СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

Пояснювальна записка

до магістерської дипломної роботи
другого (магістерського) рівня визої освіти
(освітньо-кваліфікаційний рівень)

на тему: Децентралізована платформа підтримки blockchain-рішень

Виконав: студент 2 курсу, групи ІСТ-21дм

спеціальності 126 Інформаційні системи та технології
(шифр і назва спеціальності)

Жеро М.А.
(прізвище та ініціали)

Керівник: Захожай О.І.
(прізвище та ініціали)

Рецензент:
(прізвище та ініціали)

Київ – 2022 року

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ ДО МАГІСТЕРСЬКОЇ ДИПЛОМНОЇ РОБОТИ

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ
Факультет інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування

Освітньо-кваліфікаційний рівень магістр
Спеціальність 126 Інформаційні системи та технології
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТП
_____ д.т.н., доц. Лифар В.О.
(підпис)
"16" листопада 2022 р.

ЗАВДАННЯ
на магістерську дипломну роботу студенту
ЖЕРО Максим Андрійович
(прізвище, ім'я, по батькові)

1. Тема роботи Децентралізована платформа підтримки blockchain-рішень

керівник роботи Захожай О.І., д.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "__"____20_року №__
2. Строк подання студентом роботи 16 листопада 2022 року
3. Вихідні дані до роботи Дослідити методи і засоби децентралізації blockchain-платформ

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) вступ, аналіз предметної області, дослідження методів і засобів, реалізація рішення

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 08.10.2022 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області	08.10 - 12.10	
2	Дослідження методів і засобів побудови платформи	12.10 - 18.10	
3	Аналіз результатів	18.10 - 01.11	
4	Оформлення роботи	01.11 - 16.11	

Студент _____ Жеро М.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Захожай О.І.
(підпис) (прізвище та ініціали)

ЗМІСТ

ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ

PoW – Proof of Work

PoS – Proof of Stake

DPoS – Delegated Proof of Stake

BFT – Byzantine Fault Tolerance

PBFT – Practical Byzantine Fault Tolerance

HBBFT – Honey Badger Byzantine Fault Tolerance

PEG – Parsing expression grammar

CFG – Context free grammar

VM – Virtual Machine

PVM – Program virtual machine

MRE – Managed runtime environment

ISA – Instruction set architecture

EVM – Ethereum Virtual Machine

RPC – Remote Procedure Call

REST – Representational State Transfer

I/O – Input/Output operations

ВСТУП

З кожним роком кількість інформації непреривно збільшується, а кількість цифрової інформації і поготів, та, судячи з усього, ця тенденція не планує припинятися. Із ростом кількості цифрової інформації, зростає і кількість ресурсів, необхідних для обробки цієї інформації. Хоча вартість цих ресурсів непреривно зменшується, проте великі системи потребують чимало коштів. Системи стають складнішими, необхідно ще більше потужностей для обробки інформації.

Такі системи складно підтримувати, саме тому і почався пошук альтернативних рішень. В останній час все більше людей дивиться у бік децентралізованих рішень. Проте, у існуючих рішень є проблеми із масштабованість та швидкістю транзакцій в тому, чи іншому вигляді.

Вирішення даних проблем може кардинально змінити укореніле уявлення про системи обробки інформації, та дати поштовх до створення більш оптимальних систем.

Результати можуть бути використані у всіх галузях виробництва, де тим чи іншим чином оброблюється інформація. Саме тому обрана проблематика дослідження є вельми актуальною на даний момент.

Метою даної магістерської дисертації є дослідження концепцій для створення гнучкої платформи для розробки швидких децентралізованих додатків та реалізація прототипу платформи. Проаналізувати існуючі платформи, запропонувати алгоритм консенсусу даних в розподілених системах, алгоритм сереалізації даних, механізми для оптимізації використання ресурсів в децентралізованих системах, спроектувати систему котра підтримує запропоновані концепції.

Відповідно до мети роботи необхідно вирішити такі завдання:

- розглянути поняття про платформи для створення децентралізованих додатків;
- описати устрій спроектованої системи для розробки блокчейн додатків;
- описати реалізацію запропонованих концепцій;
- навести результати випробовування розробленого рішення;

– розробити стартап-проект.

