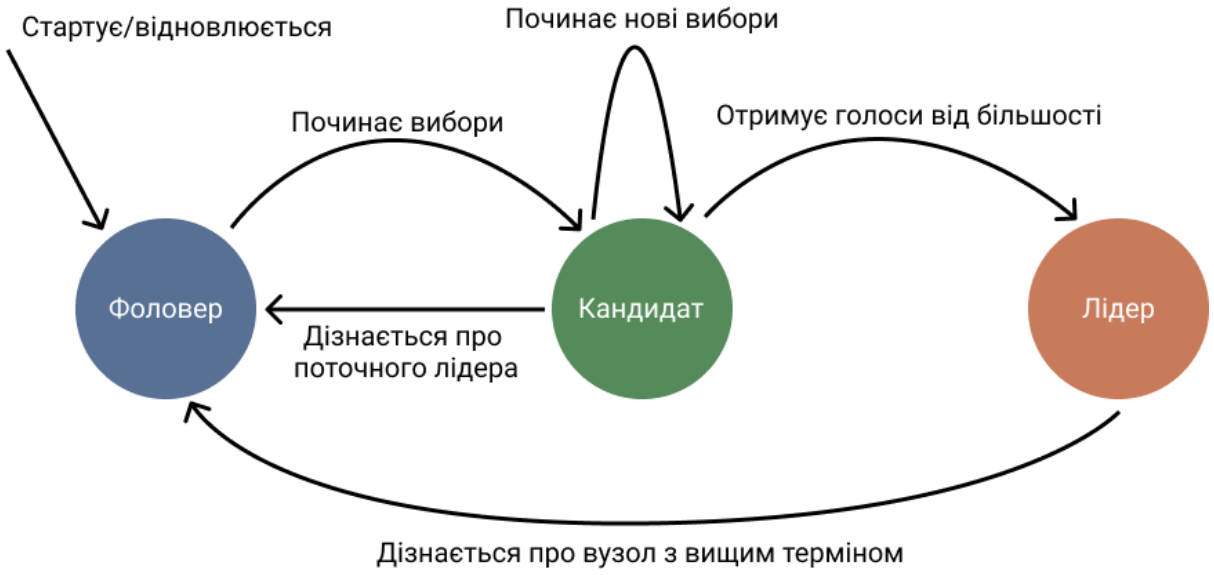


Рисунок 2.5 – Зміна станів вузлів системи

3) Терміни (T)

У контексті розгляду алгоритму Raft, вважається, що час у системі поділений на відрізки довільної довжини, що називаються термінами [33]. Кожен термін має монотонно зростаючий номер. Термін починається з виборів лідера, коли один або кілька серверів стають кандидатами. У разі, якщо кандидат отримує більшість голосів, він стає лідером до кінця цього терміну. Якщо ж голоси розділилися, і жоден з кандидатів не може отримати більшість голосів, спрацьовує таймаут, і цей термін закінчується (рис 2.6). Після цього починається новий термін з новими кандидатами і виборами. Номер терміну служить в якості логічної часової відмітки в кластері. Він допомагає серверам визначати, яка інформація є більш актуальною на поточний момент.

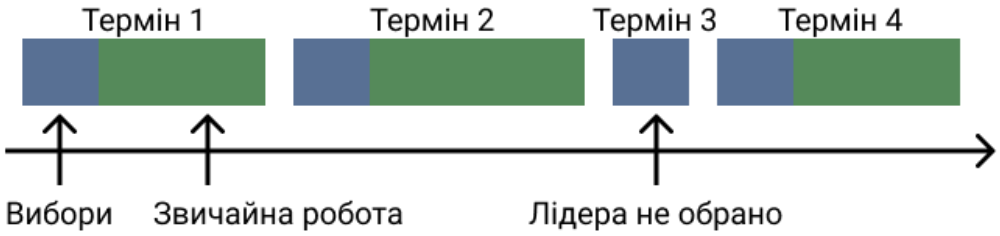


Рисунок 2.6 – Терміни у розподіленій системі

4) Тип виклику сервера (*R*)

Сервери взаємодіють один з одним за допомогою обміну запитами та відповідями. Базовий алгоритм використовує два види викликів:

– requestVote – використовується кандидатами під час виборів. Запит містить номер терміну кандидата і метадані про лог кандидата. Відповідь містить номер терміну відповідаючого сервера і значення «true», якщо сервер голосує за кандидата; «false», якщо сервер голосує проти кандидата;

– appendEntries – використовується лідером для реплікації логу, а також для механізму heartbeat (перевірка працездатності серверів). Запит містить номер терміну лідера, колекцію записів, які потрібно додати в лог (або порожню колекцію в разі heartbeat), деякі метадані про лог лідера. Відповідь містить номер терміну фоловерів і значення «true», якщо фоловер успішно додав записи в свій лог; «false», якщо додати записи в лог не вдалося.

2.4.3 Основні етапи роботи алгоритму

Алгоритм Raft умовно ділить процес досягнення консенсусу на два етапи:

1. Вибір лідера

Для визначення моменту, коли пора починати нові вибори, Raft покладається на heartbeat. Фоловер залишається фоловером до тих пір, поки він отримує повідомлення від чинного лідера або кандидата. Лідер періодично розсилає всім іншим серверам запит на перевірку heartbeat.

Якщо фоловер не отримуватиме жодних повідомлень деякий час, він цілком закономірно припустить, що вузол–лідер відмовив, і стане ініціатором виборів. Для цього він збільшує на 1 свій номер терміну, переходить у стан «кандидат», голосує сам за себе і розсилає запит «RequestVote» всім іншим серверам. Після цього кандидат чекає однієї з трьох подій:

- кандидат отримує більшість голосів (включаючи свій) і перемагає у виборах. Кожен сервер голосує в кожному терміні лише одного разу, за першого кандидата, від якого отримає запит, тому набрати в конкретний термін більшість голосів може тільки один кандидат. Сервер, що переміг, стає лідером, починає розсилати heartbeat і обслуговувати запити клієнтів до кластеру (рис 2.7);
- кандидат отримує повідомлення від вже чинного лідера поточного терміну або від будь-якого сервера більш старшого терміну. У цьому випадку кандидат розуміє, що вибори, в яких він бере участь, вже не актуальні. Йому не залишається нічого, окрім як визнати нового лідера / новий термін і перейти в стан фоловера (рис 2.8);
- кандидат не одержує за деякий таймаут більшість голосів. Таке може статися у разі, коли кілька фоловерів стають кандидатами, і голоси розподіляються серед них так, що жоден з них не отримує більшості. У цьому випадку термін закінчується без лідера, а кандидат відразу ж починає нові вибори на наступний термін (рис 2.9).

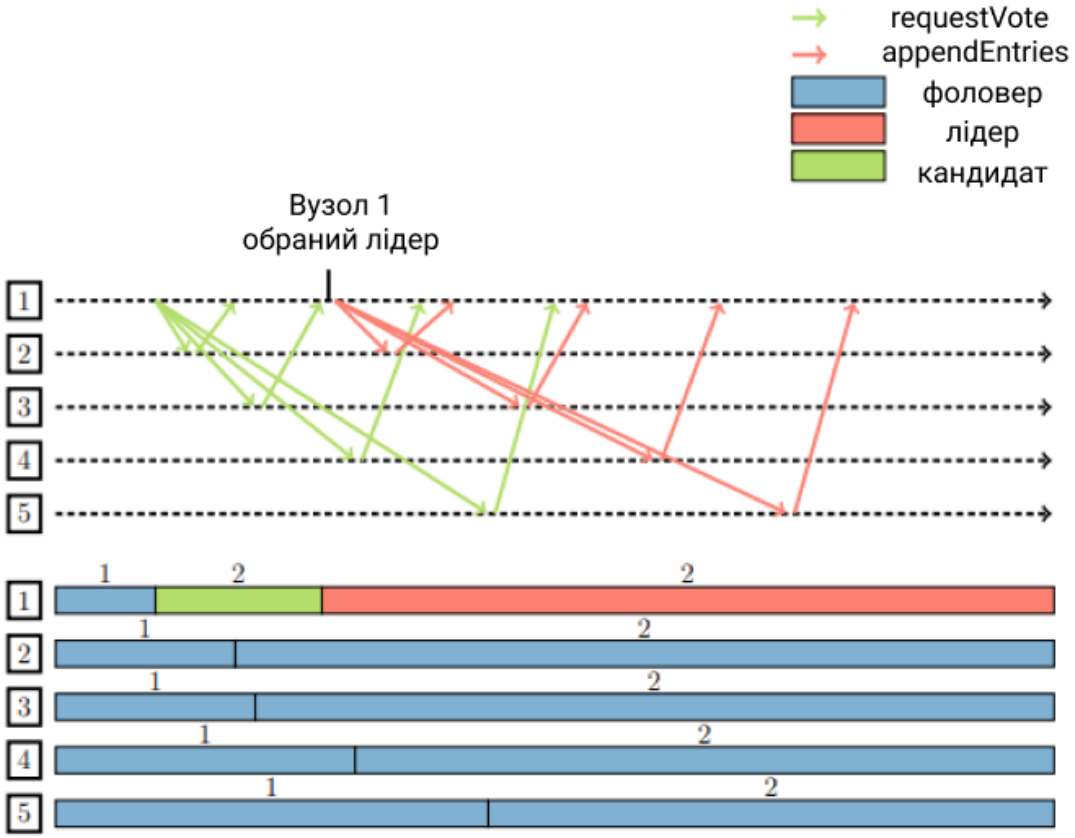


Рисунок 2.7 – Приклад успішних виборів: усі вузли починають свою роботу і входять у термін 1 як фоловери. Вузол 1 першим досягає свого тайм–ауту і стає кандидатом у терміні 2. Вузол 1 передає RequestVotes всім іншим вузлам, які відповідають позитивно, надаючи вузлу 1 свій голос за термін 2. Після того, як вузол 1 отримує 3 голоси, він стає лідером і відправляє heartbeat AppendEntries запит на всі інші вузли.

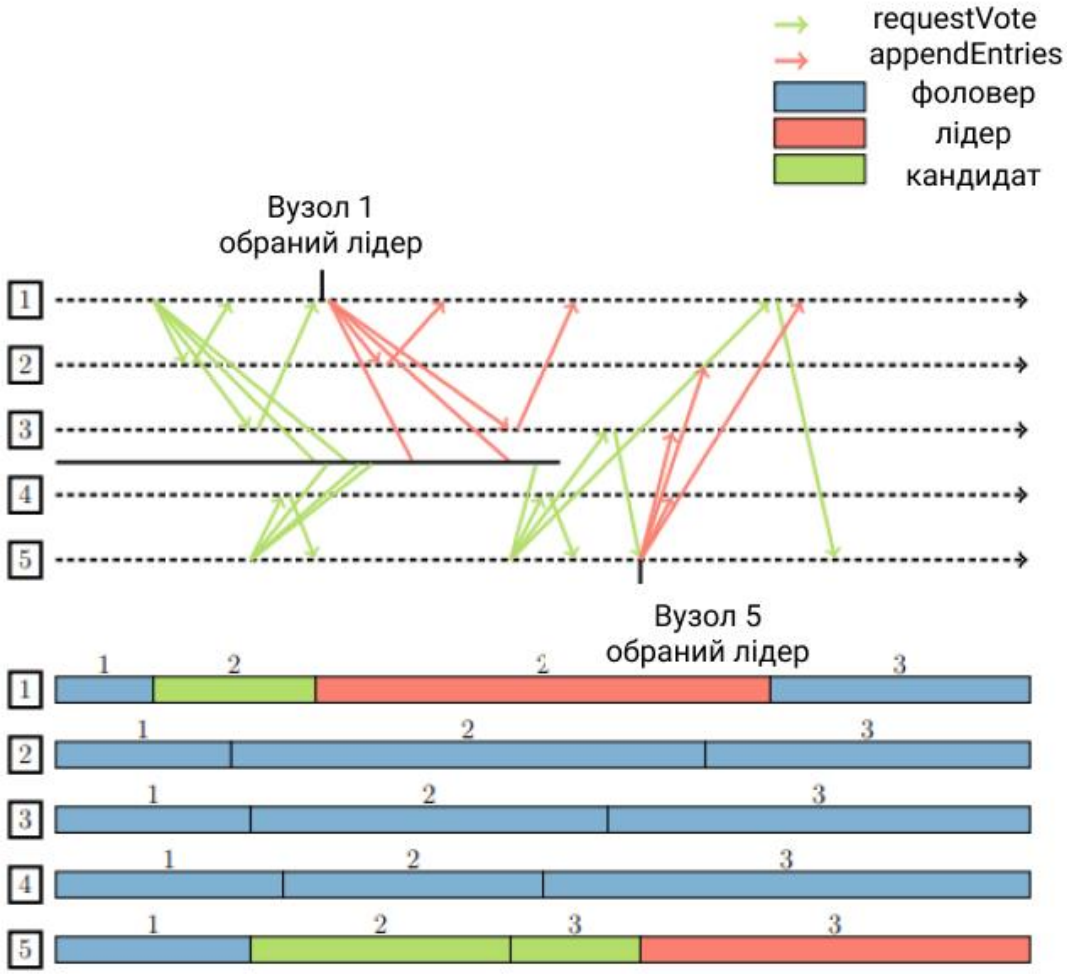


Рисунок 2.8 – Приклад виборів, під час яких система зіткнулась з розділенням мережі, що в кінцевому підсумку призвело до відставки лідера.

Цей кластер починає роботу з розділу між вузлами 1/2/3 і 4/5. Вузли 1 і 5 кожен тайм–аут і починають вибори на термін 2. Вузол 1 обирається лідером вузлами 2 і 3, але вузол 5 не може отримати голоси від більшості вузлів. Після усунення розділу вузол 5 перезапускає вибори в терміні 3. Вузли з 1 по 4 віддають голоси вузлу 5, оскільки термін тепер дорівнює 3, тому вузол 5 обирається лідером у терміні 3.

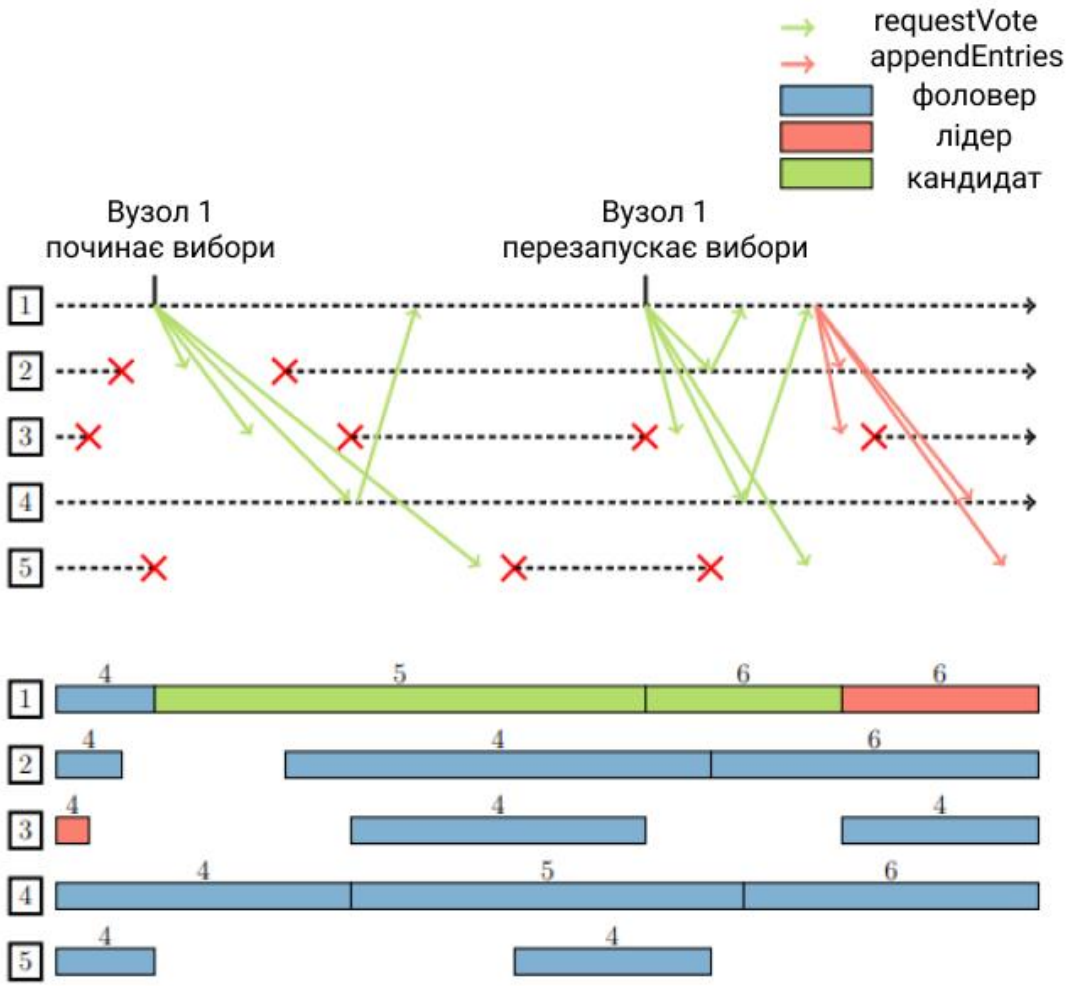


Рисунок 2.9 – Приклад тайм–ауту виборів через відмову багатьох вузлів.

На початку цього сценарію вузол 3 є лідером у терміні 4, і всі вузли знають про це, але вузли 2, 3 і 5 незабаром виходять з ладу. Вузол 1 є першим вузлом, який досягає тайм–ауту і починає вибори в термін 5. Вузол 1 не може отримати голоси від вузлів 2, 3 і 5, тому в кінцевому підсумку час очікування стік і вибори перезапустились у терміні 6. Тим часом вузол 2 відновився, а вузли 3 і 5 відновилися і знову вийшли з ладу. Потім вузол 1 отримує голоси від вузлів 2 і 4 і перемагає на виборах, щоб стати лідером у терміні 6.

2. Реплікація логів

Коли лідер обраний, на нього лягає відповідальність за управління розподіленим логом. Лідер приймає від клієнтів запити, що містять деякі команди. Лідер кладе в свій лог новий запис, що містить команду, а потім надсилає «AppendEntries» всім фоловерам, для того щоб вони реплікували запис на свій лог. На малюнку 2.10 показано приклад набору журналів, виду ключ–значення. Команди, які можуть виконуватися над журналами, це додати: наприклад. x ← 5, яка пов’язує значення 5 з ключем x, і знайти: наприклад, !y, що повертає значення, пов'язане з ключем y.

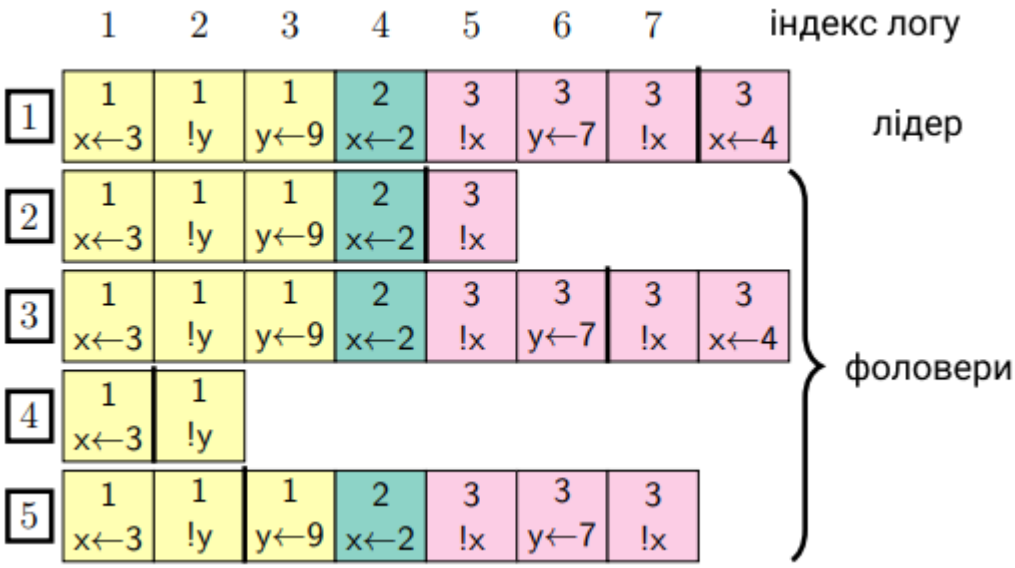


Рисунок 2.10 – Приклад станів журналу для кластеру з 5 вузлів. Вузол 1 є лідером, що зробив перші 7 записів у журналі, що були репліковані на більшості вузлів, у даному випадку на вузлах 3 і 5. Вузли 2 і 4 могли вийти з ладу або втратили свої повідомлення в мережі, тому їх логи відстають у актуальності. Лідер, вузол 1, відповідає за оновлення журналів цих вузлів.