Об'єкт дослідження – платформа для створення децентралізованих додатків.

Предметом дослідження є концепції побудови децентралізованих додатків, які дозволяють створювати ефективні рішеня.

Основним здобутком дисертації є методологія проектування алгоритму консесусу, для швидкого та надійного обміну даними у децентралізованих мережах, проектування алгоритму пошуку вузлів, для стабілізації децентралізованої системи та проектування механізмів використання ресурсів для підтримки децентралізованих додатків, зокрема проектування та оптимізація програмних віртуальних машин для виконання криптографічних операцій та оптимізація механізма виконання I/O операцій.

Отримана в результаті виконаної дослідницької роботи методологія оптимізації були опубліковані у статті «ЗАХИЩЕНА P2P КОМУНІКАЦІЯ НА ОСНОВІ ТЕХНОЛОГІЇ BLOCKCHAIN» в сучасному науковому журналі «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення» в 2019 році.

1 АНАЛІЗ ІСНУЮЧИХ ПЛАТФОРМ ДЛЯ СТВОРЕННЯ ДЕЦЕНТРАЛІЗОВАНИХ ДОДАТКІВ

1.1 Платформа для створення додатків Ethereum

Платформа для створення практично будь-яких децентралізованих онлайн-сервісів на базі блокчейна (Ðapps), що працюють на базі розумних контрактів. Реалізована як єдина децентралізована віртуальна машина. Ідея була втілена 30 липня 2015 року. Оскільки Ethereum сильно спрощує і здешевлює впровадження блокчейна, його впроваджують як великі гравці, такі як Microsoft, IBM, Acronis, Сбербанк, банківський консорціум R3, так і нові стартапи.

У кінці 2013 року Ethereum запропонував дослідник і програміст криптовалют Віталік Бутерін. Розвиток фінансувався через інтернет-краудфандінг, який проходив у липні-серпні 2014 року. Потім система вийшла у світ 30 липня 2015 року, при цьому 72 мільйони монет були створені "попередньо". Це становить близько 68 відсотків загального обсягу постачання у 2019 році [1].

У 2016 році, внаслідок експлуатації вразливості в програмному забезпеченні проекту, та подальшого розкрадання ефіру на суму 50 мільйонів доларів, Ethereum було розділено на дві окремі блокчейн – нова окрема версія стала Ethereum (ETH), а оригінал продовжувався як Ethereum Classic (ETC).

Ethereum є дуже потужною платформою розробки через технологію розумних контрактів.

Розумний контракт – це код, який працює на EVM. Смарт-контракти можуть приймати та зберігати ефір, дані або комбінацію обох. Потім, використовуючи логіку, запрограмовану в договорі, він може поширювати цей ефір на інші рахунки або навіть інші смарт-контракти.

Як приклад, приведемо відомий контракт з Бобом та Алісою (рисунок 1.1). Аліса хоче найняти Боба, щоб побудувати їй внутрішній дворик, і вони використовують контракт для депонування (місце для зберігання грошей, доки умова не буде виконана), щоб зберігати свій ефір до остаточної транзакції.

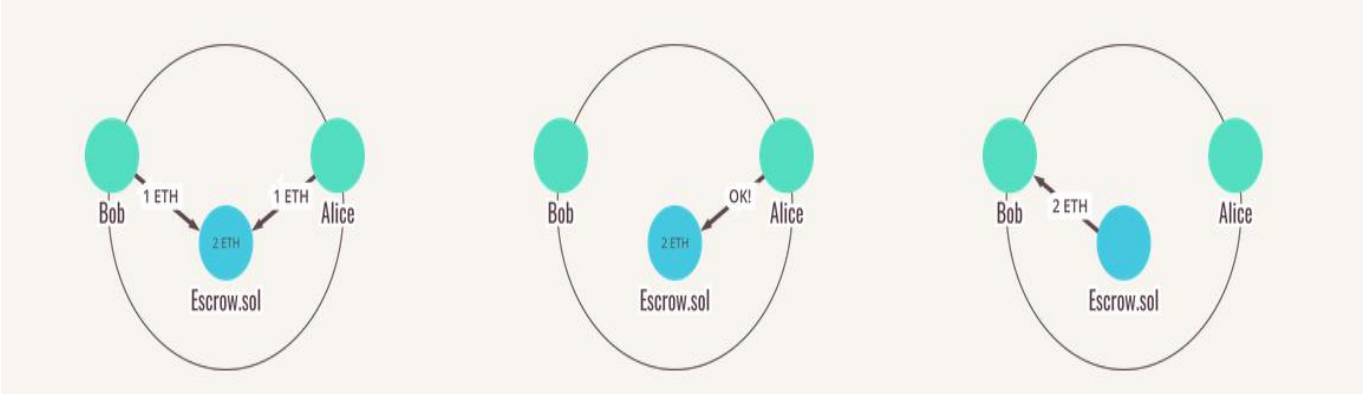


Рисунок 1.1 – Приклад виконання розумного контракта Боба та Аліси

Розумні контракти пишуться мовою, що називається "Solidity". Solidity має статичний тип і підтримує механізм успадкування, бібліотеки та складні,визначені користувачем типи. Синтаксис Solidity схожий на JavaScript [2].

Програми, що використовують смарт-контракти для їх обробки, називаються "децентралізованими програмами або "dapps".

Користувацькі інтерфейси для цих daaps складаються з відомих мов, таких як HTML, CSS та JavaScript. Саму програму можна розмістити на традиційному веб-сервері або на децентралізованій файловій службі, наприклад, Swarm або IPFS.

Враховуючи переваги блокчейна Ethereum, dapp може бути рішенням для багатьох галузей, включаючи, але не обмежуючись ними:

- бухгалтерський облік;
- фінанси;
- логістика;
- нерухомість;
- електронні магазиги (маркетплейси).

1.2 Платформа для створення додатків Hyperledger

Комплексний проект розробки блокчейну з відкритим вихідним кодом та пов'язаних з цим інструментів, який було розпочато Linux Foundation у грудні

2015 року. Основною метою проекту є підтримка спільної розробки мереж з розподіленим реєстром, заснованих на технології блокчейн.

У грудні 2015 року Linux Foundation анонсувала створення проекту Hyperledger. Імена компаній-засновників були оголошені у лютому 2016, до яких 29 березня того ж року приєдналися ще десять учасників і було затверджено склад ради правління. 29 травня виконавчим директором проекту призначили Брайана Белендорфа [3].

Метою проекту є посилення міжгалузевої співпраці за допомогою технології блокчейну та мереж з розподіленим реєстром.

Особлива увага приділяється підвищенню продуктивності та надійності цих систем (у порівнянні з аналогічними криптовалютними розробками), аби вони могли використовуватися технологічними, фінансовими та компаніями-постачальниками в масштабах глобальних комерційних оборудок.

Проект поєднуватиме незалежні відкриті стандарти та протоколи за допомогою фреймворків для створення специфічних модулей, включно з блокчейнами з власними механізмами досягнення консенсусу та порядком збереження даних, а також ідентифікаційними сервісами, контролем доступу та смарт-контрактами. Попри чутки, згідно з заявою Брайана Белендорфа, введення та використання власної криптовалюти у проекті ніколи не відбудеться.

На початку 2016 проект розпочав розглядати пропозиції щодо створення вихідного коду та інших ключових технологічних елементів. Однією з перших пропозицій було поєднати попередні розробки Digital Asset, механізм досягнення консенсусу від Blockstream та OpenBlockchain від IBM. Пізніше ця технологія отримала назву Fabric (рисунок 1.2).