Далі розглянуто випадок, коли жоден вузол не виходить з ладу і комунікація між серверами є надійною. Можна з упевненістю припустити, що всі вузли знаходяться в одному терміні і всі журнали ідентичні. Лідер відправляє запит AppendEntries, який включає (серед іншого) запис журналу, який лідер намагається

відтворити. Кожен вузол додає запис у свій журнал і успішно відповідає. Потім лідер застосовує команду до свого кінцевого автомата, оновлює свій індекс фіксації та повертає результат клієнту. У наступному запиту AppendEntries лідер інформує всі інші вузли про свій оновлений індекс фіксації, вузли застосовують команду до своїх кінцевих автоматів і оновлюють свій індекс фіксації. Цей процес повторюється багато разів.

Далі розглянуто випадок, коли деякі повідомлення були втрачені або вузли вийшли з ладу та відновилися, що залишило їхні журнали неузгодженими. Відповідальність лідера – виправити це, реплікуючи його журнал на всі інші вузли. Коли фоловер отримує AppendEntries, він містить індекс журналу та термін, пов’язаний з попереднім записом. Якщо ця інформація не збігається з останнім записом у журналі даного вузли, то він відповідає лідеру невдачею. Тепер лідер усвідомлює, що журнал непослідовний і його потрібно виправити. Лідер зменшить попередній індекс журналу та термін для цього вузла, додаючи записи до журналу, допоки невідповідний вузол не відповість успішно і не буде оновленим, як показано на малюнку 2.11. Кожен вузол зберігає свій журнал у постійному сховищі, яке включає історію всіх команд та пов’язані з ними терміни. Кожен вузол також має значення Commit Index, яке представляє собою останню команду, яка буде застосована до реплікованого кінцевого автомата.

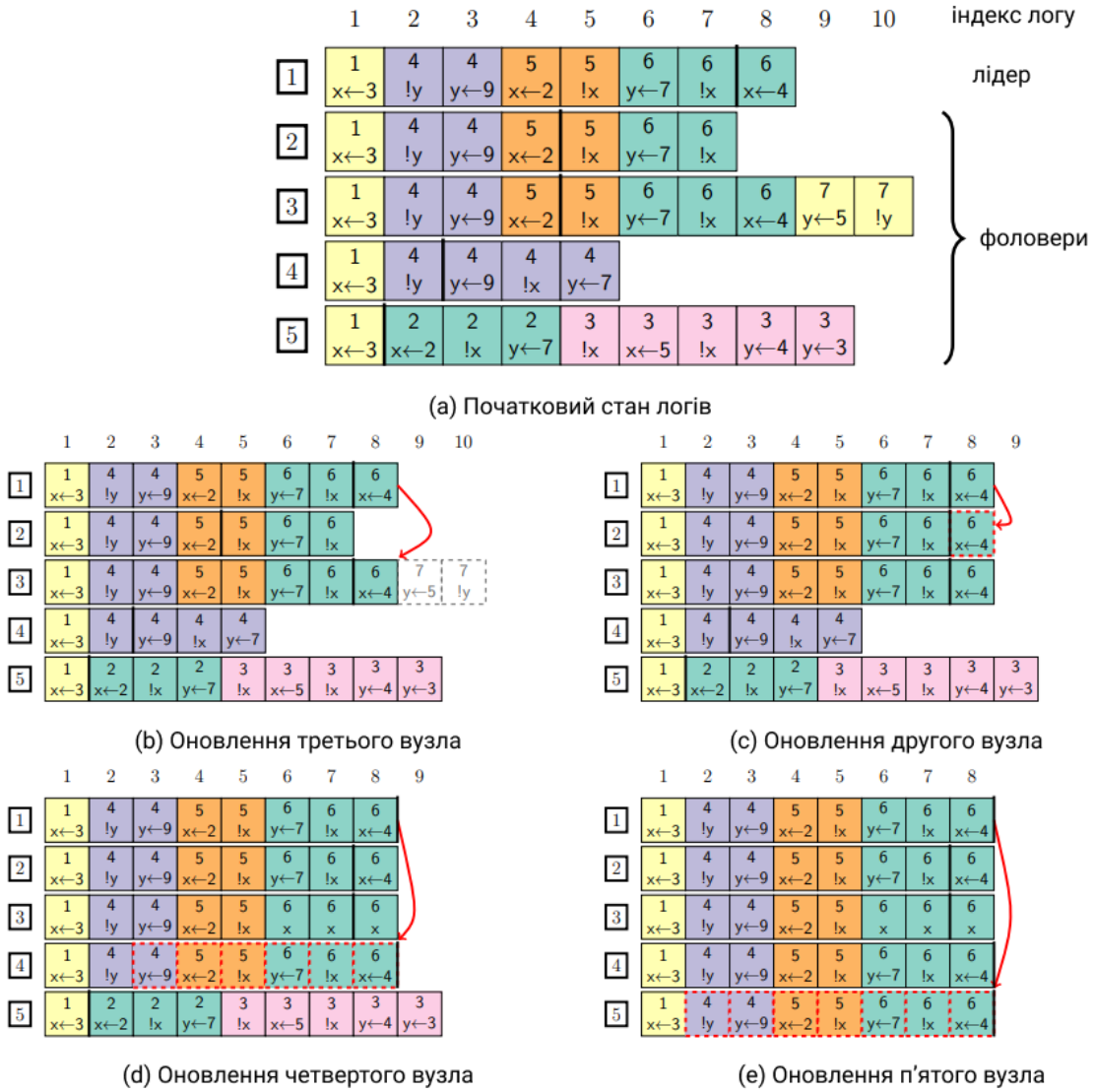


Рисунок 2.11 – Приклад реплікації логів:

(b) Вузол 1 щойно став лідером терміна 6, він відправляє AppendEntries з <PrevLogIndex, PrevLogTerm >=<8, 6> і лише вузол 3 є послідовним, тому він видаляє свої зайві записи;

(c) Вузол 1 повторює AppendEntries зі значенням <7, 6> до вузлів 2, 4 і 5, і лише вузол 2 є послідовним, тому додає у свій лог значення <8, 6>;

(d) Тепер вузол 1 може зафіксувати запис за індексом 8, а наступний AppendEntries містить значення <2, 4> і вузол 4 видаляє неправильні записи за індексами 3–5 і додає нові записи;

(e) Остаточний AppendEntries має значення <1, 1> і вузол 5, нарешті, узгоджений.

2.4.4 Гарантії надійності роботи алгоритму

Оскільки лідер дублює свій лог до логів всіх інших вузлів, цей лог повинен містити всі попередні записи. Для цього Raft накладає додаткові обмеження на протокол. Вузол проголосує за інший вузол лише в тому випадку, якщо його лог є принаймні таким самим оновленим, як лог даного вузла (визначається як той, що має останній запис із вищим терміном, або, якщо умови рівні, довший лог).

Протокол забезпечує лінеаризовану семантику, гарантуючи, що команда виконується між першим відправленням команди та першою успішною відповіддю. Протокол не гарантує певного чергування запитів клієнта, але гарантує, що всі автомати стануть бачити команди в одинаковому порядку. Протокол передбачає, що якщо клієнт не отримує відповідь на свій запит протягом часу очікування клієнта або відповідь негативна, він буде повторювати цей запит до успіху. Для забезпечення семантики, що піддається лінеаризації, ми повинні переконатися, що кожна команда застосовується до кожного автомата стану щонайменше один раз. Для вирішення цієї проблеми кожна команда клієнта має відповідний серійний номер. Кожен модуль консенсусу кешує останній серійний номер, оброблений від кожного клієнта, та надану відповідь. Якщо модулю консенсусу двічі дається одна і та ж команда, тоді вдруге він просто поверне кешовану відповідь. Протокол ніколи не дасть помилково позитивний результат, стверджуючи, що команда була скоєна тоді, коли вона насправді цього не зробила, але протокол може давати помилково негативні результати, стверджуючи, що команда не була скоєна, коли вона насправді є. Для вирішення цієї проблеми семантика клієнта вказує, що кожен клієнт повинен повторювати команду доти, доки її не буде успішно виконано. Кожен клієнт може мати щонайменше одну команду, що не виконується, і команди повинні надсилатися у порядку серійних номерів.

Алгоритм Raft надає системі набір властивостей, які разом гарантують надійність його виконання [35]:

- election safety (безпека виборів) – в рамках одного терміну може бути обрано не більше одного лідера. Ця властивість випливає з того, що кожен

сервер голосує в рамках кожного терміну лише одного разу, а для встановлення лідера необхідна більшість голосів;

– leader append–only (лідер лише прикріплює) – лідер ніколи не перезаписує і не стирає, не рухає записи в своєму лозі, тільки дописує в кінець логу нові записи. Ця властивість слідує безпосередньо з опису алгоритму – єдина операція, яку лідер може здійснювати зі своїм логом – добавляти записи в кінець;

– log matching (відповідність логів) – якщо логи двох серверів містять запис з однаковим індексом і номером терміну, то обидва логи ідентичні аж до цього запису включно. Дана властивість доводиться методом математичної індукції. Припустимо в логах уже є деякі записи. Алгоритм Raft надає користувачеві механізм, який дозволяє даній властивості виконуватись при будь-яких операціях з розподіленим логом – Consistency check (перевірка відповідності). Далі даний механізм розглянуто на декількох прикладах:

a. Нехай у системі є лідер 4-го терміну та фоловер. Їхні логи співпадають та складаються з трьох записів (рис 2.12а):

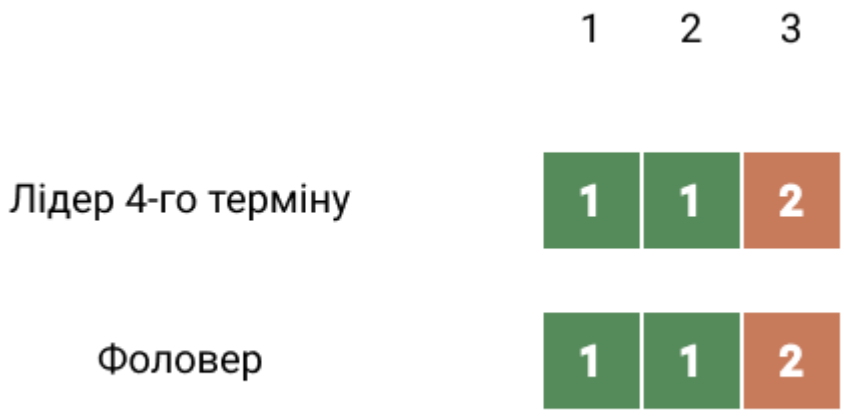


Рисунок 2.12а – Ілюстрація прикладу співпадаючих логів

До лідера приходить новий запис від клієнта. Він записує його в свій лог та відправляє запит AppendEntries фоловеру. Але, окрім самого запису, він

також добавляє до запиту інформацію про те, що запис потрібно добавити за індексом 4, а за індексом 3 повинен знаходитись запис із другого терміну (рис 2.12б):

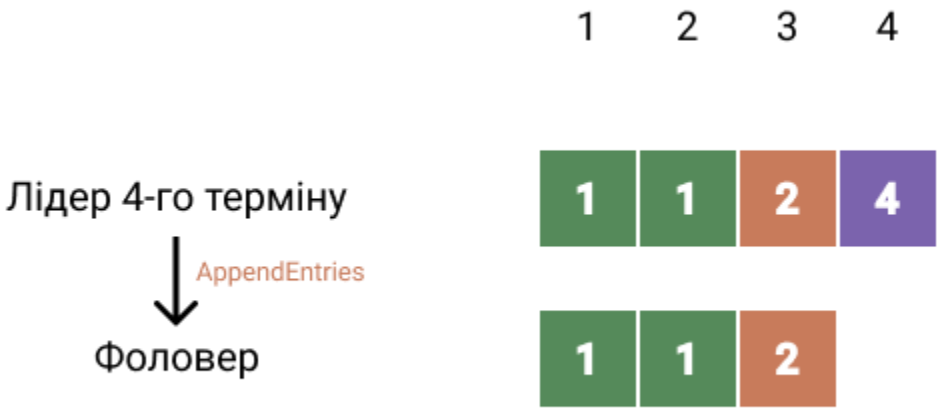


Рисунок 2.12б – Додавання нового запису у лог

Запис в логу фоловера співпадає з вказаним у запиті, тому фоловер добавляє запис з четвертого терміну у свій лог і відповідає лідеру про успішне завершення операції (рис 2.12в):

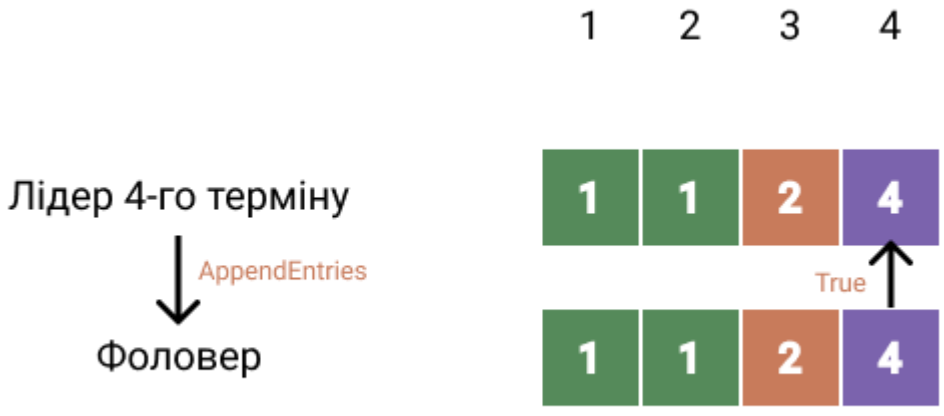


Рисунок 2.12в – Успішне завершення операції

b. Тепер розглянемо схожий з наведеним вище приклад, але з іншими початковими умовами. Нехай перед початком операції логи фоловера та лідера відрізняються між собою (рис 2.12г):

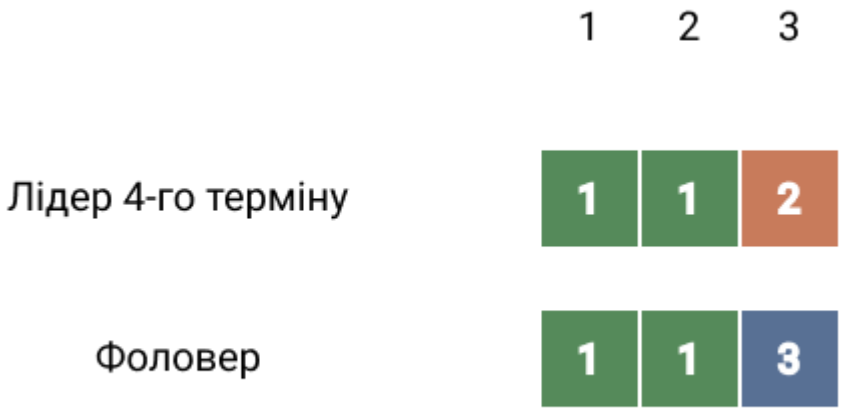


Рисунок 2.12г – Ілюстрація прикладу не співпадаючих логів

Коли лідер отримає нові дані для запису в лог, він відправить фоловеру такий самий запит на запис нових даних, але тепер фоловер відповість на запит невдачею, оскільки записи по третьому індексу у них відрізняються (рис 2.12д):

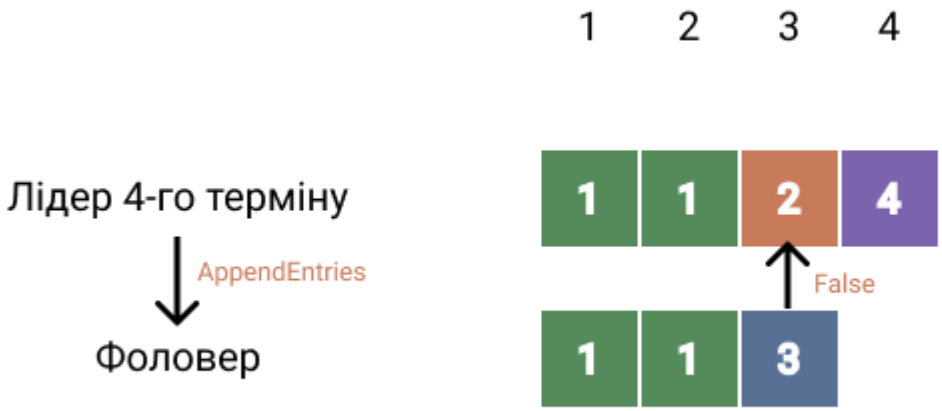


Рисунок 2.12д – Невдалий запис у лог фоловера

У такому випадку лідер “відкочується” назад у своїх записах і намагається записати у лог фоловера те значення, що знаходиться у його лозі за індексом 3, так само включаючи у запит попереднє значення (за індексом 2). Тепер фоловер відповідає успіхом і перезаписує свій лог починаючи з індекса 3 (рис 2.12е):

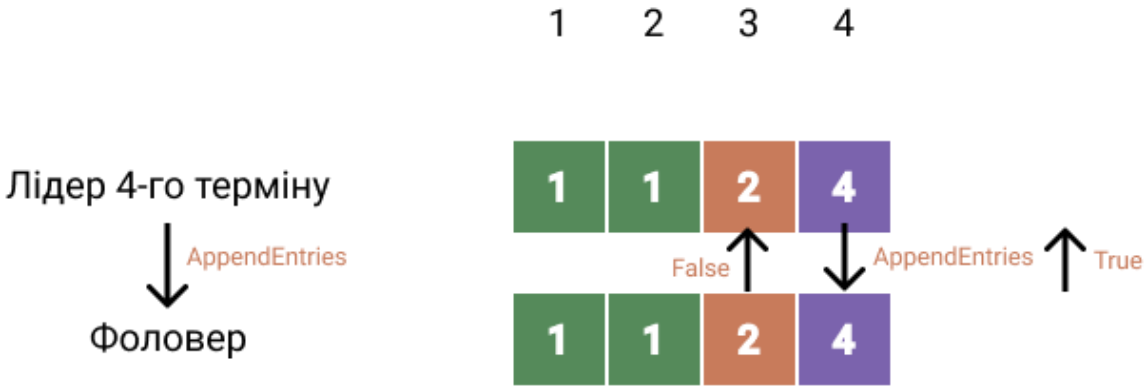


Рисунок 2.12е – Успішний перезапис логу фоловера

Лог фоловера може суттєво відрізнятися від логу лідера, у ньому можуть бути відсутні деякі записи, або існувати некоректні записи з попередніх голосувань. У будь-якому випадку механізм consistency check гарантує, що рано чи пізно, логи всіх фоловерів будуть співпадати із логом лідера.

– leader completeness (повноцінність лідера) – якщо запис в лозі закомічений в поточний термін, то логи лідерів усіх наступних термінів будуть включати цей запис.

Нехай у кластері працює три сервери–вузли, що мають по три однакові записи у своїх логах. Перший вузол S1 – лідер поточного першого терміну (рис 2.13).