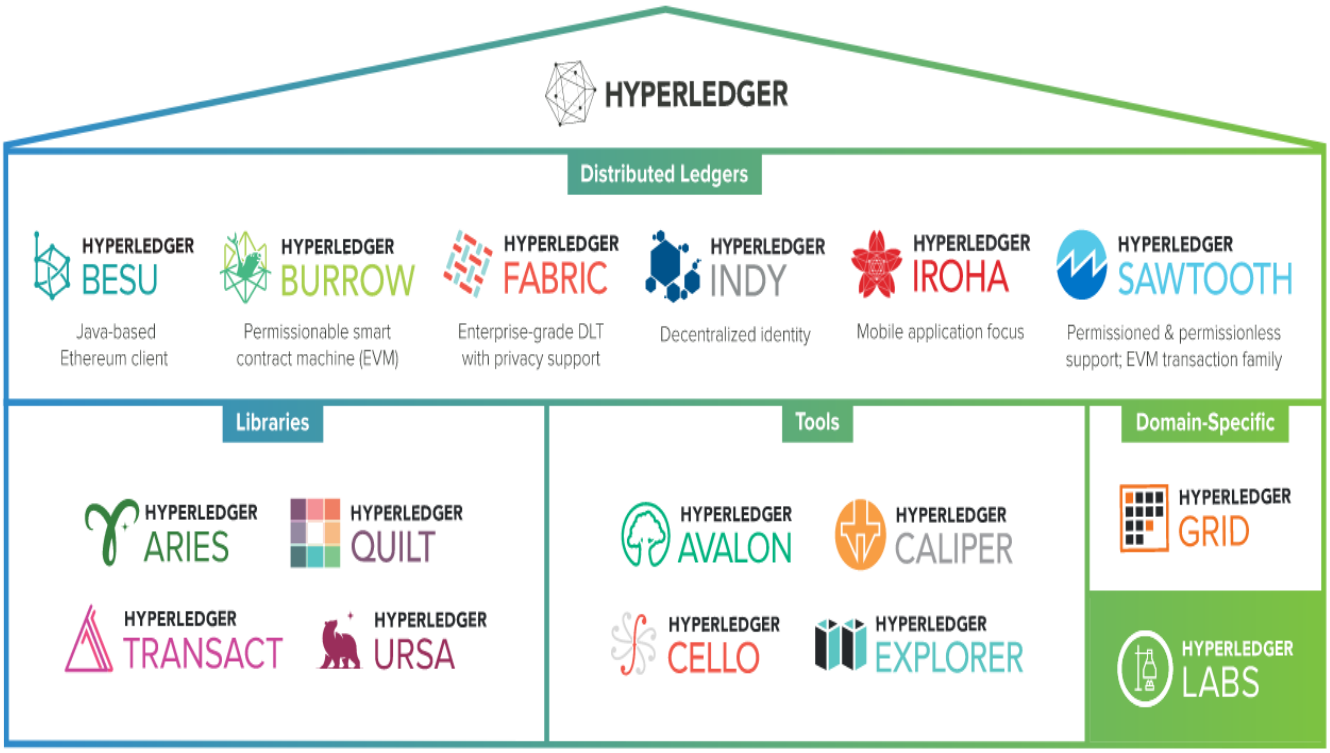


Рисунок 1.2 – Структурна схема платформи Hyperledger Fabric

У травні розпочалася розробка мережі з розподіленим реєстром Sawtooth від Intel.

12 червня 2017 року проект анонсував готову до промислового використання версію Hyperledger Fabric 1.0 який одразу почав набирати популярність на ринку ICO. Того ж місяця London Stock Exchange Group, спільно з IBM, заявила про початок розробки блокчейн-платформи на базі Hyperledger Fabric для випуску цифрових акцій італійських компаній.

У серпні 2017, компанія Oracle приєдналася до консорціуму Hyperledger і оголосила про початок розробки власного хмарного блокчейн-сервісу.

У вересні 2017 Королівський банк Канади почав використовувати Hyperledger для міжбанківських розрахунків з США.

Hyperledger забезпечує такі функціональні можливості для мережі:

– управління ідентичністю – Hyperledger Fabric надає послугу посвідчення членства, яка керує ідентифікаторами користувачів та аутентифікує всіх учасників

мережі, списки контролю доступу можуть використовуватися для надання додаткових рівнів дозволу через авторизацію конкретних мережевих операцій;

– приватність та конфідейційність – Hyperledger дозволяє конкуруючим інтересам бізнесу та будь-яким групам, які потребують приватних, конфіденційних транзакцій, співіснувати в одній і тій же дозволеній мережі, через приватні канали, які представляють із себе обмежені шляхи обміну повідомленнями, які можна використовувати для забезпечення конфіденційності та конфіденційності транзакцій для конкретних підмножин членів мережі, усі дані, включаючи інформацію про трансакції, учасників та каналів, на каналі є невидимими та недоступними для будь-яких членів мережі, яким явно не надано доступ до цього каналу;