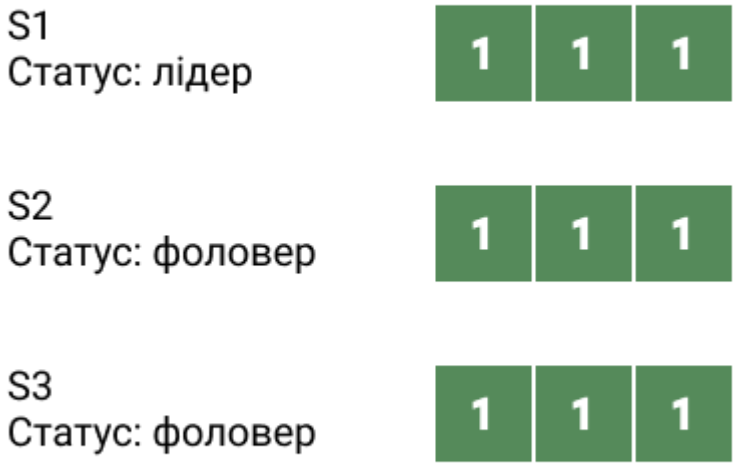


Рисунок 2.13 – Стан вузлів у кластері

Лідер S1 отримує від клієнта нові дані та записує їх у свій лог. Після цього він відправляє запит AppendEntries вузлам S2 та S3. Сервер S2 успішно отримує дані для запису, але зв'язок між серверами S1 та S3 нестабільний, тому дані втрачаються. Оскільки S1 знає, що запис уже є на двох вузлах із трьох, він може підтвердити, що запис закомічено і відповісти клієнту успіхом. S1 також буде відправляти S3 запис до тих пір, поки той не відповість про успішне збереження. У таких умовах, доки сервер S3 не обновить свій лог, він не зможе стати лідером, оскільки його лог є менш актуальним. Цей механізм називається Election restriction – вузол буде голосувати за інший вузол, тільки якщо його лог є не менш актуальним за лог даного вузла, порівнюючи два параметри: довжину логу та номер терміну останнього запису. Ці два параметри кандидати включають у запит RequestVote, щоб фоловери змогли порівняти актуальність записів у логах.

Головнішим при порівнянні вважається той лог, номер терміну останнього запису якого є вищим (рис 2.14а):

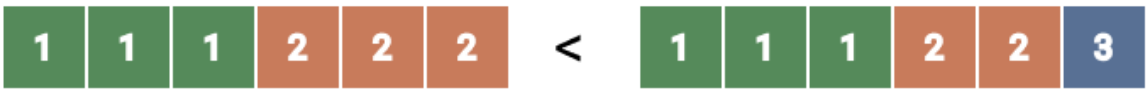


Рисунок 2.14а – Порівняння за номером терміну останнього запису

Якщо номери термінів останніх записів між двома логами співпадають, головним буде той лог, який є довшим (рис 2.14б):

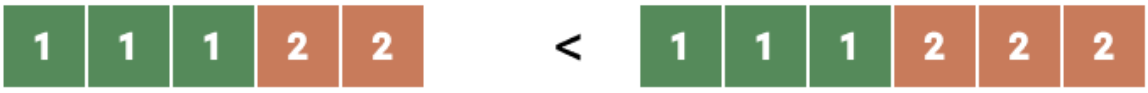


Рисунок 2.14б – Порівняння за довжиною логу

Якщо ж два параметри співпадають, то вважається, що логи ідентичні (рис 2.14в):

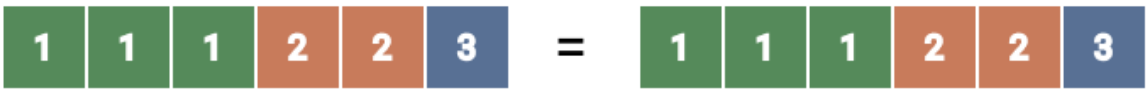


Рисунок 2.14в – Ідентичні логи

Таким чином, лог вузла, що містить закомічений запис, завжди буде актуальнішим за лог того вузла, що його не містить. Вузол, у якого є актуальний закомічений запис, не буде голосувати за той вузол, у якого такого запису немає. А оскільки закомічений запис є на більшості серверів, той сервер, що її

не має і стає кандидатом, не зможе стати лідером, тому що більшість за нього не проголосує.

– state machine safety – коли сервер записує запис з деяким індексом, жоден інший сервер не запише інший запис з використанням цього індексу.

2.5 Порівняльна характеристика алгоритмів Raft та Paxos

Алгоритми Paxos та Raft у загальному схожі між собою, але відрізняються використаним підходом для вибору лідера (Таблиця 2.1):

Таблиця 2.1 – Порівняння вибору лідера у Paxos та Raft

Умова	Paxos	Raft
У кожному терміні обраний лише один лідер	Сервер S може бути кандидатом у термін T лише у тому випадку, якщо T mod N = S. Тоді у рамках одного терміну буде лише один кандидат і, відповідно, один лідер.	Фоловер може стати кандидатом у будь-який термін. Кожен з фоловерів може проголосувати лише за одного кандидата у терміні, тому лідером стане лише той один кандидат, що матиме більшість голосів.
Лог лідера містить усі раніше закомічені записи	Кожна відповідь на RequestVote включає у себе всі записи логу фоловера. Коли кандидат отримує усі відповіді, він додає записи з найвищим номером терміну у свій лог.	Кандидат отримує голос від фоловера тільки в тому разі, якщо його лог є актуальним. Це гарантує те, що лідером може стати лише той кандидат, чий лог актуальний з логом більшості фоловерів.
Лідер безпечно комітить записи з попередніх термінів	Записи логу з попередніх термінів додаються до логу лідера з терміном лідера. Лідер реплікує ці записи як власні.	Лідер реплікує логи на інші вузли не змінюючи терміни записів. Лідер не може вважати свої записи закоміченими, поки він не реплікує порядок своїх записів на інші вузли.

Аби зробити вибір на користь одного із даних алгоритмів консенсусу, їх можна також порівняти за двома критеріями: зрозумілість та ефективність.

Зрозумілість. Raft гарантує, що якщо два логи містять один і той же запис, то він матиме однаковий індекс та термін в обох цих логах. Іншими словами, кожній операції запису присвоюється унікальний індекс і термін. Однак це не виконується у алгоритмі Paxos, де майбутній лідер може присвоїти операції вищий термін. У Paxos записи логу, з індексом меншим, ніж індекс поточного запису, можуть бути перезаписані. Це безпечно, оскільки запис логу буде перезаписаний у рамках тієї ж операції, але такий підхід не є достатньо інтуїтивним, як підхід Raft. В цілому вважається, що підхід алгоритму Raft є більш зрозумілим для реалізації.

Ефективність. У Paxos, якщо кандидатами одночасно стануть кілька вузлів, на виборах виграє кандидат з вищим терміном. У Raft, якщо декілька серверів одночасно стають кандидатами, голоси між ними можуть розділитися, а оскільки вони матимуть однаковий термін, жоден з них не виграє вибори. Raft пом'якшує це тим, що фоловери чекають деякий випадковий час перед тим як почати нові вибори. Таким чином, Raft є дещо повільнішим і матиме більшу дисперсію у часі, необхідному для обрання лідера. Однак сама фаза виборів лідера у Raft є легшою, ніж у Paxos. Raft дозволяє стати лідером лише тому кандидату, що має актуальний оновлений журнал і тому не потребує додаткових записів в лог під час виборів лідера. У Paxos кожна позитивна відповідь RequestVote включає в себе записи логу фоловера з індексом, більшим за індекс запису кандидата. Існують різні варіанти зменшення кількості надісланих записів логу, але в кінцевому підсумку завжди буде необхідно надсилати деякі записи логу, якщо лог лідера ще не оновлений. Додатково до цього в обох алгоритмах, як тільки кандидат стає лідером, він копіює свій лог на всі інші вузли. У Paxos лідер присвоює записам логу новий термін, і можлива така ситуація, що лідер надсилає копію запису логу на сервер, у якого вже є ця копія. Така ситуація неможлива у Raft, де кожен запис логу зберігає один і той самий термін протягом усього часу перебування у логу.

Враховуючи усе вищесказане, порівняно з Paxos, підхід Raft до обрання лідера є напрочуд ефективним, будучи значно простішим. Тому цей алгоритм доцільно обрати для подальшої роботи із задачею консенсусу у розподілених системах.

2.6 Способи оптимізації алгоритму Raft досягнення консенсусу

2.6.1 Введення динамічного протоколу голосування для алгоритму Raft

Основною причиною високої ресурсоємності алгоритму Raft є використання консенсусного голосування більшості для визначення кворуму *Q* запису і кворуму *Q* виборів [36-37]. Оскільки ці кворуми повинні містити більшість серверів кластера, кластер Raft повинен містити щонайменше $2n + 1$ сервери, щоб мати можливість сприймати n збоїв серверів. Набагато ефективнішим рішенням є зробити ці кворуми динамічними та вимагати, щоб вони містили більшість саме активних серверів даного кластера, таким чином виключаючи сервери, які аварійно завершили роботу та не були офіційно реінтегровані [38]. Отже, якщо кластер містить чотири сервери, його початкова більшість буде визначена як три сервери з чотирьох. Якщо один із серверів вийде з ладу, новою більшістю буде два з трьох. Після другого збою сервера це буде два з двох. Що станеться після третього збою, залежить від того, як протокол обробляє зв'язки. Оригінальний протокол динамічного голосування (DV) не призначає ваги серверам і не має ніяких правил обробки розриву зв'язку. Як результат, для отримання оновлень потрібно мінімум два сервери, що обмінюються даними. Просте розширення, відоме як динамічно-лінійне голосування (DLV) [39], вирішує цю проблему, застосовуючи правила впорядкування до серверів кластеру. У загальному випадку, і DV, і DLV вимагають $n + 2$ серверів на кластер, щоб мати змогу сприймати n збоїв. Реалізація цієї властивості у алгоритмі Raft, призводить до 25-відсоткового зменшення ресурсоємності Raft. Проте для покладання на динамічні кворуми, щоб вирішувати, коли проводити оновлення та обирати нових лідерів, доведеться підтримувати додаткові динамічні метадані. Найпростішим рішенням є використання когортних наборів і надання кластеру управління ними у верхній частині алгоритму Raft.

2.6.2 Когортні набори

Когортний набір представляє собою набір серверів, яким дозволено брати участь у виборах лідера. Цей набір може зберігатися у бітовій мапі. За оновлення цієї мапи відповідає лідер кластера. Він буде робити це кожен раз, коли виявить (а), що сервер перестав відповідати на запит на оновлення та (б) що сервер, який раніше був недоступний, відновив свою роботу. Оновлення мапи надсилаються на інші сервери так само, як і звичайні оновлення: сервер присвоює кожному оновленню порядковий номер у межах поточного терміну та пересилає його своїм фоловерам через виклик AppendEntries. Усі фоловери, які мають оновлені логи, додають нове оновлення до своїх логів та повідомляють про це лідера. Як тільки лідер реплікує оновлення на більшості серверів у поточному когортному наборі, він остаточно зберігає його (комітить), застосовує його до поточної системи і повідомляє своїх фоловерів.

Зазвичай лідер виявляє збої та відновлення фоловерів, відстежуючи, які фоловери підтверджують його запити AppendEntries. Коли запити на оновлення, що створюються користувачем, стають рідшими, цей підхід може призвести до неприпустимих затримок між часом, коли сервер виходить з ладу, і часом, коли лідер кластера виявляє цю помилку. Найпростішим рішенням цієї проблеми є дозвіл лідера генерувати фіктивний запит AppendEntries щоразу, коли минув фіксований інтервал без жодних запитів на оновлення, створених користувачами.

2.6.3 Обробка невдач лідера

Відповідно, за умови використання когортних наборів, лідер може враховувати лише ті голоси, які належить серверам, що входять до останнього встановленого когортного набору. Досягти цього можна у наступні кроки:

1) будь-який кандидат на лідерство буде включати власну версію когортного набору до повідомлення, яке він надсилає іншим серверам;
2) як і у виборчому протоколі Raft, сервери, які отримують запрошення на голосування, утримують свій голос, якщо виконується будь-яка з трьох наступних умов:

a) вони вірять, що у них все ще є лідер;

b) вони вже проголосували за іншого кандидата;

c) їх власний лог є більш актуальним, ніж лог кандидата;

3) сервери, які голосують за кандидата, додадуть до своїх голосів власну версію когортного набору;

4) коли кандидат підраховує отримані голоси, він порівнює їх версії когортного набору. Роблячи це, він визначає поточне значення цього набору та ігнорує голоси всіх серверів, що не входять до нього, при визначенні результатів виборів.

Варто зазначити, що не обов'язково всі сервери кластеру матимуть однакові збережені когортні набори. Далі розглянуто випадок, коли кластер складається з чотирьох серверів *A, B, C, D,* де сервер *A* є лідером. Коли усі чотири сервери коректно працюють, склад їхніх когортних наборів є ідентичним:

A	*B*	*C*	*D*
{A,B,C,D}	*{A,B,C,D}*	*{A,B,C,D}*	*{A,B,C,D}*

Після того, як сервер *D* завершить свою роботу аварійно або перестане відповідати на запити лідера, когортний набір серверів кластера оновиться:

A	*B*	*C*	*D*
{A,B,C}	*{A,B,C}*	*{A,B,C}*	*{A,B,C,D}*

Припускається, що сервер *D* відновив свою роботу і лідер *A* додав його до свого когортного набору, але припинив роботу перед тим, як розіслати дане оновлення усім вузлам кластера:

A	*B*	*C*	*D*
{A,B,C,D}	*{A,B,C}*	*{A,B,C}*	*{A,B,C,D}*

Оскільки сервер A не встиг оновити когортні набори вузлів, він залишиться у когортних наборах усіх працюючих вузлів до появи нового лідера.

2.6.4 Реалізації

Далі проведено оцінку трьох можливих реалізації алгоритму Raft з точки зору доступності кластера, ризику втрати даних та енергоємності. Ці три реалізації виглядають наступним чином:

1) кластер, що складається із чотирьох вузлів, який вимагає як мінімум два сервера, щоб підтримувати нові оновлення даних;
2) кластер, що складається із чотирьох вузлів, який вимагає наявності всього одного робочого вузла для підтримки працездатності;
3) конвенційний кластер Raft, що складається з 3 або з 5 вузлів.

Використано три показники ефективності кластера в оцінці даних реалізацій:

1) доступність кластера, тобто відрізок часу, коли він буде спроможний обробляти запити клієнта;
2) незахищеність кластера від непоправної відмови вузлів, коли кластер буде працювати лише з двома коректно працюючим вузлами;
3) незахищеність кластера від непоправної відмови вузлів, коли кластер буде працювати лише з одним працюючим вузлом.

Для того, аби вивести ці значення, необхідно зробити деякі припущення щодо роботи кластера. Нехай кожен вузол працює незалежно один від одного і так само незалежно може відмовити. Кожного разу, коли вузол падає, система автоматично запускає процес його відновлення. Ми припускаємо, що падіння серверів це незалежні випадкові події розподілені за експоненційним законом з середнім λ. За аналогією, час відновлення роботи вузла має експоненційний розподіл з середнім μ. Обидві гіпотези необхідні для того, аби представити систему у вигляді процесу Маркова [40] з кінцевою кількістю можливих станів. Припускається також, що лідери можуть швидко визначати відмову вузлів з допомогою періодичних фіктивних запитів.

2.6.4.1 Кластер з чотирьох вузлів, що забороняє роботу системи з одним працюючим вузлом

Далі розглянуто перший варіант реалізації – кластер з чотирьох вузлів, який потребує принаймні двох коректно працюючих вузлів, щоб підтримувати оновлення даних і захистити сервер від непоправної відмови одного з них. Поведінку такої системи можна описати з допомогою діаграми зміни станів вузлів (рис. 2.15).

Кластер має всього 9 можливих станів. Стан 4 – це початковий стан системи, коли усі вузли працюють коректно і більшість (кворум) складається з трьох вузлів із чотирьох можливих. Відмова будь-якого із вузлів переводить систему у стан 3 і кворум складатиметься з двох вузлів із трьох можливих. Падіння ще одного вузла переведе систему у стан 2, кворум складатиметься з двох вузлів із двох можливих.

Відмова третього вузла переведе систему у стан 1’, в такому разі система буде непрацездатною, адже неможливо буде визначити кворум серверів. Розглянемо три відмінні переходи системи зі стану 1’:

1) падіння останнього вузла призведе до переходу системи у стан 0’’;
2) відновлення одного із вузлів, що має актуальний лог, поверне систему у стан 2 і знову дозволить системі обробляти запити користувача;
3) відновлення будь-якого з двох інших вузлів переведе систему у стан 2’, у якому вона буде не здатна обробляти запити користувача.

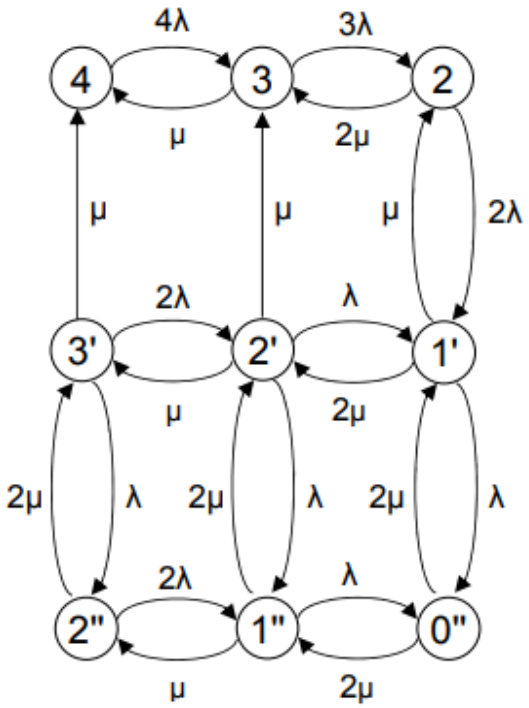


Рисунок 2.15 – Діаграма зміни станів системи з конфігурацією (1)

Стани системи, помічені одним штрихом, це такі стани, при яких система є недоступно, навіть якщо вона включає один сервер, що має поточну версію логу. Існує два способи, якими система може вийти зі стану 0’’:

1) відновлення одного із серверів з поточною версією логу переведе систему у стан 1’;

2) відновлення одного із вузлів, що має застарілий лог, переведе систему у стан 1’’.

Усі стани, що помічені двома штрихами, також відображають недоступність системи. Стани з одним та’ двома штрихами відрізняються лише композицією наборів працюючих вузлів. Усі переходи від станів з двома штрихами до станів з одним штрихом означають відновлення одного з двох вузлів з поточною версією логу. Аналогічно усі переходи системі зі станів з одним штрихом до станів без штрихів означають відновлення іншого сервера з поточною версією логу і дозволяють системі знову обробляти запити клієнта. Стану 3’’ не існує, адже будь-яка група з трьох

серверів, що відновлюються, буде обов'язково включати в себе один сервер з поточною версією логу.

Умови рівноваги для такого кластеру є наступними:

$$4\lambda p_4 = \mu(p_3 + p_{3'})$$
$$(3\lambda + \mu)p_3 = 4\lambda p_4 + 2\mu p_2 + \mu p_{2'}$$
$$(3\lambda + \mu)p_{3'} = \mu p_{2'} + 2\mu p_{2''}$$
$$(2\lambda + 2\mu)p_2 = 3\lambda p_3 + \mu p_{1'}$$
$$(2\lambda + 2\mu)p_{2'} = 2\lambda p_{3'} + 2\mu(p_{1'} + p_{1''})$$
$$(2\lambda + 2\mu)p_{2''} = \lambda p_{3'} + \mu p_{1''}$$
$$(\lambda + 3\mu)p_{1'} = 2\lambda p_2 + \lambda p_{2'} + 2\mu p_{0''}$$
$$(\lambda + 3\mu)p_{1''} = \lambda p_{2'} + 2\lambda p_{2''} + 2\mu p_{0''}$$
$$4\mu p_{0''} = \lambda(p_{1'} + p_{1''}),$$

де p_i це ймовірність перебування кластера у стані i з додатковою умовою, що $\sum_i \ p_i = 1$.