– ефективна обробка – Hyperledger Fabric призначає мережеві ролі за типом вузла, для забезпечення одночасності та паралелізму в мережі виконання транзакцій відокремлено від впорядкування та зобов'язань транзакцій. Виконання транзакцій перед їх замовленням дозволяє кожному вузлу однорангових обробляти одночасно кілька транзакцій, це одночасне виконання збільшує ефективність обробки кожного партнера та прискорює доставку транзакцій до служби замовлення;

– chaincode – це визначення програмного забезпечення як активів та інструкція щодо транзакцій для зміни активів, chaincode виконує правила читання або зміни пар ключових значень або іншої інформації бази даних стану, його функції виконуються в базі даних поточного стану і ініціюються через пропозицію транзакцій.

1.3 Платформа для створення додатків Corda

Платформа з відкритим вихідним кодом, яка дозволяє підприємствам здійснювати операції безпосередньо та у суворій конфіденційності за допомогою інтелектуальних контрактів, зменшуючи витрати на операції та ведення записів та оптимізацію бізнес-операції. У світі блокчейн платформ, де всі дані передаються всім учасникам, сувора модель конфіденційності Corda дозволяє бізнесу здійснювати операції без проблем. R3 поставляє два повністю сумісні

дистрибутиви платформи – Corda, безкоштовне завантаження коду, доступного на GitHub та Corda Enterprise, комерційній версії, яка пропонує функції та послуги, які налаштовані для сучасних підприємств [4].

Основоположним об'єктом в концепції є "державний" об'єкт, який є цифровим документом, який реєструє існування, зміст та поточний стан угоди між двома або більше сторонами. Він призначений для обміну лише з тими, хто має законну причину бачити це. Для забезпечення узгодженості глобальної спільної системи, де не всі дані видно всім учасникам, використовують безпечні криптографічні хеші для ідентифікації сторін і даних, а також для зв’язку користувачів з попередніми перетвореннями, щоб забезпечити ланцюги походження. Головну базу даних визначають як сукупність незмінних об'єктів стану.

На відміну від більшості існуючих сьогодні платформ для створення блокчейнів додатків, Corda була побудована з чіткою метою запису та забезпечення ділових угод між торговими партнерами. Як такий, платформа використовує унікальний підхід до розподілу даних та семантики транзакцій, підкреслюючи особливості розподілених баз даних, привабливих для фірм, а саме надійне виконання контрактів в автоматизованому та примусовому виконанні.

На рисунку 1.3 зображений приклад, державний об'єкт, який представляє депозит у розмірі 100 фунтів стерлінгів у комерційному банку, що належить вигаданій судноплавній компанії. Державний об'єкт посилається на код договору, який регулює його переходи, який, ймовірно, буде написаний один раз і повторно використаний величезною кількістю штатів, і може посилатися на хеш тої, що регулює юридичну прозу.