Розв’яжемо систему лінійних рівнянь і замінимо відношення $\frac{\lambda}{\mu} = \rho$. Отримано наступні результати:

1) доступність кластера $A_{4R}(\rho)$:

$$A_{4R}(\rho) = p_4 + p_3 + p_2 = \frac{3\rho^6+23\rho^5+68\rho^4+97\rho^3+21\rho+3}{(\rho+1)^5(3\rho^3+8\rho^2+6\rho+3)}$$

2) незахищеність кластера від роботи на двох вузлах $E_{4R,2}(\rho)$:

$$E_{4,2}(\rho) = p_2 = \frac{3\rho^6 + 17\rho^5 + 39\rho^4 + 42\rho^3 + 18\rho^2}{(\rho + 1)^5(3\rho^3 + 8\rho^2 + 6\rho + 3)}$$

2.6.4.2 Кластер з чотирьох вузлів, що дозволяє роботу системи з одним працюючим вузлом

Далі розглянуто як буде поводити себе система, якщо не буде виконуватись умова, що необхідно принаймні два вузли для того, щоб кластер обробляв запити клієнта, і достатньо буде лише одного працюючого вузла.

На рисунку 2.16 зображено схему переходів між станами системи у даній конфігурації, система може мати всього 8 станів. У стані 2 система буде мати два

робочі вузли і кворум буде складатися з цих двох вузлів. При падінні ще одного вузла із цього стану можливі два наступні переходи:

1) якщо вузол, який відключився від кластера, був фоловером, то система переходить у стан 1 і той вузол, що залишився, коректно обробляє запити клієнта.

2) якщо вузол, який відключився від кластера, був лідером, система переходить у стан 1’ і не обробляє нові запити.

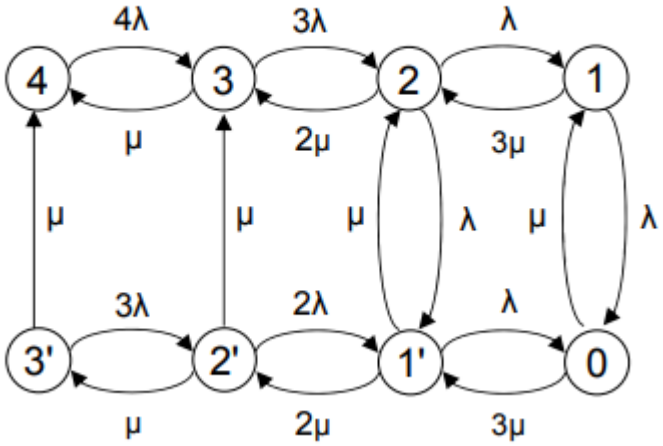


Рисунок 2.16 – Діаграма зміни станів системи з конфігурацією (2)

Стани 1 та 1’ мають однакові переходи у стан 0’, але переходи після відновлення серверів, які були відключені, для них відрізняються. Коли кластер перебуває у стані 1, відновлення будь-якого з вузлів переведе його у стан 2, тоді як зі стану 1’ можливі наступні переходи:

1) перехід у стан 2 відбувається, якщо сервер, який відновив свою роботу, має оновлений лог.

2) перехід у стан 2’ відбувається, якщо відновлений сервер має лог застарілої версії.

Умови рівноваги для такого кластеру є наступними:

$$4\lambda p_4 = \mu(p_3 + p_{3'})$$

$$(3\lambda + \mu)p_3 = 4\lambda p_4 + 2\mu p_2 + \mu p_{2'}$$

$$(3\lambda + \mu)p_{3'} = \mu p_{2'}$$

$$(2\lambda + 2\mu)p_2 = 3\lambda p_3 + 3\mu p_1 + \mu p_{1'}$$
$$(2\lambda + 2\mu)p_{2'} = 3\lambda p_{3'} + 2\mu p_{1'}$$
$$(\lambda + 3\mu)p_{1'} = \lambda p_2 + \mu p_{0'}$$
$$(\lambda + 3\mu)p_1 = \lambda p_2 + 2\lambda p_{2'} + 3\mu p_{0'}$$
$$4\mu p_{0'} = \lambda(p_1 + p_{1'}),$$

де p_i це ймовірність перебування кластера у стані i з додатковою умовою, що $\sum_i \ p_i = 1$.

Розв'яжемо систему лінійних рівнянь і замінимо відношення $\frac{\lambda}{\mu} = \rho$. Отримано наступні результати:

1) доступність кластера $A_{4R}(\rho)$:

$$A_{4R}(\rho) = p_4 + p_3 + p_2 + p_1 = \frac{6\rho^6 + 35\rho^5 + \ 102\rho^4 + 152\rho^3 + 113\rho^2 + 39\rho + 6}{(\rho + 1)^4(\rho^3 + 17\rho^2 + 15\rho + 6)}$$

2) незахищеність кластера від роботи на двох вузлах $E_{4,2}(\rho)$:

$$E_{4,2}(\rho) = p_2 = \frac{18\rho^4 + 42\rho^3 + 36\rho^2}{(\rho + 1)^3(6\rho^3 + 17\rho^2 + 15\rho + 6)}$$

3) незахищеність кластера від роботи з одним вузлом $E_{4,1}(\rho)$:

$$E_{4,1}(\rho) = p_1 = \frac{6\rho^6 + 17\rho^5 + \ 24\rho^4 + 12\rho^3}{(\rho + 1)^4(6\rho^3 + 17\rho^2 + 15\rho + 6)}$$

2.6.4.2 Конвенційний Raft кластер

Такі ж самі формули можна вивести і для систем, у яких працює класичний алгоритм консенсусу, що складаються з трьох або з п'яти вузлів. Такі системи вимагають більшості від початкової кількості вузлів для коректної роботи кластеру. Виходячи з цього, доступність такого кластера вираховується за формулою:

$$A = \frac{1}{\rho + 1}$$

Звідси, для кластера з трьома вузлами:

$$A_3(\rho) = A^3 + 3A^2(1 - A) = \frac{3\rho + 1}{(\rho + 1)^3}$$

$$E_{3,2}(\rho) = 3A^2(1-A) = \frac{3\rho}{(\rho+1)^3}$$

Для кластера, у якому п’ять вузлів:

$$A_5(\rho) = A^5 + 5A^4(1-A) + 10A^3(1-A)^2 = \frac{10\rho^2 + 5\rho + 1}{(\rho+1)^5}$$

$$E_{5,2}(\rho) = 0$$

2.6.5 Порівняння реалізацій алгоритму прийняття консенсусу Raft

На рисунку 2.17 зображено графік доступності кластера при чотирьох конфігураціях: динамічне голосування, що не дозволяє роботу системи з одним вузлом, динамічне голосування, при якому робота одного вузла можлива, класична конфігурація алгоритму Raft з трьома та п'ятьма вузлами. Доступність кластера тут залежить від відношення частоти відмов вузлів до частоти їх відновлення – ρ, що належить проміжку від 0 до 0,20. Нульова відмітка означає те, що кластер ніколи не відмовить, а відмітка 0,20 – те, що кластер буде відповідати на запити клієнта 83,3 відсотки часу.

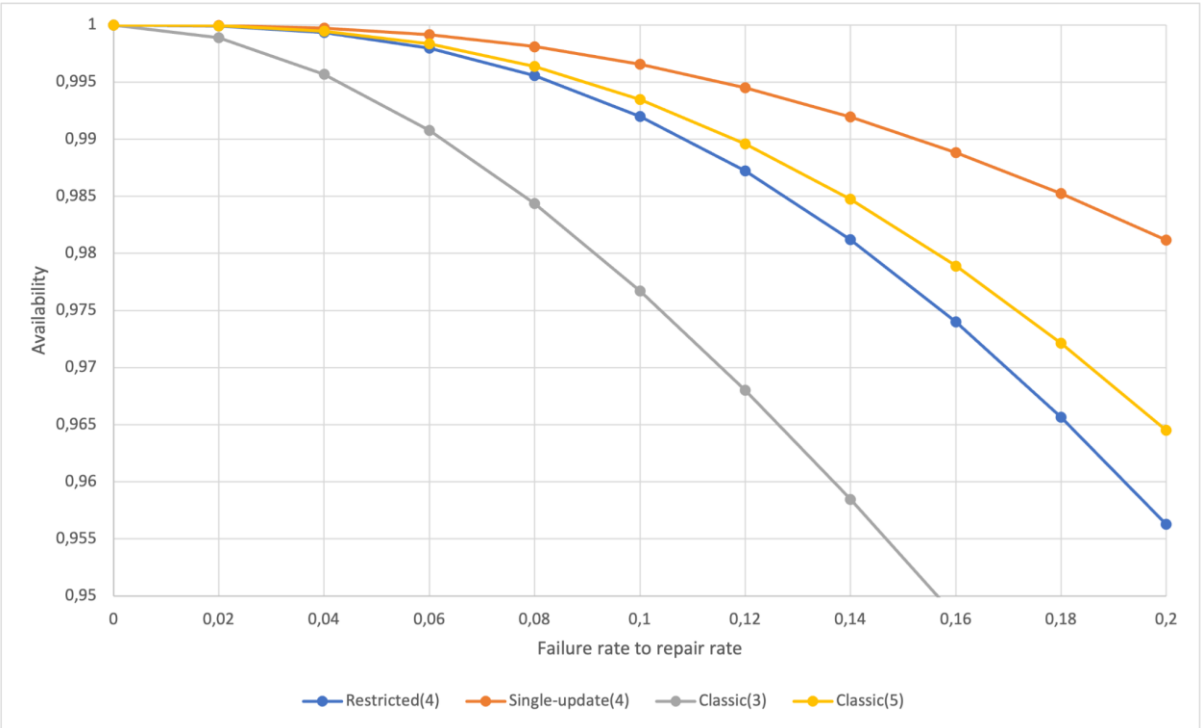


Рисунок 2.17 – Порівняння показника доступності кластера

Як можна бачити, динамічний протокол голосування із дозволом роботи одного вузла дає найкращі результати по доступності кластера, значно підвищуючи її у порівнянні з роботою конвенційних кластерів. Тим не менш, динамічне голосування першого типу показує дещо гірші показники доступності.

З огляду на типову частоту відмов серверів, що складає не більше однієї відмови на місяць, і типову тривалість відновлення сервера, що складає приблизно 24 години, відношення $\frac{\lambda}{\mu}$ стандартного кластера рідко буде перевищувати 0,033. Виходячи з цього, доступність кластерів з чотирьох вузлів з динамічною конфігурацією практично не відрізняється від доступності класичного кластера, що складається з 5 вузлів.

Побудовано графік, що порівнює показники незахищеності кластера від ситуації, коли у системі лишається лише два працюючі вузли, які відповідають за обробку запитів клієнта та оновлення логу. На рисунку 2.18 видно, що конфігурації динамічного голосування дають кращі показники, ніж конвенційний кластер Raft з трьох серверів. Конвенційний кластер, що складається з 5 вузлів, стійкий до даної ситуації, адже за алгоритмом потребує більшості для коректної роботи, а більшість для нього складає мінімум 3 вузли.

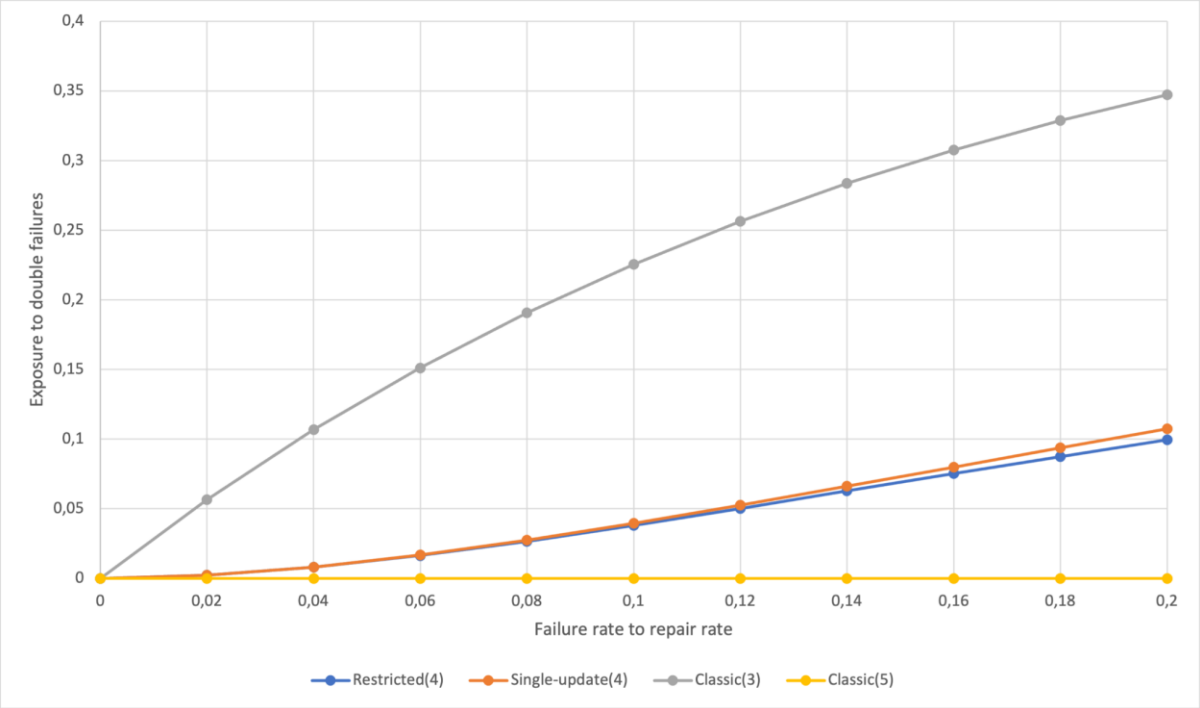


Рисунок 2.18 – Порівняння показників незахищеності кластера від роботи на двох вузлах

Що стосується оновлення логів на одному вузлі (рисунок 2.19), усі реалізації, окрім конфігурації динамічного голосування з дозволом роботи на одному вузлі, стійкі до даної ситуації, оскільки вона неможлива в умовах роботи даних алгоритмів. Для конфігурації динамічного голосування незахищеність значно зростає, якщо зростає відношення відмов вузлів до їх відновлення.

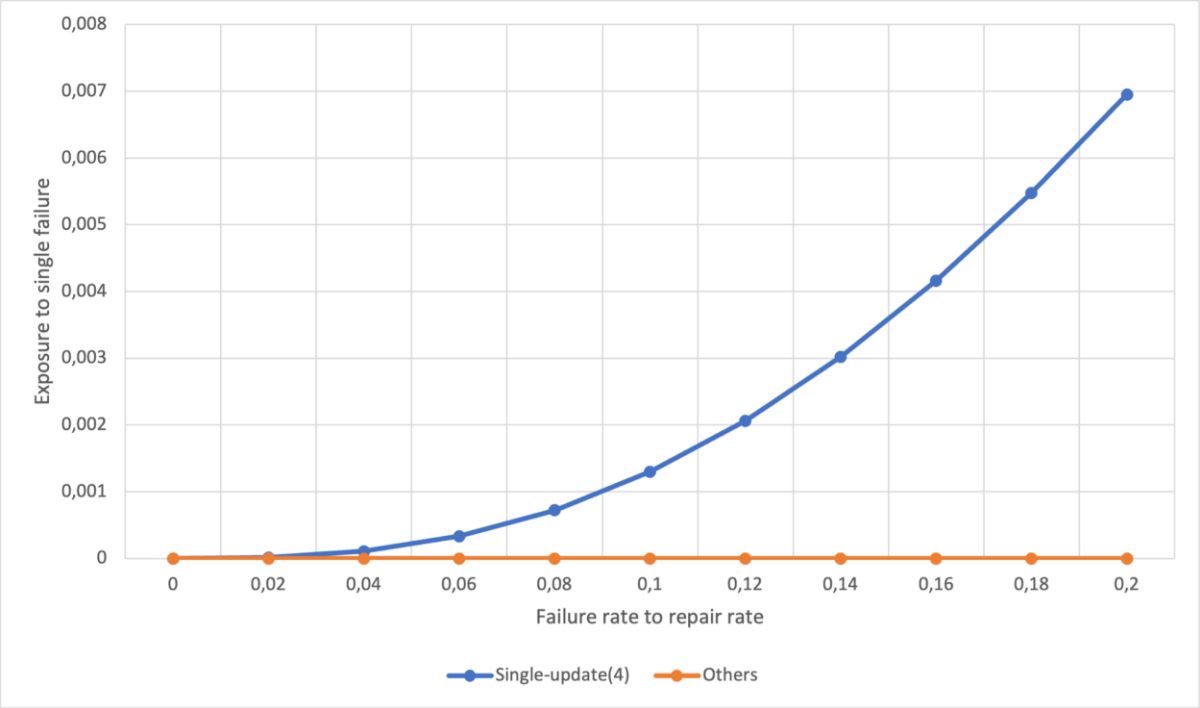


Рисунок 2.19 – Порівняння показників незахищеності кластера від роботи на одному вузлі

2.6.6 Додаткова оптимізація виборів лідера

Для належного рівня доступності системи необхідний відповідний діапазон тайм-ауту фоловера. Якщо таймер фоловера занадто малий, це може призвести до непотрібних виборів лідера, якщо ж він занадто великий – фоловери будуть повільно помічати відмову лідера. Таймер кандидата повинен бути достатньо великим, щоб отримати всі відповіді на RequestVotes, і при цьому достатньо малим, щоб не відкладати перезапуск виборів. Таймери фоловерів та кандидатів повинні мати достатньо широкий діапазон, щоб уникнути розділених голосів. Таймер лідера повинен бути меншим за час очікування фоловера. Точний розмір – це компроміс між використанням смуги пропускання та швидкістю, з якою інформація поширюється по кластеру. Висновок, зроблений авторами протоколу, полягає в тому, що T = 150 мс є хорошим консервативним тайм-аутом загалом. Перша умова відома як вимога часу, де BroadTime – це середній час, коли вузол надсилає RPC паралельно кожному серверу кластера та отримує відповіді на них.

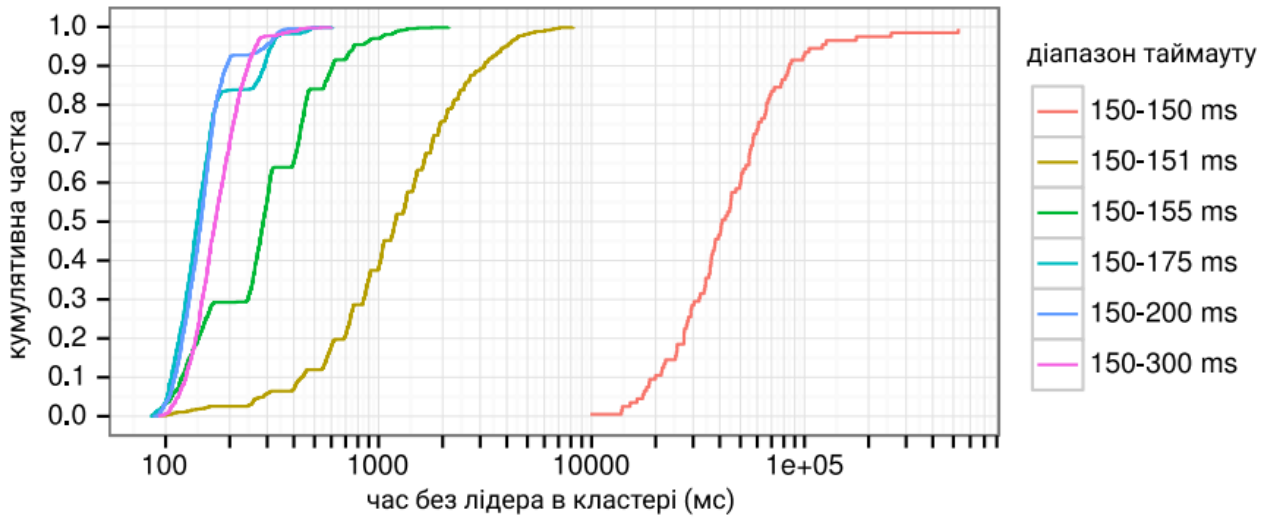
Гіпотеза оптимізації полягає в тому, що час, необхідний для обрання лідера в середовищі кластерів, можна значно покращити, не просто встановивши таймер кандидата на діапазон таймера фоловера. Оскільки автори використовують однаковий діапазон таймерів для кандидатів та фоловерів, ми очікуємо мінімум 150 мс (і до двох разів більше) перед тим, як розпочати вибори, незважаючи на те, що в середньому вузол отримує всі свої відповіді протягом 15мс. Це означає, що мінімальний час очікування можна змінити, надавши йому деяке оптимізоване розраховане значення. Це допоможе покращити час, необхідний для обрання лідера, і зробити протокол більш стійким до ворожих мережевих середовищ.