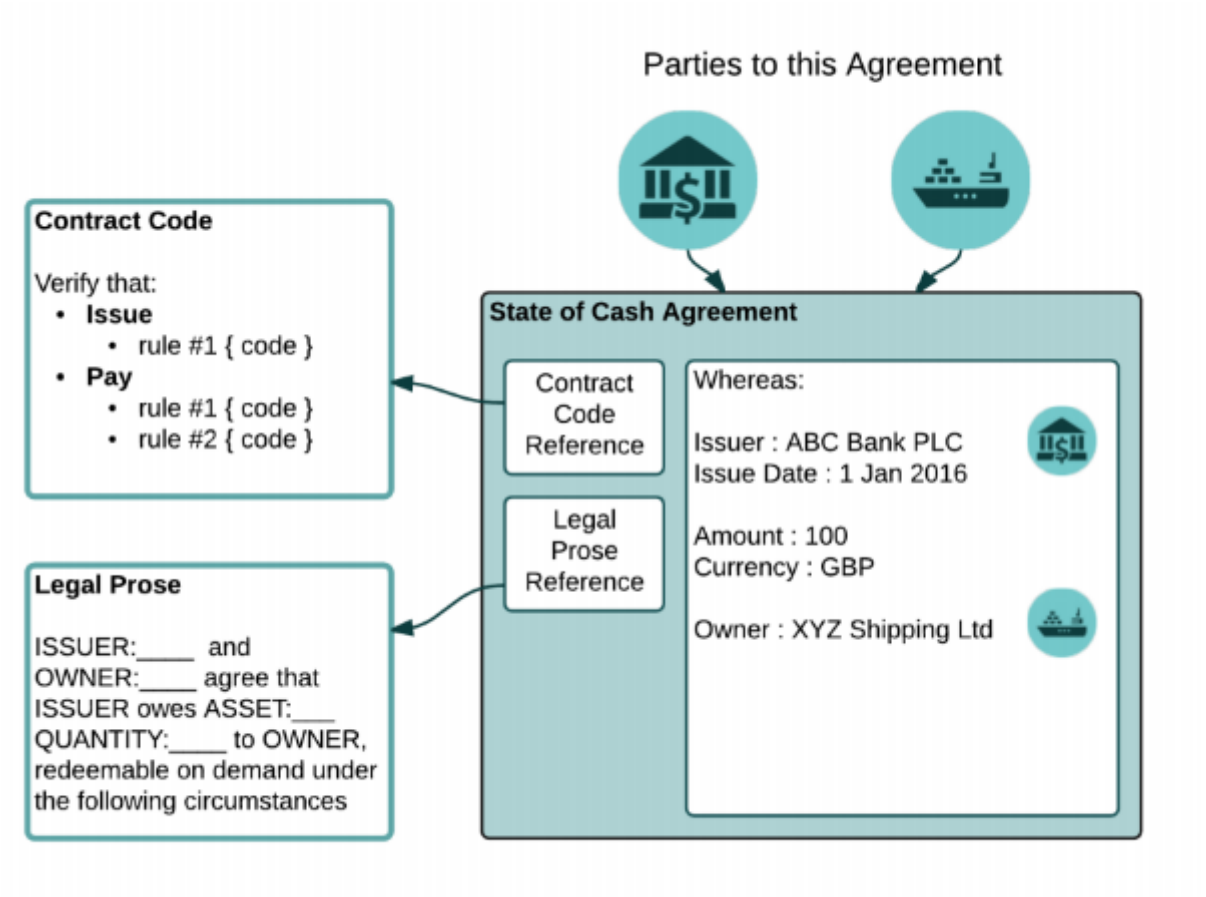


Рисунок 1.3 – Схема роботи децентралізованого банку на платформі Corda

1.4 Платформа для створення додатків Quorum

Quorum – протокол розподіленої бази даних на основі Ethereum, розроблений для забезпечення таких галузей, як фінанси, логістика, роздрібна торгівля, нерухомість, тощо з дозволеною реалізацією Ethereum, що підтримує конфіденційність транзакцій та контрактів [5].

Quorum включає мінімалістичне відгалуженя клієнта Go Ethereum (geth) і, як такий, використовує роботу, яку розпочала спільнота розробників Ethereum.

Основними відмінностями Quorum є:

– конфіденційність транзакцій та контрактів;
– кілька механізмів консенсусу на основі голосування;
– мережеве/однорангове управління дозволами;
– більш висока продуктивність.

На даний момент, Quorum складається з архітектурних компонентів, зображених на рисунку 1.4.

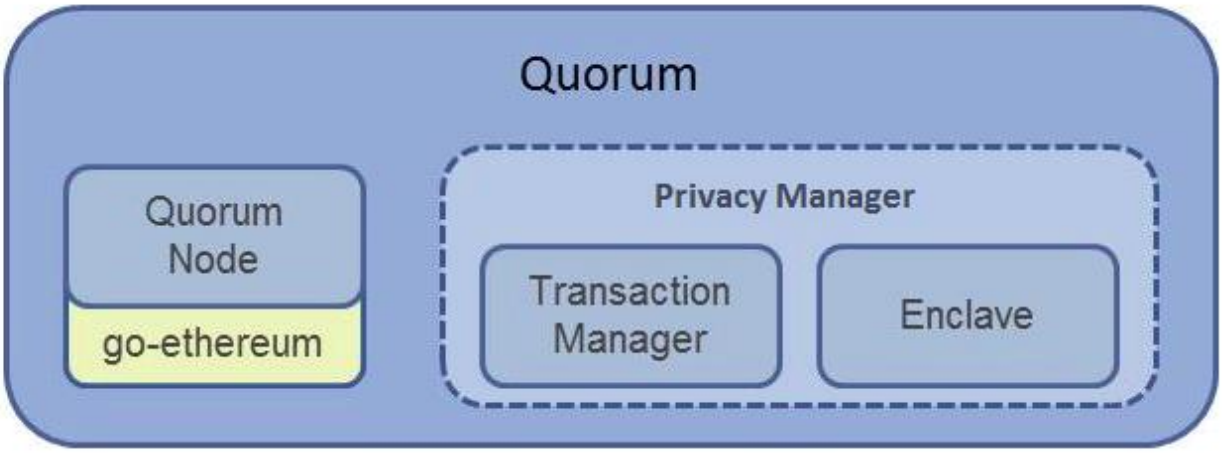


Рисунок 1.4 – Схема логічної архітектури Quorum

2 КОНЦЕПЦІЯ ТА ТЕХНОЛОГІЧНІ ПАРАДИГМИ ПЛАТФОРМИ

Для того, щоб розробити оптимальну платформу, необхідно вирішити ряд проблем, пов'язаних із децентралізованими системами, серед них:

- низька швидкодія, яка обумовлена декількома причинами:
  - повільні алгоритми консенсусу;
  - неоптимальні структури збереження данних;
  - неоптимальні формати для обміну даними;
- завищене використання платформою ресурсів – децентралізованим системам необхідно виконувати більше операцій, тому що уся бізнес логіка знаходиться на клієнтській стороні, , якщо клієнту не вистачає ресурсів, то це може спричинити відмову функціонування додатку;
- незручний доступ до мережі – користувачам необхідно встановлювати додаткове ПО, для того, щоб потрапити до системи.

В даному розділі розглянуто концепції та механізми, які допоможуть усунути ці недоліки.

2.1 Програмна віртуальна машина

Програмна віртуальна машина (PVM), іноді називається віртуальною машиною програми або керованим середовищем виконання (MRE), працює як звичайна програма всередині операційної системи та підтримує єдиний процес. Вона створюється при запуску цього процесу і руйнується, коли він закінчується. Її мета полягає в тому, щоб створити незалежне від платформи середовище програмування, яке відкидує деталі апаратного обладнання або операційної системи і дозволяє програмі виконуватися однаково на будь-якій платформі [6].

Процес VM забезпечує абстракцію високого рівня, мову програмування високого рівня (порівняно з низькорівневою абстракцією ISA системи VM). Програмна віртуальна машина реалізуються за допомогою інтерпретатора;

ефективність, порівнянна зі скомпільованими мовами програмування, може бути досягнута за допомогою компіляції "just in time".

Цей тип VM набув популярності в мові програмування Java (рисунок 2.1), яка реалізована за допомогою віртуальної машини Java. Інші приклади включають віртуальну машину Parrot і .NET Framework, яка працює на VM під назвою " Common Language Runtime". Усі вони можуть служити шаром абстракції для будь-якої мови комп'ютера.