2.6.7 Експериментальні дослідження

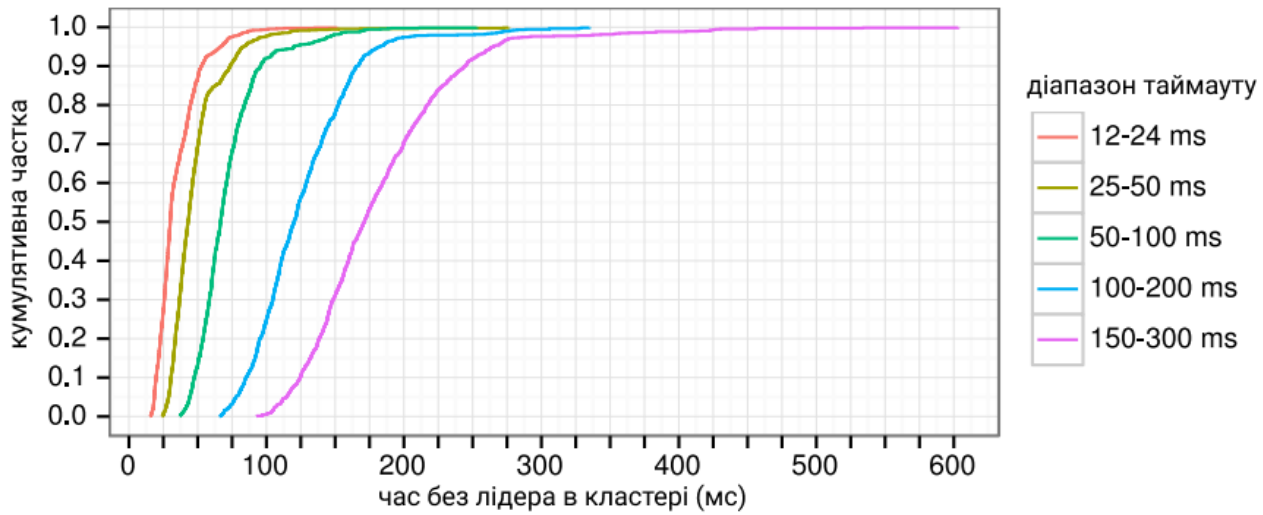
2.6.7.1 Порівняння часу проведення виборів нового лідера

Для того, аби експериментально дослідити та порівняти ефективність алгоритму консенсусу на реальних кластерах, а не на спрощених моделях, необхідно побудувати LAN мережу, що використовує протоколи для передачі даних між вузлами та у якій існує ймовірність затримки пакетів даних. Для цього необхідно мати доступ до фізичних ресурсів, тому розглянемо відповідне дослідження [41], у якому автори відтворили та протестували роботу алгоритму Raft на кластері з п'яти вузлів, що спілкуються між собою через TCP/IP протокол, що у свою чергу використовує гігабітний Ethernet для передачі пакетів.

Тестова система була налаштована таким чином, що лідер кластера періодично та неодноразово припиняв свою роботу, а система визначала час, протягом якого було виявлено даний збій і обрано нового лідера. Тест виміряв час від моменту збою старого лідера до того, як інші сервери отримають перший heartbeat запит від нового лідера (рисунок 2.20). Лідер зазнав аварії випадковим чином протягом інтервалу heartbeat, який становив половину мінімального тайм-ауту виборів для всіх тестів. Таким чином, найменший можливий час простою становив приблизно половину мінімального таймауту виборів.



(а) Час обирання нового лідера при зміні діапазону випадковості в тайм-ауті виборів



(б) Час обирання нового лідера при масштабуванні мінімального тайм-ауту виборів

Рисунок 2.20 – Графік залежності часу виявлення та заміни лідера в кластері від діапазону таймауту виборів. Кожна лінія представляє 1000 випробувань і відповідає певному тайм-ауту виборів; наприклад, «150-155 мс» означає, що тайм-аути виборів обирались випадковим чином і рівномірно розподілені між 150 мс і 155 мс. Сходинки, які відображаються на графіках, показують, коли відбувається розділення голосів (кластер повинен зачекати ще одного тайм-ауту виборів, перш ніж можна буде обрати лідера). Вимірювання проводилися на кластері з п’яти серверів із часом трансляції приблизно 15 мс.

З допомогою тестової системи автори намагалися створити найгірший сценарій виборів лідера. По-перше, вони синхронізували heartbeat запити старого лідера, перш ніж змусити його вийти з ладу; це призвело до того, що таймери виборів у фоловерів запускалися приблизно в один і той же час, що у свою чергу призвело до виникнення багатьох розділених голосів, якщо значення тайм-ауту були недостатньо випадковими. По-друге, сервери в кожному випробуванні мали різну довжину логу, тому два з чотирьох серверів не мали права стати лідерами.

Рисунок 2.20 (а) показує, що вибори завершуються менш ніж за одну секунду, коли діапазон часу очікування досить широкий. Невеликої рандомізації тайм-ауту виборів достатньо, щоб уникнути розділення голосів на виборах. За відсутності рандомізації, вибори лідера постійно тривали більше 10 секунд через велику кількість розділених голосів фоловерів (коли різні фоловери голосують за різних лідерів, в результаті чого лідер у терміні не обирається, адже жоден з них не набирає більшості голосів). Додавання лише 5 мс у діапазон рандомізації значно допомагає, що призводить до середнього часу простою в 287 мс. Використання більшого діапазону покращує поведінку в найгіршому випадку: із випадковим діапазоном 50 мс час завершення виборів у найгіршому випадку (понад 1000 спроб) становив 513 мс.

На рисунку 2.20 (б) показано, що час простою можна зменшити, зменшивши таум-аут виборів. При тайм-ауті виборів 12-24 мс для обрання лідера в середньому потрібно лише 35 мс (найдовша спроба зайняла 152 мс). Однак зменшення тайм-аутів нижче цієї позначки порушує вимогу алгоритму Raft щодо часу: лідери мають труднощі з надсиланням heartbeat запитів фоловерам до того, як інші сервери почнуть нові вибори. Це може призвести до непотрібних змін лідера та зниження загальної доступності системи. Автори рекомендують використовувати консервативний тайм-аут виборів, наприклад 150-300 мс; такі тайм-аути навряд чи призведуть до непотрібних змін лідера, забезпечать низький рівень розподілу голосів і продовжать надавати системі хорошу доступність.

2.6.7.2 Тестування продуктивності системи

До цього ми тестували реплікацію логів на прикладі одного клієнта, який відправляє дані до кластеру. Реальні системи працюють за принципом багатопоточності – коли безліч клієнтів одночасно надсилають запити на запис нових даних. Перевіримо як поведе себе наша система, при наявності одного потоку, десяти та ста одночасних клієнтських потоків. Кожен поток відправляє запит лідеру на запис значення, вагою в 1024 байти, отримує відповідь, і відразу ж відправляє новий запит.

На рисунку 2.21 показано поточну пропускну здатність кластера. Використовуючи багатопотоковий підхід із 100 потоками, кластер із трьома серверами підтримує близько 19 500 кілобайт записів в секунду. Очевидно, що продуктивність погіршується при використанні кластерів більших розмірів, оскільки лідер повинен надсилати кожен запис більшій кількості фоловерів.

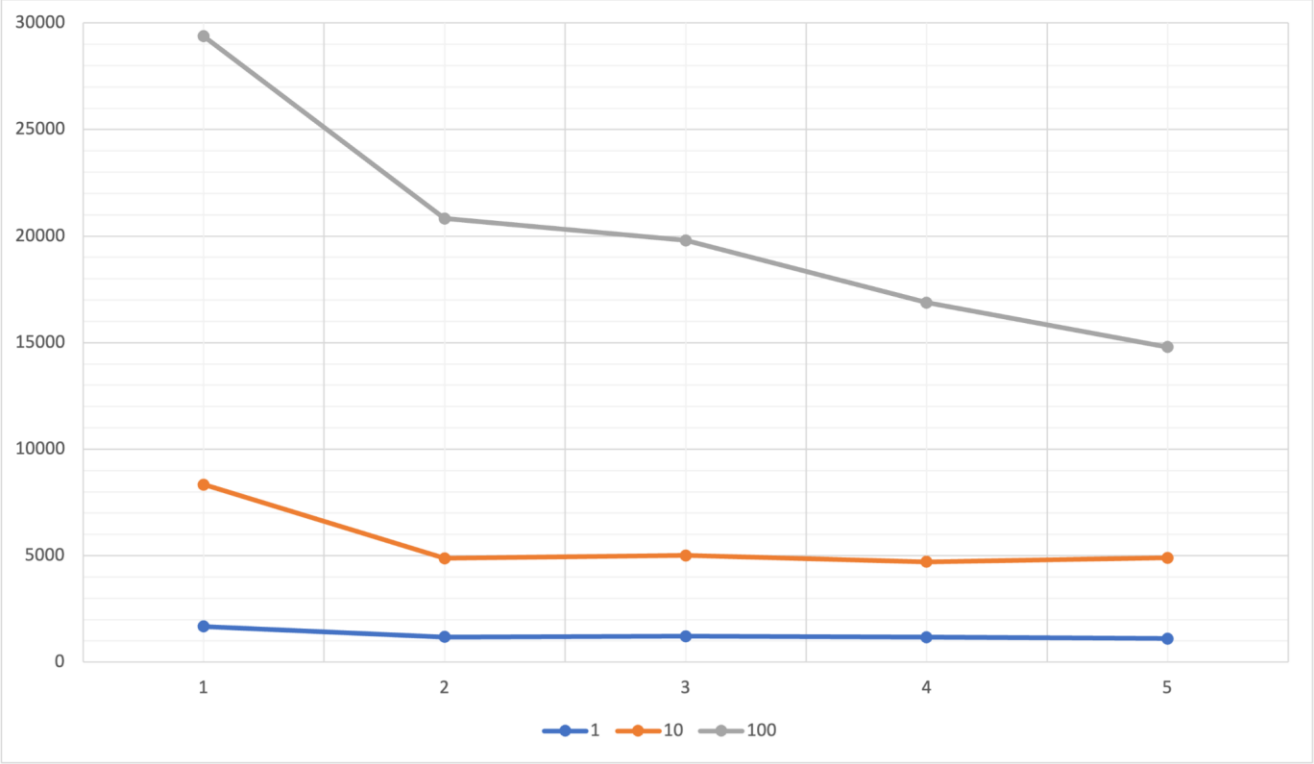


Рисунок 2.21 – Графік пропускної здатності кластера в залежності від кількості клієнтських потоків

На рисунку 2.22 показано поточну затримку кластера. Затримка для запису розміром у кілобайт становить близько 0,7 мс для кластера з одним сервером і близько

1,0 мс для кластерів з двома-п’ятьма серверами. Оскільки у віртуальному кластері час запису на диск є практично нескінченно швидким, для даного графіку враховано час для тривалого запису одного кілобайта на реальний диск, який складає приблизно 0,25 мс.

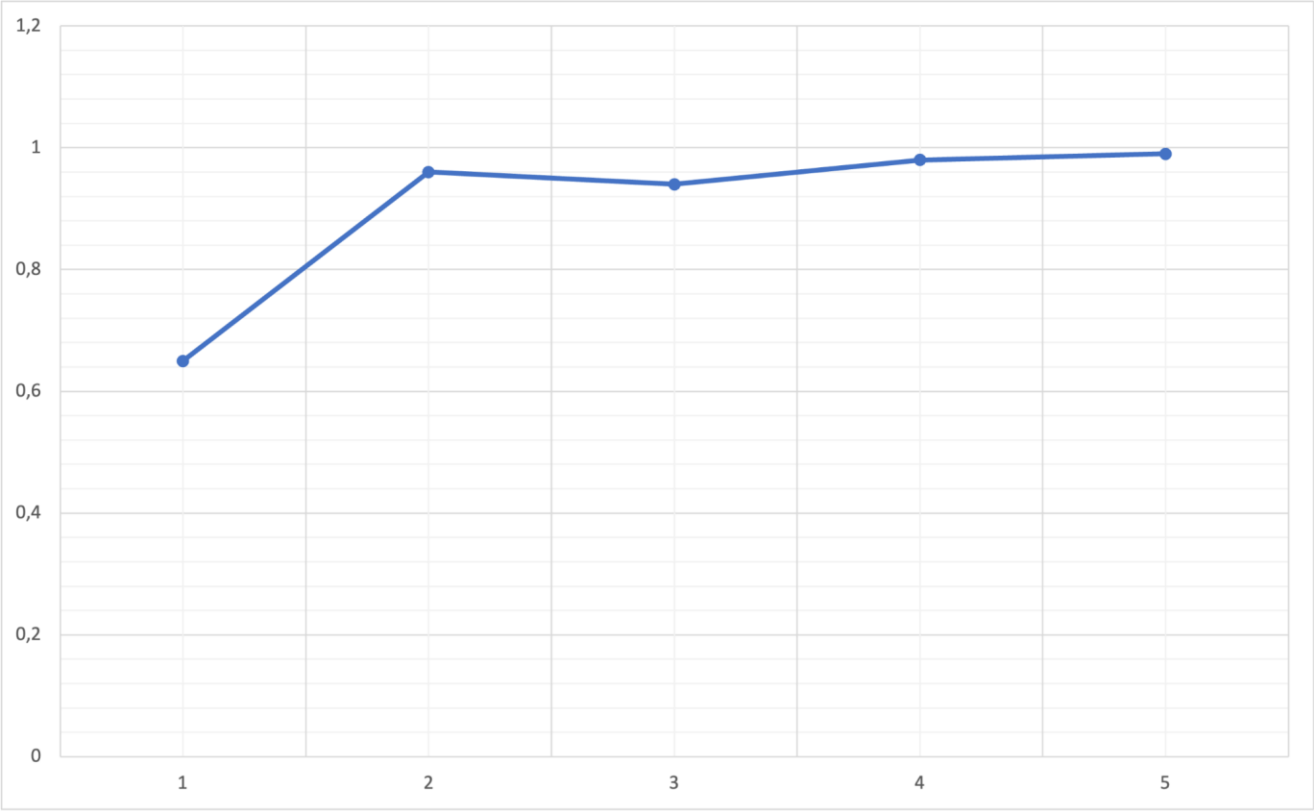


Рисунок 2.22 – Графік затримки запису даних на диск

Отримані вимірювання є обнадійливими, і я вважаю, що поточної продуктивності буде достатньо для широкого класу додатків. Проте є ще багато можливостей для вдосконалення.

2.6.8 Перспективи подальшого розвитку

Алгоритми досягнення консенсусу мають безліч перспектив подальшого розвитку. Деякі з них наведено нижче:

– розділення тайм-аутів фоловерів і кандидатів. Це може призвести до значних покращень показників продуктивності і дозволить налаштовувати їх більш відповідним чином, враховуючи їх цілі;

– підтвердження лідера клієнтом. У кластері завжди існує деяка ймовірність падіння вузлів, якщо клієнт не може визначити який вузол є лідером, користувач отримує повідомлення про несправність системи, хоча у наступну секунду лідер може бути обраним. Фіксація нового лідера клієнтом і додаткові налаштування його таймера допоможуть уникнути такої ситуації;

– різноманітні топології мережі. Алгоритм прийняття консенсусу Raft може зазнати серйозних труднощів з доступністю кластера в деяких мережевих топологіях. Необхідно дослідити роботу алгоритму в залежності від налаштувань мережі і внести додаткові зміни;

– одночасні запити клієнта. Використання серійних номерів для термінів та логів операцій обмежує кожного клієнта у тому, що він не може виконувати запити послідовно. Інакше може статися ситуація, коли порядок запису команд не відповідатиме дійсності;

– команди читання. Команди читання не потрібно реплікувати на всіх вузлах. Досить виконати їх лише для вузла-лідера, припускаючи, що лідер зафіксував запис зі свого терміну й нещодавно відправив успішні AppendEntries до більшості вузлів;

– нестабільність клієнта. Щоб допускати збій вузла-клієнта та його подальше відновлення, клієнти повинні зберігати свій поточний серійний номер і не виконану команду в енергонезалежній пам’яті або отримати новий ідентифікатор клієнта. Усі клієнти повинні мати різні ідентифікатори.

– стійкість до бризантійських відмов. Одним із найбільш перспективних способів розвитку алгоритмів досягнення консенсусу у розподілених системах є обробка візантійських відмов, тобто врахування можливості спотворення даних або їх втрати під час відправки.

Висновок до розділу

У даному розділі було детально розглянуто теоретичні засади алгоритмів досягнення консенсусу у розподілених системах. Передумовою для існування таких алгоритмів стала задача візантійських генералів, що простими словами описує суть

проблеми. Було розглянуто спрощений алгоритм роботи будь-якого алгоритму консенсусу, проведено детальний порівняльний аналіз алгоритмів Raft та Paxos, після чого для реалізації та подальших покращень обрано саме алгоритм Raft.

Запропоновано замінити консенсусне голосування більшості у алгоритмі Raft на динамічний протокол голосування, розглянуто три реалізації такої системи: кластер з чотирьох вузлів, що забороняє роботу системи з одним працюючим вузлом, кластер з чотирьох вузлів, що дозволяє таку роботу, та конвенційний кластер з 3-х або 5-ти вузлів. Проведено теоретичний аналіз даних реалізацій, побудовано процеси Маркова та виведено формули доступності та стійкості до роботи на двох/одному вузлах для кожної системи. Порівняння виведених показників показало, що введення динамічного протоколу дає можливість знизити енергоємність систем, зберігаючи показники доступності та стійкості до відмов.

3 ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вимоги до програмного забезпечення

Програмне забезпечення представлене у вигляді клієнт–серверного застосунку. Клієнтська частина застосунку (її веб-складова) надає виключно адміністративні функції, серверна ж частина представлена як набір API для керування розподіленою системою і може бути інтегрована з будь-якою іншою системою чи модулем.

Відповідно до схеми варіантів використання, складемо набір функціональних вимог до системи (таблиця 3.1):

Таблиця 3.1 – Функціональні вимоги до системи

№	Назва варіанту використання	Вимога	Пріоритет
1	Переглянути інформацію про всі вузли	Система повинна надавати можливість адміністратору переглядати інформацію про вузли кластера, а саме: ідентифікатор, хост, порт, статус активності, стан.	Високий
2	Оновити інформацію про усі вузли	Система повинна надавати можливість оновити таблицю з даними про усі вузли кластера.	Високий
3	Керувати станом вузла	Система повинна надавати можливість змінювати стан кожного з вузлів при виборі його з–поміж інших у таблиці.	Високий
4	Підключити вузол до кластера	Система повинна надавати можливість запустити роботу вузла, тим самим підключивши його до кластера.	Високий
5	Відключити вузол від кластера	Система повинна надавати можливість зупинити роботу вузла, тим самим відключивши його від кластера.	Високий

6	Переглянути записи кожного вузла	Система повинна надавати можливість відображення інформації логу, що зберігається всередині мапи вузла	Високий
7	Записати дані у кластер	Система повинна надавати можливість вносити у журнал кластера нові записи вигляду ключ–значення.	Високий

Для системи можна також сформувати набір деяких нефункціональних вимог:

– система повинна зберігати працездатність при відключенні будь-якого з вузлів з будь-яких незалежних від адміністратора причин;

– лог кластера повинен зберігатися та оновлюватись, на момент виходу зі строю усіх серверів він повинен мати найбільш актуальні дані;

– система повинна надавати можливість збереження у лог до 50000 записів

– система повинна надавати можливість для горизонтального масштабування – довільне збільшення кількості вузлів кластера;

– система повинна відповідати на запити клієнта до 5 секунд;

– система повинна однаково ефективно працювати у браузерах Chrome, Mozilla та Safari;

– система повинна надавати можливість реінтеграції API з будь-яким модулем сторонньої системи;

– система повинна регулярно логувати усі операції, що виконуються в кластері.

3.2 Архітектура програмного забезпечення

Структура кластера, на якому реалізується та тестується даний алгоритм, виглядає наступним чином:

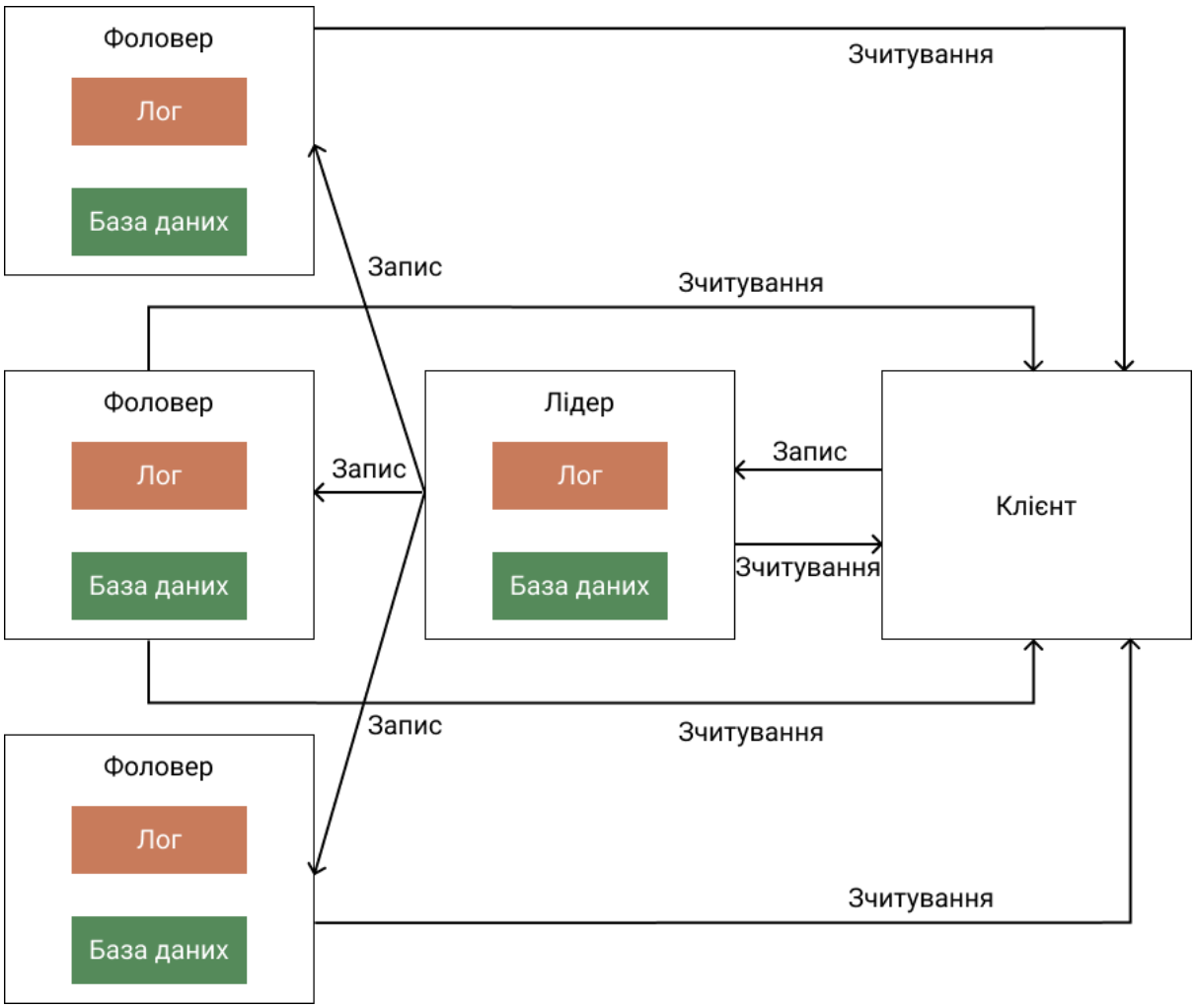


Рисунок 3.1 – Діаграма компонентів системи

Кластер складається з одного клієнтського вузла та з нескінченної кількості вузлів-серверів. Кожен вузол забезпечує доступ до сховища логу операцій, в якому послідовно фіксуються операції по зміні даних. Також кожен вузол–сервер має доступ до бази даних, у якій безпосередньо зберігаються дані. База даних і лог у кожного вузла окремі.

У поточній реалізації використовується “вбудоване” рішення як для логу так і для бази даних – конкурентоспроможні (паралелізовані) структури списку та мапи. У разі розширення функціональності системи, можна розширити створений інтерфейс на використання інших типів сховищ даних. Кожна операція, окрім типу та даних, містить в собі номер терміну, в рамках якого вона створена. Важливо, що всі операції вставляються у лог фоловера у тому ж порядку, в якому вони знаходяться у лозі лідера.

Переведення операцій із логу на базу даних (фактично – вставка даних у базу) – задача розподіленої машини станів. Машина станів – це такий механізм, який відповідає за зміну стану кластера, обмежуючи застосування некоректних змін (операції не по порядку, від “відключеного” лідера тощо).

Дані логу (операції) реплікуються лідером іншим вузлам. Після підтвердження отримання операції більшістю вузлів, дані з інформацією зберігаються у базу даних лідера та змінюється стан машини автоматів. Інші вузли отримують інформацію про підтвердження операції і також записують дані у власні бази даних.

Користувач передає дані через HTTP, зараз вони мають вигляд ключ–значення, де значення це символьний рядок. Дані зберігаються у конкурентну мапу.

Клієнтський вузол включає у себе два окремі сервери – API сервер, який відповідає за надання списку операцій, які можна застосовувати до кластеру, та WEB сервер, який являє собою інтерфейс для зручності керування кластером.

3.3 Обґрунтування вибраних методів та програмних засобів

У якості програмного засобу для реалізації серверної частини системи обрано мову програмування Java [42]. Java – мова програмування високого рівня загального призначення, зазвичай використовується для створення веб–додатків та мобільних додатків. Основні її переваги наступні:

– об'єктно–орієнтоване програмування – концепт програмування, який пропонує маніпулювати структурами даних як об'єктами, визначаючи їхню поведінку та зв'язки поміж ними. На противагу процедурному програмуванню, ООП пропонує більш організований та запланований підхід до розробки програмного забезпечення [43], допомагає уникнути помилок за рахунок інкапсуляції перевикористання функціоналу;

– мова високого рівня – Java має достатньо простий та інтуїтивний синтаксис, що робить цю мову легкою для вивчення та використання. Вона чітка та має дуже строгі обмеження щодо написання коду, що допомагає розробнику спрямувати свої думки у правильному напрямку;

– стандарт для корпоративних обчислень – Java найчастіше використовується для великих програмних продуктів. Вона містить у собі безліч бібліотек, що допомагають розробникам реалізувати будь-яку функціональність для корпоративного клієнта;

– високий рівень безпеки – сама по собі Java не захищає програму від вразливостей, проте вона має деякі особливості, що допомагають захистити програму від найбільш поширених недоліків безпеки. На відміну від мови C, Java не має покажчиків. Вона також пропонує у кожній програмі використовувати Security Manager – спеціальний інструмент, через який можна вказувати правила безпеки та доступу до даних;

– незалежність від конкретної платформи – Java програми компілюються у байт–код і працюють на будь-якій платформі, що підтримує JVM (Java Virtual Machine). Усі основні операційні системи, такі як Windows, Linux, MacOS, підтримують JVM;

– автоматичне управління пам'яттю – ефективність програм напряму залежить від пам'яті, а пам'ять обмежена. Використовуючи мови, які пропонують ручне управління пам'яттю, програмісти ризикують не виділити память для необхідних процесів, що призведе до заповнення пам'яті та відмови сервісів. Натомість збірник сміття самостійно шукає об'єкти, що більше не використовуються програмою, та очистити від них пам'ять;

– багатопоточність – для того, аби оптимізувати використання оперативної пам'яті, Java дозволяє програмі одночасно виконувати процеси у декількох незалежних потоках. Це прискорює виконання складних алгоритмів та дозволяє сервісам швидко відповідати на запити клієнта.

Основою для написання клієнтського та бекенд серверу обрано фреймворк Spring Boot [44], що по суті є розширенням фреймворку Spring для спрощення налаштувань проекту і його подальшого розвитку. Spring Boot зводить до мінімуму написання серверних конфігурацій і підтримує вбудований контейнер, який дозволяє веб–додаткам працювати незалежно і без необхідності застосування сторонніх сервісів. Центральною частиною Spring є інверсія контролю – абстрактний принцип,

набір рекомендацій для написання слабо зв’язаного коду, що означає, що кожен елемент системи повинен бути як можна більше ізольованим від інших, не покладаючись у своїй роботі на детальну реалізацію інших компонентів. Реалізацією цього принципу у Spring є впровадження залежностей. Запити до вузлів кластера відправляються конкурентно з допомогою бібліотеки java.util.concurrent та зокрема класу CompletableFuture.

Для демонстрації роботи кластера з великою кількістю вузлів, використовується Docker – інструментарій для управління ізольованими Linux–контейнерами [45]. Налаштування кластера відбувається у yaml файлах.

Для реалізації веб–сервера адміністрування кластера використовується мова Javascript та бібліотека React JS [46]. У порівнянні з багатьма іншими веб–фреймворками та бібліотеками, вона має ряд наступних переваг:

– швидкість – React дозволяє розробникам використовувати окремі частини свого додатку як на стороні клієнта, так і на стороні сервера, що в кінцевому підсумку прискорює процес розробки. Простіше кажучи, різні розробники можуть писати окремі частини, і всі внесені зміни не ламають логіку програми;

– гнучкість – у порівнянні з іншими інтерфейсними фреймворками, код React легший в обслуговуванні та більш гнучкий завдяки своїй модульній структурі. Така гнучкість, у свою чергу, заощаджує величезну кількість часу та коштів для бізнесу;

– продуктивність – бібліотека React JS була розроблена для забезпечення високої продуктивності. Ядро фреймворка пропонує віртуальну програму DOM і рендеринг на стороні сервера, завдяки чому складні програми працюють надзвичайно швидко;

– зручність використання – розгорнути React програму досить легко, якщо у вас є базові знання JavaScript. Досвідчений розробник JavaScript може легко вивчити усі тонкощі бібліотеки та сміливо використовувати її у корпоративній розробці.

3.3 Деталі реалізації

3.3.1 Програмні рішення

Далі описано деталі роботи розроблюваного алгоритму у контексті практичного застосування:

– визначення лідера клієнтом: клієнт отримує метадані системи: він надсилає кожному вузлу запит на отримання його власних метаданих, оскільки кожен вузол зберігає інформацію про свій стан, клієнт переглядає отриману інформацію та обирає вузол зі статусом “Лідер”, який гарантовано буде одним на цілий кластер, і відправляє запит на його айпі адресу;

– контекст вузла: кожен вузол зберігає у своєму контексті наступну інформацію: ідентифікатор вузла, статус (лідер, фоловер, кандидат), ідентифікатор вузла, за який даний вузол проголосував у даному терміні, номер поточного терміну, індекс останньої операції, що була збережена в лог, список вузлів кластера, когортний набір кластера;

– обробка падіння лідера: для кожного вузла в docker-compose.yaml файлі розгортання вказані election-timeout (в секундах). У системі постійно спрацьовує ElectionTimer, який через вказаний таймаут робить вузол кандидатом і ініціює нове голосування. Таймер скидається, коли фоловер отримує heartbeat від поточного лідера.

3.3.2 Специфікація функцій

У таблиці 3.2 описуються основні функції класів програмного забезпечення.

Таблиця 3.2 – Специфікація функцій

Назва класу	Назва функції	Опис функції
ElectionServiceImpl	public void processElection()	Метод, що викликається кандидатом для того, щоб зібрати голоси більшості фоловерів, тобто фактично починає нові

		вибори у кластері.
ElectionServiceImpl	List<AnswerVoteDTO> getVoteFromAllPeers(Long term, List<Integer> peers)	Метод, що розсилає паралельні запити на голосування усім вузлам кластера.
ElectionServiceImpl	CompletableFuture<AnswerVoteDTO> getVoteFromOnePeer(Integer id, Long term)	Метод, що відправляє запит на голосування одному окремому вузлу та обробляє відповідь, надану ним.
ElectionServiceImpl	winElection(Long term)	Метод, що обробляє ситуацію, коли поточний вузол–кандидат отримує більшість голосів та стає лідером. Коли лідер виграє в голосуванні, він збільшує наступний індекс кожного вузла у своєму контексті. Починаючи з цього індексу лідер буде обновляти логи фоловерів. Реальний індекс фоловера може не співпадати із тим, що має лідер, це буде коригуватися під час обміну інформацією з фоловером.
ElectionServiceImpl	loseElection(Long term)	Метод, що обробляє ситуацію, коли з якихось причин поточний вузол–кандидат програє у виборах і повертає його у стан фоловера.
ElectionServiceImpl	AnswerVoteDTO vote(RequestVoteDTO dto)	Метод, що викликається фоловером для того, щоб обробити запит на голосування від

		кандидата. Фоловер голосує або не голосує за запропоновану кандидатуру в залежності від ряду умов.
ReplicationServiceImpl	void appendRequest()	Метод, що викликається лідером для того, щоб записати значення, отримане від клієнта, у журнали фоловерів кластера. Якщо фоловер відповідає, що реплікація не вдалася, це означає, що індекс останньої операції фоловера не дорівнює попередньому індексу поточної операції, який ми йому відправили і він не може вставити цю операцію. Іншими словами, лог фоловера відстав від логу лідера більш ніж на одну операцію і фоловер не може прийняти останню операцію, тому що ще не прийняв попередні. Для цього фоловера змінюємо метадані, знижуємо індекс наступної очікуваної операції і повторюємо відправку. Ця процедура повторюватиметься, поки лог фолловера не прийде у відповідність до логу лідера.
ReplicationServiceImpl	List<AnswerAppendDTO> sendAppendToAllPeers(List<Integer> peers)	Метод, що відправляє паралельні запити на реплікацію даних усім вузлам кластера.
ReplicationServiceImpl	CompletableFuture<A	Метод, що відправляє

	nswerAppendDTO> sendAppendForOnePeer(Integer id)	запит на реплікацію даних одному окремому вузлу кластера і обробляє відповідь, отриману від нього.
ReplicationServiceImpl	AnswerAppendDTO append(RequestAppendDTO dto)	Метод, що викликається на стороні вузла–фоловера, щоб обробити запит на реплікацію даних. Якщо індекс попередньої операції, отриманий від лідера, більший за реальний індекс останнього елемента лога фоловера або їх термін не збігається, це означає, що перед тим, як вставити надіслану операцію, треба вставити попередні. Фоловер відповідає відмовою та відправляє поточний останній індекс логу, лідер має повторити спробу, але вже з попередньою операцією. Якщо у логу фоловера вже є операція з таким індексом, і вона не збігається з операцією, отриманою від лідера, видаляємо цю операцію у лозі фолловера і всі операції з більшим індексом.

3.4 Інструкція користувача

3.4.1 Перевибори

Коли адміністратор запускає кластер, він може перевірити його початковий стан користуючись веб–сторінкою розташованою за 8080 портом (рис 3.2):

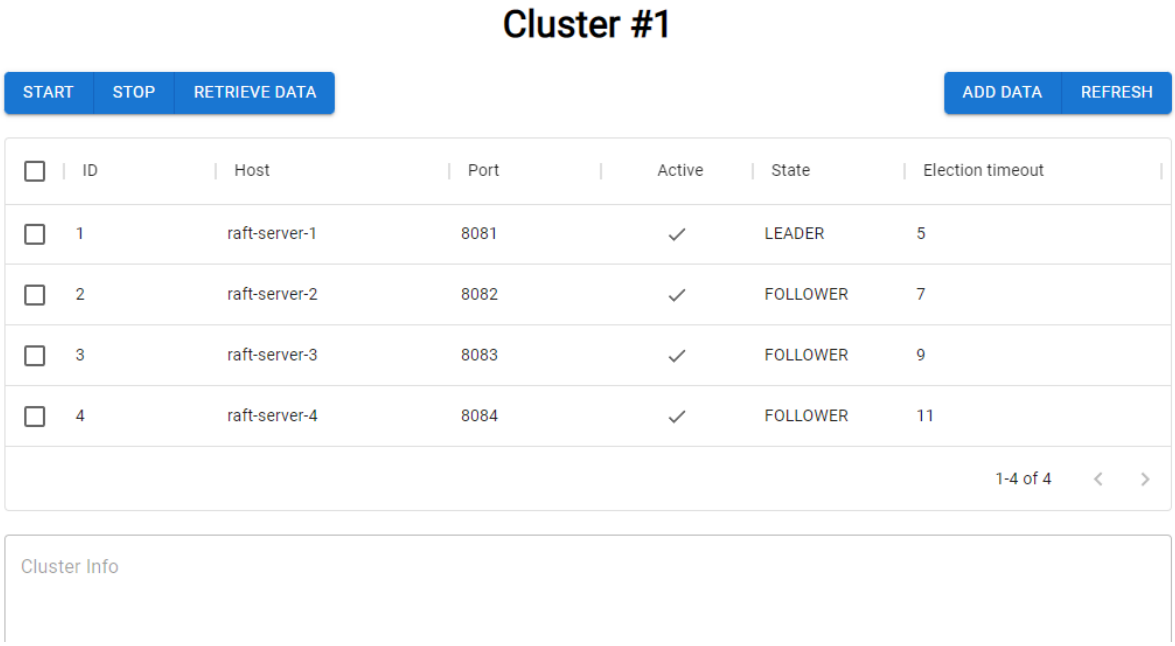


Рисунок 3.2 – Початковий стан кластера

На головній сторонці адміністратор може бачити список вузлів кластера, їхні основні атрибути, поле для відображення записів журналу вузла та контрольну панель з кнопками, що дозволяють маніпулювати вузлами. Для кожного вузла у таблиці відображається його ідентифікатор, хост, порт, статус активності, статус у системі і таймаут нових виборів.

Далі проведено перевірку того, як поведе себе кластер при втраті лідера. Це може статися у випадку непоправної аварії у вузлі-лідері, що призведе до повного його відключення від кластера. Проте для того, аби симулювати падіння віртуального сервера, адміністратор має можливість штучно відключити його від кластера. Для цього у таблиці необхідно вибрати необхідний вузол і натиснути кнопку Stop. Бачимо, що поточний лідер це вузол з ідентифікатором 1. Відключимо його від кластера і оновимо інформацію про кластер (рис 3.4–3.5):

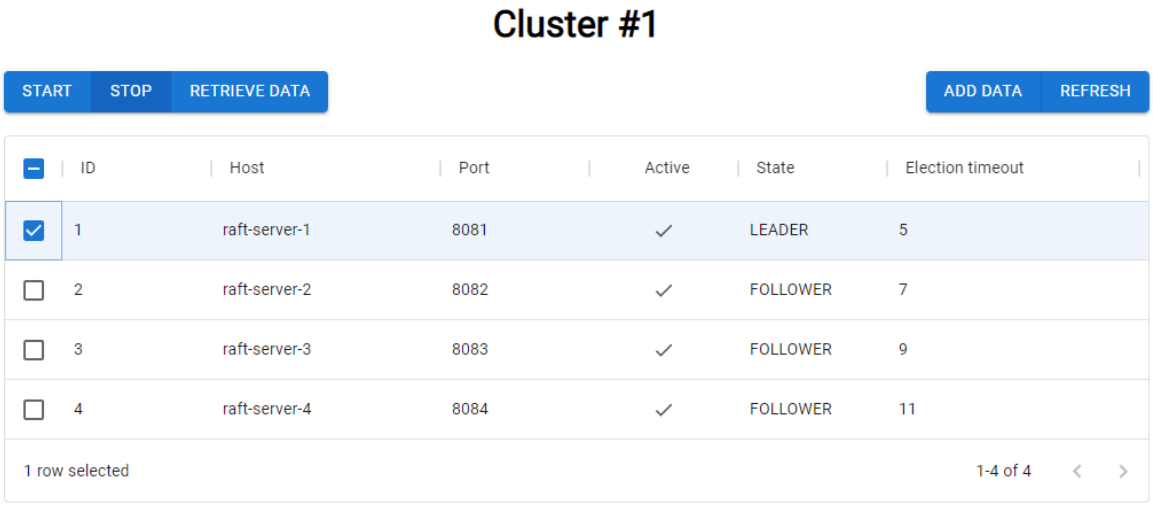


Рисунок 3.4 – Відключення вузла від кластера

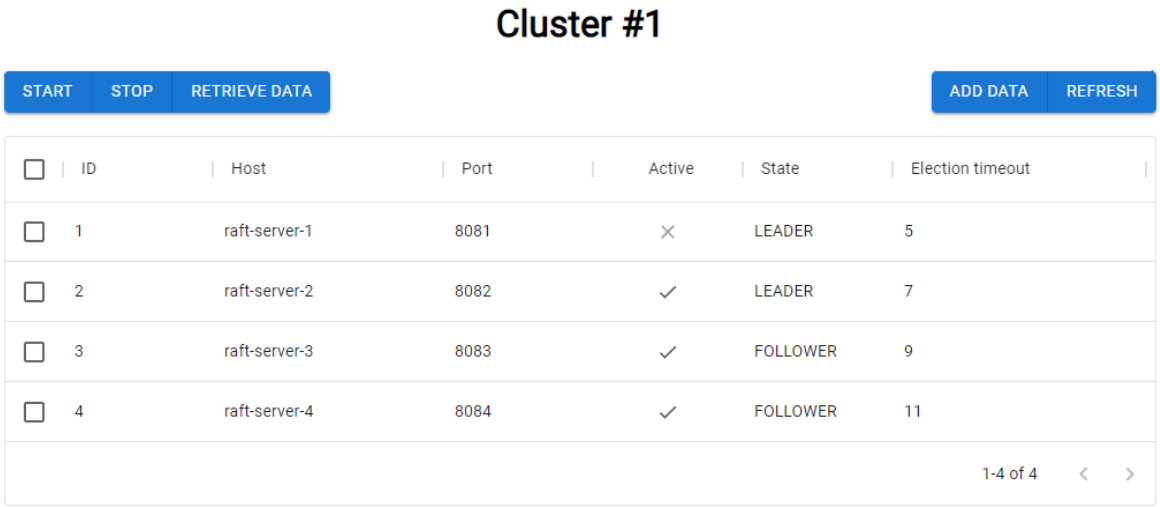


Рисунок 3.5 – Оновлена інформація про кластер

Як можна бачити, стан активності вузла 1 змінився на “неактивний”. Коли кластер залишився без лідера, таймаут другого вузла збіг першим, адже він має найменше значення, і він ініціював нові вибори. Після того, як другий вузол отримав більшість голосів, він став новим лідером кластера, що можна побачити у таблиці. Конвенційний кластер Raft при відключенні ще одного лідера перестане працювати і не обере нового лідера, адже більшість для даного вузла буде втрачена. Два вузли, які залишаться працювати у системі, будуть по черзі ініціювати нові вибори, але жодні не закінчаться успіхом, поки не відновиться перший вузол. Проте модифікований

кластер з динамічним протоколом голосування витримує падіння ще одного сервера. Відключимо поточного лідера і подивимось на результат у загальній таблиці вузлів (рис 3.6):

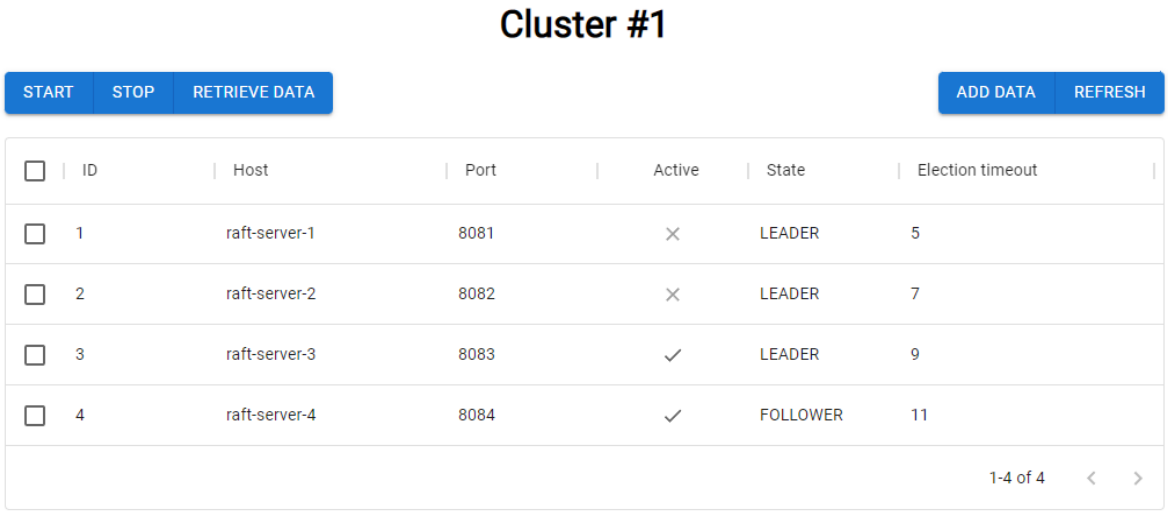


Рисунок 3.6 – Стан кластера після відключення другого вузла

Як можна бачити, система обрала нового лідера. Цілком логічно, що наступним лідером став вузол з ідентифікатором 3, оскільки його таймаут збіг швидше, ніж відповідний показник четвертого вузла, що навмисно був вказаний у конфігурації кластера як найбільший. Тепер система працює на двох вузлах, падіння ще одного з них призведе до того, що система не буде відповідати на повідомлення і залишиться у неробочому стані, адже дана реалізація використовує простий динамічний протокол голосування. Спробуємо відключити ще один вузол (рис 3.7):

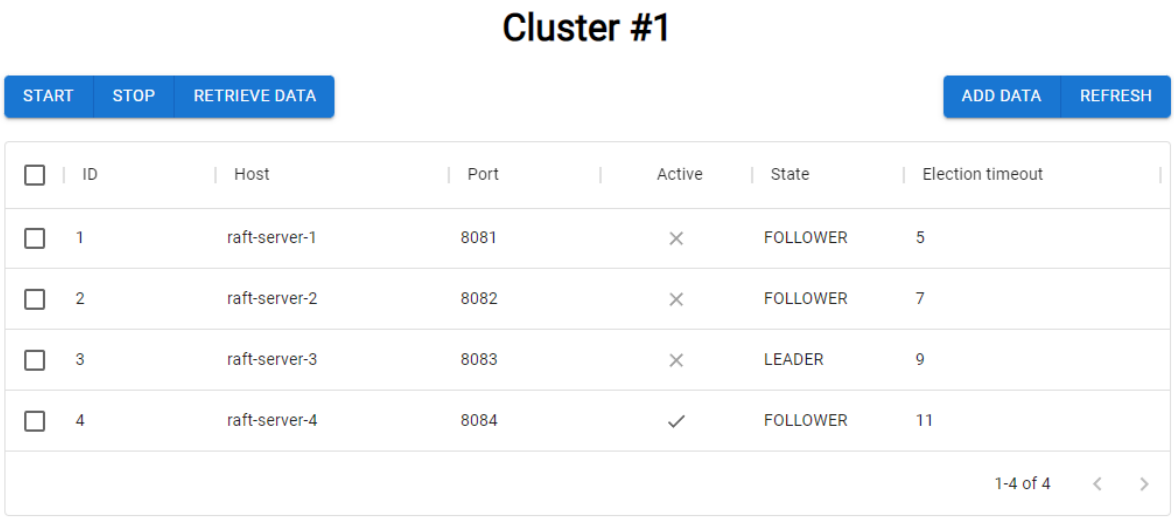


Рис 3.7 – Система у неробочому стані

Тепер система не обробляє нові запити клієнта, адже не має лідера, що відповідальний за це. Повернемо до роботи вузли, що були зупинені. Для цього обираємо необхідний вузол у таблиці і натискаємо кнопку Start. Відключені вузли повертаються до кластера зі статусом фоловерів (рис 3.8):

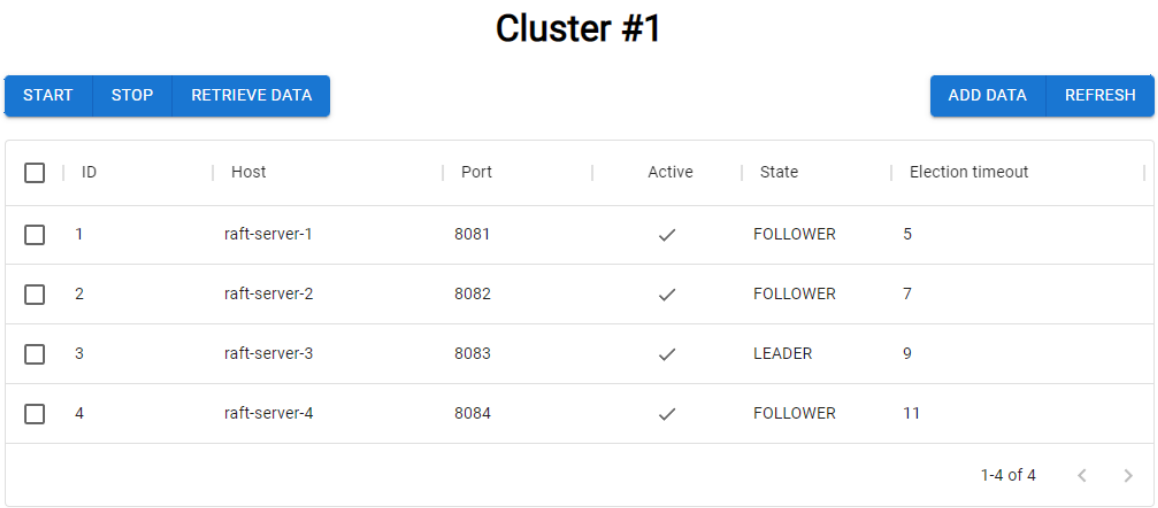


Рисунок 3.8 – Стан кластера після підключення двох серверів

3.4.2 Штатна реплікація

Далі проведено власне реплікацію даних. Адміністратор має доступ до записування даних до кластера. Усі дані наразі мають вигляд ключ-значення. Для цього йому необхідно натиснути на кнопку Add Data у контрольній панелі, після чого відкривається діалогове вікно для введення даних (рисунок 3.9) :

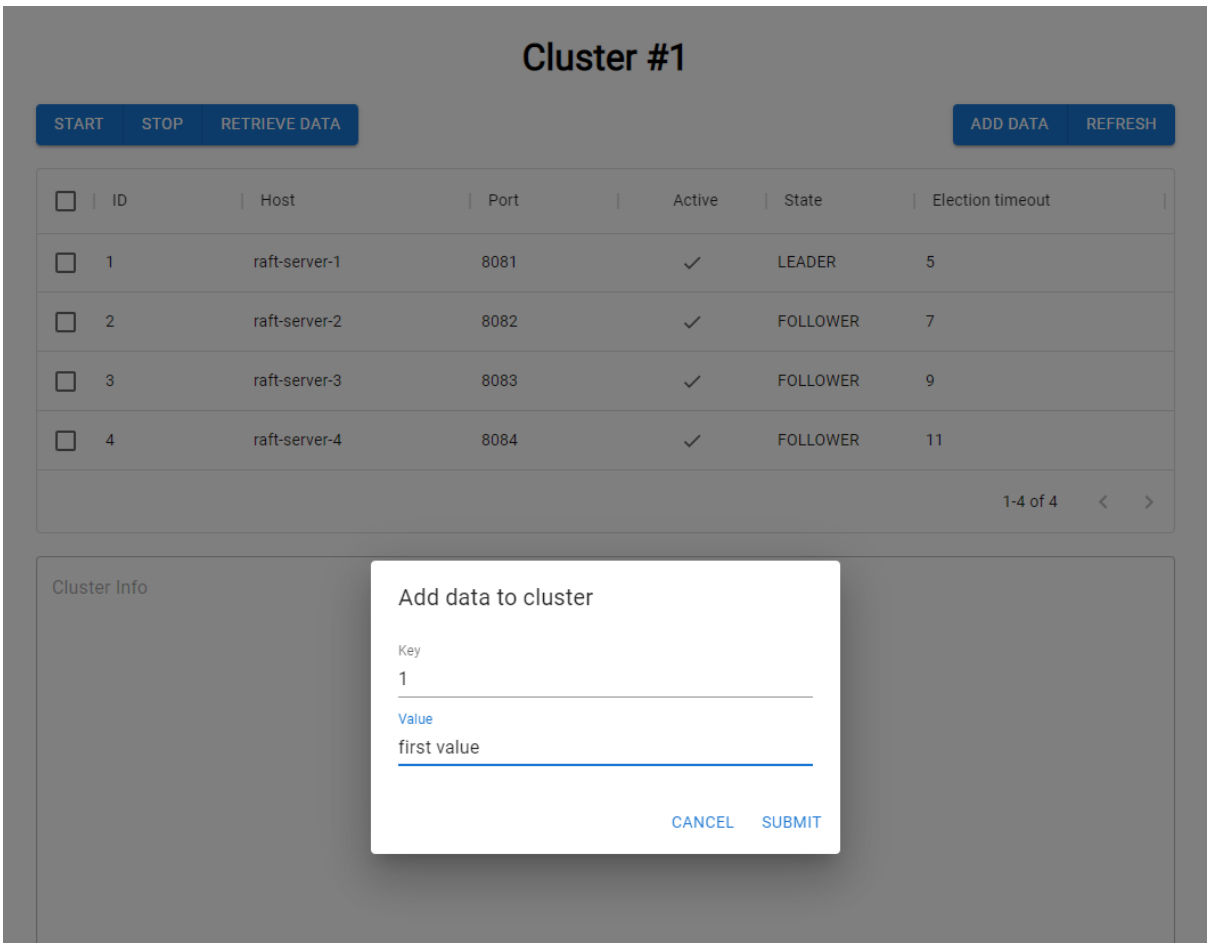


Рисунок 3.9 – Діалогове вікно для запису нових даних у кластер

Після натискання кнопки Submit дані записуються у журнал лідера, він відправляє його на голосування усім іншим вузлам. Якщо у кластері більшість вузлів одностайно голосують за запропоноване значення, воно реплікується на всіх вузлах системи. Адміністратор може перевірити це, використовуючи кнопку Retrieve data у контрольній панелі. Для цього необхідно обрати вузол, інформація з локального журналу якого цікавить адміністратора, і натиснути кнопку. У текстовому полі під таблицею відображається відформатований JSON формат збережених даних. Для початку, перевірено чи записалася інформація у журнал лідера (рис 3.10):

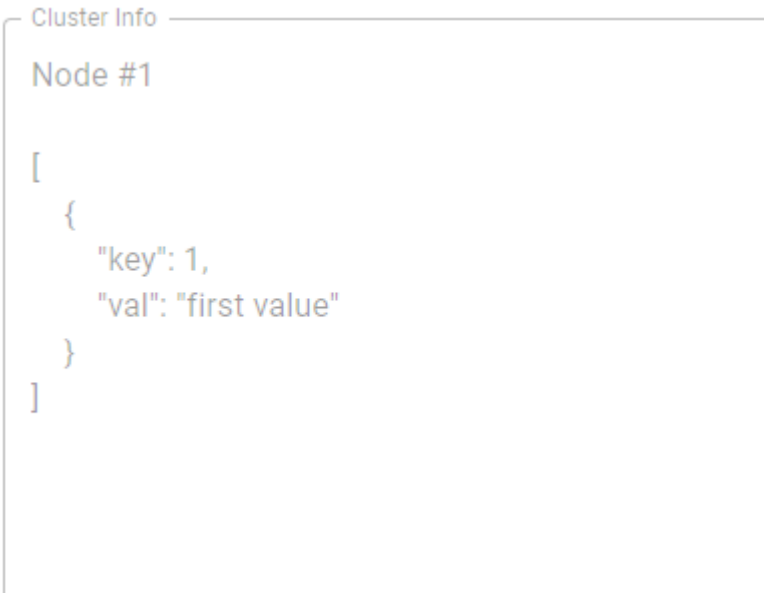


Рис 3.10 – Інформація логу вузла–лідера

Далі обрано будь-який вузол фоловер і перевірено інформацію, що збережена у його журналі (рис 3.11):

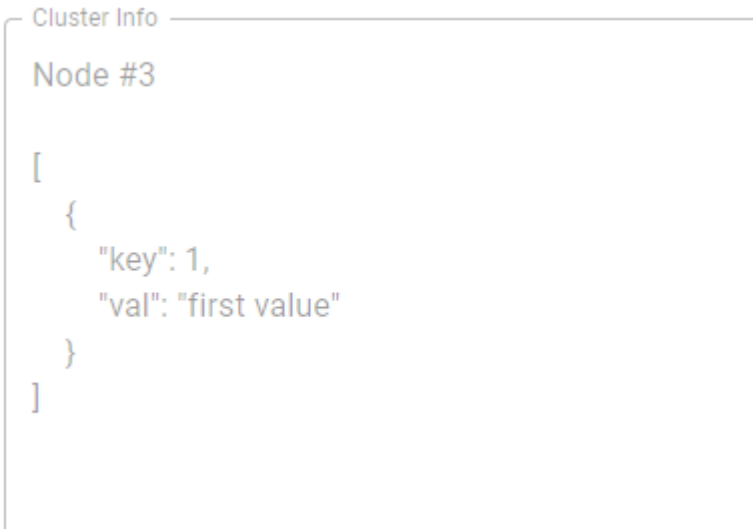


Рис 3.11 – Інформація логу вузла–фоловера

Як можна бачити, запис успішно реплікований на усіх вузлах. Можна зробити висновок про те, що система працює коректно і відповідає усім поставленим умовам.

Висновок до розділу

У даному розділі було сформовано функціональні та нефункціональні вимоги до програмного забезпечення, детально описано архітектуру програмного забезпечення, наведено діаграму структури кластеру. Було обрано основні технології та методи розробки, серед яких мова програмування Java та бібліотека react JS для розробки інтерфейсу, обґрунтовано вибір кожної з них. Описано специфікацію функцій розробленого програмного забезпечення, наведено опис основних програмних рішень для реалізації алгоритму. У розділі також описана інструкція для адміністратора розподіленої системи, який є основним користувачем даного продукту, покроково розписані етапи перевиборів у кластері та штатна реплікація отриманих від клієнта даних, наведено відповідні екранні форми.

4 РОЗРОБКА СТАРТАП-ПРОЕКТУ

4.1 Опис ідеї проекту

У рамках стартап-проекту планується створити discovery-систему Replica, що дозволятиме масштабувати системи, що складаються з багатьох сервісів, деякі з яких потребують реплікування, використання спільних даних тощо. Детальний опис проекту наведений у таблиці 4.1.

Таблиця 4.1 – Опис ідеї стартап проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Ідея полягає у розробці децентралізованого discovery-сервісу Replica, що надає необхідні інструменти для горизонтального масштабування додатків	Stateful-додатки	Налаштування реплік серверів, що містять деякий стан (конфігураційні дані, метадані, результати обчислень), без використання баз даних та оркестраторів.
	SaaS та PaaS	Використання бібліотеки для розробки власних оркестраторів задля забезпечення консистентності даних в джерелах зберігання даних.

Проведемо порівняльний аналіз запланованого стартап-проекту з наявними конкурентами (таблиця 4.2).

Таблиця 4.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№	Техніко-економічні характеристики ідеї	(потенційні) концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Replica	Zookeeper	Consul			
1	Сховище даних виду ключ-значення	+	+	+		+	
2	Динамічна реєстрація нових сервісів	+	-	+			+
3	Перевірка поточного стану вузлів	+	+	+		+	
4	Мережева томографія для побудови ефективних маршрутів для даних	-	-	+	+		
5	Перегляд даних сховища	+	-	-			+
6	Обробка користувацьких подій	-	-	+	+		
7	Фізичні репліки	-	-	-		+	

Як бачимо, у результаті порівняльного аналізу техніко-економічних характеристик ідеї сервісу Replica з його популярними конкурентами, можна зробити висновок, що сервіс є достатньо конкурентоспроможним на ринку сервісів для підтримки розподілених систем.

4.2 Технологічний аудит ідеї проекту

Для того, аби визначитися з технологіями для реалізації ідей проекту, проведемо технологічний аудит, результати якого вказані у таблиці 4.3.

Таблиця 4.3 – Технологічна здійсненність ідеї проекту

№	Техніко-економічні характеристики ідеї	Технології реалізації	Наявність технології	Доступність технології
1	Сховище даних виду ключ-значення	Java, concurrent data types	Наявна	Доступна
2	Динамічна реєстрація нових сервісів	Конфігурація yaml файлів, HTTP API	Наявна	Доступна
3	Перевірка поточного стану вузлів	Java, concurrent API	Наявна	Доступна
4	Мережева томографія для побудови ефективних маршрутів для даних	DNS	Наявна	Недоступна
5	Перегляд даних сховища	ReactJs	Наявна	Доступна
6	Обробка користувацьких подій	HTTP API, Java, React, Chronos	Наявна	Доступна
7	Фізичні репліки	LAN мережа	Наявна	Недоступна

За результатами аудиту, можна зробити висновок, що стартап-проект Replica є технологічно здійсненним.

4.3 Аналіз ринкових можливостей запуску стартап-проекту

Визначимо ринкові можливості, які можна використати під час ринкового впровадження проекту. Аналіз попиту стартап-проекту наведено у таблиці 4.4.

Таблиця 4.4 – Попередня характеристика потенційного ринку стартап-проекту

№	Показники стану ринку	Характеристика
1	Кількість головних гравців	Групу головних гравців ринку складають великі компанії, що випускають власні продукти, та маленькі стартапи
2	Загальний обсяг продаж	Клієнтами сервісів для горизонтального масштабування є більше 1000 бізнес-продуктів
3	Динаміка ринку	Зростаюча
4	Наявність обмежень для входу	Створення РОС та залучення інвестицій
5	Специфічні вимоги до стандартизації та сертифікації	Міжнародні стандарти роботи розподілених систем
6	Середня норма рентабельності в галузі	Невідома

За попереднім оцінюванням, ринок є привабливим для входження. Визначимо потенційні цільові групи нашого проекту (таблиця 4.5).

Таблиця 4.5 – Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Можливість горизонтального	Стартапи, великі enterprise проекти,	Великі проекти, на відміну від	Зручний інтерфейс, можливість

масштабування додатку, адміністрація кластера	що впроваджують у свої застосунки нові сервіси та вбачають потребу у горизонтальному масштабуванні системи.	стартапів, потребують більшої кількості функціоналу у discovery-сервісах, що підвищує їхній рівень очікувань	інтеграції на будь-якій стадії розробки проекту, функціонал керування розподіленою системою

Тепер проведемо аналіз ринкового середовища, а саме, складемо таблицю факторів загроз (таблиця 4.6) та факторів можливостей (таблиця 4.7) для стартап-проекту Replica.

Таблиця 4.6 – Фактори загроз

Фактор	Зміст загрози	Реакція компанії
Поява нових конкурентів	Поява нових discovery-сервісів на ринку продажів продуктів адміністрування розподілених систем, що надають нову функціональність, чим приваблюють клієнтів	Спостерігати за розвитком ринку, постійно розвивати власний продукт
Нестача інвестицій	Нестача коштів на розвиток проекту	Оптимізація витрат, пошук нових інвесторів, демонстрація проекту на воркшопах та бізнес-зустрічах
Розвиток технологій великих корпорацій	Розвиток існуючих продуктів або створення нових великими корпораціями, що уже мають велику клієнтську базу, та можуть запропонувати більший обсяг послуг	Розглянути можливість інтеграції з сервісами-гігантами

Таблиця 4.7 – Фактори можливостей

Фактор	Зміст можливості	Реакція компанії
Вихід на світовий ринок	Впровадження продукту на міжнародні платформи	Робота над подальшим розвитком сервісу, його локалізація та регіоналізація
Партнерство	Співпраця з великими сервісами з адміністрування та оркестрації розподілених систем, впровадження власного рішення у рамках розвинутих продуктів	Робота на системою інтеграції

У таблиці 4.8 наведено аналіз існуючих пропозицій на ринку та визначено загальні риси конкуренції.

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства
Чиста конкуренція	На ринку є багато рішень для впровадження горизонтального масштабування, усі вони користуються однаковою популярністю, задовольняючи усі потреби бізнесу	Можливість розвивати продукт в умовах чистої конкуренції
Міжнародна конкурентна боротьба	Продукти не є залежними від локальних особливостей, вони продаються будь-якому бізнесу	Можливість вийти на міжнародний ринок продажів
Внутрішньогалузева боротьба	Усі продукти конкурують на b2b ринку IT послуг	Високий рівень конкуренції

Товарно-видова конкуренція	Конкуренція між послугами одного виду та призначення	Високий рівень конкуренції
Нецінова перевага	Перевага за рахунок якості наданого продуктом функціоналу	Можливість вигравати на ринку за рахунок якості реалізації продукту
Не марочна інтенсивність	Марка продукту не грає великої ролі	Можливість вигравати на ринку за рахунок якості реалізації продукту

Тепер проведемо більш детальний аналіз умов конкуренції в галузі, використовуючи класифікацію Портера (таблиця 4.9).

Таблиця 4.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Zookeeper, Consul	AWS, Azure, Google Cloud	Продукт не залежить від постачальників	Розвиток малого та середнього бізнесу	Вища якість наданих послуг, нижча ціна
Висновки	Конкуренція в галузі досить висока	Існує велика кількість потенційних конкурентів, проте попит залишається високим	Постачальники не диктують умови на ринку	Так, клієнти напряму визначають функціональність discovery-систем, якої вони потребують	Обмеження відсутні

З огляду на конкурентну ситуацію, можливості роботи на ринку галузі наявні. Продукт повинен мати обумовлену ціну, високу якість реалізації та покривати усі потреби користувачів.

На основі вищенаведеного аналізу конкуренції, а також із урахуванням характеристик ідеї проекту, вимог споживачів до товару та факторів маркетингового середовища, визначимо перелік факторів конкурентоспроможності (таблиця 4.10).

Таблиця 4.10 – Обґрунтування факторів конкурентоспроможності

Фактор конкурентоспроможності	Обґрунтування
Функціонал	Наявність доступу до сховища даних типу ключ-значення, функціонал мережевої топології, зручна адміністративна панель, динамічна конфігурація кластера тощо.
Ефективність	Правильність та швидкість роботи алгоритму досягнення консенсусу у розподілених системах, що прискорює обробку клієнтських запитів та підвищує доступність системи.
Цінова політика	Справедлива та конкурентоспроможна ціна на продукт

За визначеними факторами конкурентоспроможності, проведемо аналіз сильних та слабких сторін стартап-проекту (таблиця 4.11).

Таблиця 4.11 – Порівняльний аналіз сильних та слабких сторін

Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів конкурентів у порівнянні з Replica						
		-3	-2	-1	0	1	2	3
Функціонал	15						+	
Ефективність	16				+			
Цінова політика	18			+				

На основі попередньо проведеного аналізу ринкових можливостей, складемо SWOT-матрицю (таблиця 4.12), у якій підсумуємо усі описані чинники.

Таблиця 4.12 – SWOT-аналіз стартап-проекту

Сильні сторони: ефективність використовуваного алгоритму, легкість в налаштуванні та простота використання	Слабкі сторони: недостатня кількість необхідного функціоналу у порівнянні з конкурентами
Можливості: вихід на світовий ринок, партнерство з бізнес-гігантами	Загрози: поява нових конкурентів, розвиток технологій великих корпорацій, нестача інвестицій

4.4 Розробка ринкової стратегії проекту

Для розробки ринкової стратегії, визначимо цільові групи споживачів (таблиця 4.13).

Таблиця 4.13 - Вибір цільових груп потенційних споживачів

Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
Стартапи	Висока	85%	Висока	Середня
Великі корпорації	Низька	60%	Висока	Середня

Обрано зосередитись на невеликих стартап-проектах у якості клієнтів, що автоматично визначає стратегію концентрованого маркетингу як основну у розвитку продукту. Базова стратегія розвитку описана у таблиці 4.14.

Таблиця 4.14 – Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку

Розробка discovery-системи для горизонтального масштабування та адміністрування розподілених систем невеликих розмірів	Стратегія концентрованого маркетингу	Ефективність, універсальність, простота користування та налаштування	Стратегія спеціалізації

Наступним кроком є вибір стратегії конкурентної поведінки (таблиця 4.15).

Таблиця 4.15 – Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
Ні	Так	Так, основний функціонал систем для горизонтального масштабування	Стратегія наслідування лідеру

4.5 Розроблення маркетингової стратегії стартап-проекту

Підсумуємо попередні результати, сформувавши маркетингову концепцію продукту. Таблиця 4.16 відображає основні переваги обраної концепції.

Таблиця 4.16 – Визначення ключових переваг концепції потенційного товару

Потреба	Вигода яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
Функціонал	Розширений функціонал продукту	Нові можливості для адміністраторів

		розподілених систем (мережева топологія, підтримка користувацьких подій, сесії)
Простота налаштування	Легкість конфігурації розподіленої системи, динамічне розширення системи	Зручний та універсальний інтерфейс
Ефективність	Швидке прийняття консенсусу у системі	Ефективний розроблений алгоритм прийняття консенсусу

У таблиці 4.17 відображена трьохрівнева маркетингова модель, у якій уточнюється ідея продукту та його основні характеристики.

Таблиця 4.17 – Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>I. Товар за задумом</td><td colspan="3">Discovery-система керування розподіленими системами</td></tr>
<tr><td rowspan="4">II. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>1. Легкість конфігурації
2. Розширений функціонал
3. Ефективність
4. Відмовостійкість</td><td></td><td></td></tr>
<tr><td colspan="3">Якість: система відповідає міжнародним стандартам архітектури розподілених систем</td></tr>
<tr><td colspan="3">Продукт постачається у вигляді бібліотеки, яку можна підключати через maven/gradle artifactory, або скачувати як zip-файл, та серверу розгортання, який можна скачати як zip-файл, або використати інсталятор</td></tr>
<tr><td rowspan="2">III. Товар із підкріпленням</td><td colspan="3">Марка: Replica – інструмент керування розподіленими системами</td></tr>
<tr><td colspan="3">До продажу: бібліотека та сервер</td></tr>
<tr><td colspan="4">Після продажу: технічна підтримка</td></tr>
<tr><td colspan="4">За рахунок чого потенційний товар буде захищено від копіювання:
Захист інтелектуальної власності</td></tr>
</table>

Визначимо межі встановлення ціни для потенційного товару (таблиця 4.18). Оскільки товар не є фізичним, а є інтелектуальною власністю розробки програмного забезпечення, товарів-замінників у нього немає.

Таблиця 4.18 – Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
Відсутні	250$ на рік на групу з 10-ти аккаунтів	Високий	До 250$ на рік на групу з 10-ти аккаунтів

Визначимо оптимальну систему збуту (таблиця 4.19), що визначає як саме продукт буде постачатися користувачеві.

Таблиця 4.19 – Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Купівля щорічної підписки та інтеграція з існуючими системами	Аналіз ринку, технічна підтримка клієнтів	Канал першого рівня	Через сайт виробника

Розробимо концепцію маркетингової комунікації, за основу якої візьмемо стратегії позиціювання та специфіку поведінки клієнта (таблиця 4.20)

Таблиця 4.20 – Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Клієнти дізнаються про товар на конференціях та статей в інтернеті	Інтернет	Адміністрування розподілених систем	Проінформувати про існування продукту	Replica - розподілена система це просто

Висновок до розділу

У даному розділі було детально розглянуто стартап-проект Replica, описано його ідею, сильні та слабкі сторони, проведено технологічний аудит. Було проаналізовано ринкові можливості запуску проекту та розроблено маркетингову стратегію.

ВИСНОВКИ

У рамках даної магістерської дисертації було розроблено систему підтримки прийняття консенсусу між серверами у розподілених системах. Відповідно до поставлених завдань дослідження, було введено поняття розподілених систем та визначено задачу визначення консенсусу, проаналізовано ряд наукових праць та статей. Було також детально розглянуто алгоритми Raft та Paxos – описано етапи роботи цих алгоритмів, наведено графіки, що відображають операції вибору лідера серед вузлів кластера та реплікації логу. Порівняльна характеристика даних алгоритмів призвела до визначення алгоритму Raft як такого, якому надається перевага у вивченні, реалізації та подальшій оптимізації.

Для оптимізації алгоритму Raft досягнення консенсусу у розподілених системах було обрано метод введення динамічного протоколу голосування замість консенсусного голосування більшості під час вибору лідера кластера та реплікації операцій логу. Проаналізовано дві реалізації такого підходу: простий динамічний та динамічний лінійний протоколи голосування, проведено теоретичне порівняння кластерів із нововведеннями з конвенційними Raft кластерами. Аналіз показників доступності та стійкості до відмов вузлів показав, що кластер з динамічним протоколом голосування, що складається з чотирьох вузлів, є практично таким же доступним та стійким до відмов, як конвенційний кластер з п'яти вузлів, що дозволяє говорити про те, що введення динамічного голосування робить систему на 25% менш енергоємною, оскільки вимагає використання меншої кількості вузлів.

Для практичної перевірки наведених реалізацій алгоритмів було налаштовано тестову систему у віртуальному середовищі, що складається з декількох вузлів і обробляє запити клієнта. Система легка в масштабуванні та керуванні та цілком дозволяє оцінити роботу реалізованих алгоритмів.

У рамках розробки стартап-проекту запропоновано розробити та впровадити проект Replica, для якого проведено детальний аналіз ринку, сильних та слабких сторін, розроблено маркетингову стратегію та базову стратегію конкурентної поведінки, сформовано систему збуту та концепцію маркетингових комунікацій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. XI, XII Міжнародна науково-практична конференція з інформаційних джерел та технологій INFOCOM ADVANCED SOLUTIONS
2. Nadiminti K. Distributed Systems and Recent Innovations: Challenges and Benefits / K. Nadiminti, M. Assuncao. – 2006.
3. Wei–Tek T. Software–as–a–service (SaaS): Perspectives and challenges / T. Wei–Tek, B. Xiaoiying. // Science China. Information Sciences. – 2014. – №57. – С. 1–15.
4. Coulouris G. Distributed Systems: Concept and Design / G. Coulouris, J. Dollimore, T. Kindberg., 2011. – 1008 с. – (5).
5. Steen M. Distributed Systems / M. Steen, A. Tanenbaum., 2017. – (3).
6. Microservices: Architecting for Continuous Delivery and DevOps. // IEEE International Conference on Software Architecture. – 2018.
7. Barborak M. The Consensus Problem in Fault–Tolerant Computing / M. Barborak, M. Malek. // ACM Computing Surveys. – 1993. – №25. – С. 171–220.
8. Kshemkalyani A. Distributed Computing Principles, Algorithms, and Systems / A. Kshemkalyani, M. Singhal. – New York: Cambridge University Press, 2008. – 731 с. – (1).
9. Jalili P. High performance state–machine replication / P. Jalili, F. Penone. // Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks. – 2011.
10. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC, pp. 305–319 (2014)
11. FISCHER M. Impossibility of Distributed Consensus with One Faulty Process / M. FISCHER, N. LYNCH, M. PATERSON. // Journal of the Association for Computing Machinery. – 1985. – №32. – С. 374–382.
12. Ailijiang A. Consensus in the Cloud: Paxos Systems Demystified / A. Ailijiang, A. Charapko. – Buffalo: Computer Science and Engineering University at Buffalo, 2014.

13. Chaudhry N. Consensus Algorithms in Blockchain: Comparative Analysis, Challenges and Opportunities / N. Chaudhry, M. Yusaf. // 2018 12th International Conference on Open Source Systems and Technologies (ICOSST). – 2018.
14. Lamport, L.: The part–time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
15. Lamport, L., et al.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)
16. Corbett, J.C., Dean, J., Epstein, M., et al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. 31(3), 8:1–8:22 (2013)
17. Baker, J., Bond, C., Corbett, J.C., et al.: Megastore: providing scalable, highly available storage for interactive services. In: CIDR, pp. 223–234 (2011)
18. Zheng, J., Lin, Q., Xu, J., et al.: Paxosstore: high–availability storage made practical in WeChat. PVLDB 10(12), 1730–1741 (2017)
19. Rao, J., Shekita, E.J., Tata, S.: Using paxos to build a scalable, consistent, and highly available datastore. PVLDB 4(4), 243–254 (2011)
20. Howard H. Analysis of Raft Consensus / Heidi Howard. // Cambridge Computer Laboratory. – 2014. – №857.
21. etcd. https://etcd.io/
22. Tidb. https://pingcap.com/
23. Cao, W., et al.: PolarFS: an ultra–low latency and failure resilient distributed file system for shared storage cloud database. PVLDB 11(12), 1849–1862 (2018)
24. Arora, V., et al.: Leader or majority: Why have one when you can have both? improving read scalability in raft–like consensus protocols. In: HotCloud (2017)
25. Poke, M., Hoefler, T.: DARE: high–performance state machine replication on RDMA networks. In: HPDC, pp. 107–118 (2015)
26. Mortier R. Paxos vs Raft: Have we reached consensus on distributed consensus? / R. Mortier, H. Howard. // Workshop on Principles and Practice of Consistency for Distributed Data. – 2020. – №7.
27. Pease M. Reaching agreement in the presence of faults / M. Pease, R. Shostak, L. Lamport. // Journal of the ACM (JACM). – 1980. – №27. – С. 228–234.
28. Apache Zookeeper. https://zookeeper.apache.org/

29. The Hazelcast and Apache Kafka Transaction Processing Reference Architecture [Електронний ресурс]. – 2015. – Режим доступу до ресурсу:https://hazelcast.com/lp/the–hazelcast–and–apache–kafka–transaction–processing–reference–architecture
30. PhxPaxos. https://github.com/Tencent/phxpaxos
31. Chandra T. Paxos Made Live – An Engineering Perspective / T. Chandra, R. Griesemer, J. Redstone. // ACM PODC. – 2007.
32. Schneider F. Implementing fault–tolerant services using the state machine approach: A tutorial / F. B. Schneider. // ACM Computing Surveys. – 1990. – №22. – С. 299–319.
33. L. Lamport, “Time, clocks and the ordering of events in a distributed system,” Communications of the ACM, 21(7): 58–65, July 1978
34. Lamport L. Time, clocks and the ordering of events in a distributed system / Leslie Lamport. // Communications of the ACM. – 1978. – №21. – С. 58–65.
35. Как сервера договариваются друг с другом: алгоритм распределенного консенсуса Raft [Електронний ресурс] // Habr. – 2011. – Режим доступу до ресурсу: https://habr.com/en/company/dododev/blog/469999/.
36. Seguin J. A majority consensus algorithm for the consistency of duplicated and distributed information / J. Seguin, G. Sergeant, P. Wilms. // First International Conference on Distributed Computing Systems. – 1979. – С. 617–624.
37. Thomas R. A majority consensus approach to concurrency control / R.H. Thomas. – 1979. – №4. – С. 180–209.
38. Jehan–François P. Pirogue, a lighter dynamic version of the Raft distributed consensus algorithm / P. Jehan–François, D. Long. – 2014.
39. Jajodia S. Dynamic voting algorithms for maintaining the consistency of a replicated database / S. Jajodia, D. Mutchler. // ACM Trans. on Database Systems. – 1990. – №15. – С. 230–280.
40. Agbinya J. Markov Chain and its Applications an Introduction / Johnson Agbinya. – Copenhagen: River Publishers Denmark, 2020.

41. Ongaro D. CONSENSUS: BRIDGING THEORY AND PRACTICE : дис. докт. техн. наук / Ongaro Diego – Stanford, 2014. – 258 с.
42. Bloch J. Effective Java / Joshua Bloch., 2017. – (3rd).
43. McLaughlin B. Head First Object–Oriented Analysis and Design / Brett McLaughlin., 2007. – (1st).
44. Walls C. Spring Boot in Action / Craig Walls., 2016. – (1st).
45. Turnbull J. The docker book [Електронний ресурс] / James Turnbull // appliedstemcell. – 2017. – Режим доступу до ресурсу: https://www.appliedstemcell.com/pub/media/wysiwyg/dockerbook.pdf.
46. Banks A. Learning React: Functional Web Development with React and Redux / A. Banks, E. Porcello.