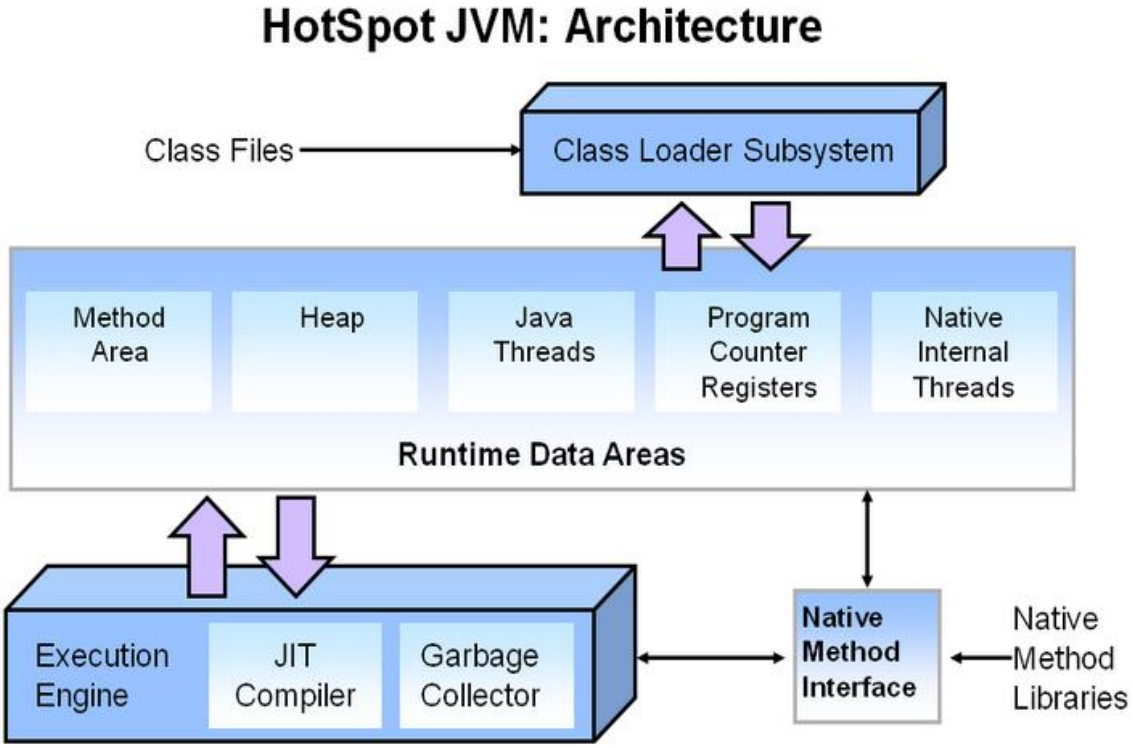


Рисунок 2.1 – Архітектура Hotspot JVM

Особливим випадком віртуальних машин управління є системи, які абстрагуються над механізмами зв'язку (потенційно гетерогенного) комп'ютерного кластеру. Такий тип VM складається не з одного програмного процесу, а з одного процесу на фізичній машині в кластері. Вони розроблені для полегшення завдання програмування багатопоточних додатків, дозволяючи програмісту зосередитися на алгоритмах, а не на механізмах зв'язку, що забезпечуються взаємозв'язком та ОС.

Вони не приховують факту, що спілкування відбувається, і не намагаються представити кластер як єдину машину.

На відміну від інших віртуальних машин, ці системи не забезпечують конкретної мови програмування, але вбудовані в існуючу мову; зазвичай така система забезпечує прив'язку для декількох мов (наприклад, C та Fortran). Прикладами є паралельна віртуальна машина (PVM) та інтерфейс передачі повідомлень (MPI).

Вони не є строго віртуальними машинами, оскільки додатки, що працюють на них, все ще мають доступ до всіх служб ОС і тому не обмежені системою.

Тож, із зазначених переваг можна виділити:

– швидкий час розробки – написання програми не займе багато часу, оскільки розробникам не потрібно створювати функції для окремих платформ та специфікацій;

– безпечний код – керовані режими виконання просувають безпечніший код, знімаючи з розробників частину відповідальності за безпеку та управління обладнанням;

– низькі витрати на розгортання – компонентна архітектура спрощує та швидше розгортає додатки у корпоративному середовищі, що характеризується багатьма платформами, пристроями та застарілими системами;

– більш якісне програмне забезпечення – керований час виконання звільняє розробників зосереджуватися на бізнес-логіці та коді, специфічних для програми, зменшуючи при цьому кількість помилок кодування;

– агностична платформа – завдяки виконанню часу перекладу між вашим додатком та операційною системою наявна можливість писати код один раз, дозволяючи клієнтам запускати програму в декількох системах;

– чистота коду – простота функціоналу дозволяє писати модульний код, який можна переробити в нові програми та нові системи.

Отже, із вище зазначених переваг можна точно сказати, шо віртуальна машина необхідна для реалізації платформи, так як це допоможе в реалізації кросплатформеності та створить безпечну середу для виконання криптографічних операції [7].

2.2 Протокол серіалізації даних

Серіалізація даних – це процес перетворення об'єктів даних (рисунок 2.2), що знаходяться у складних структурах даних, у потік байтів для цілей зберігання, передачі та розповсюдження на фізичних пристроях [8].

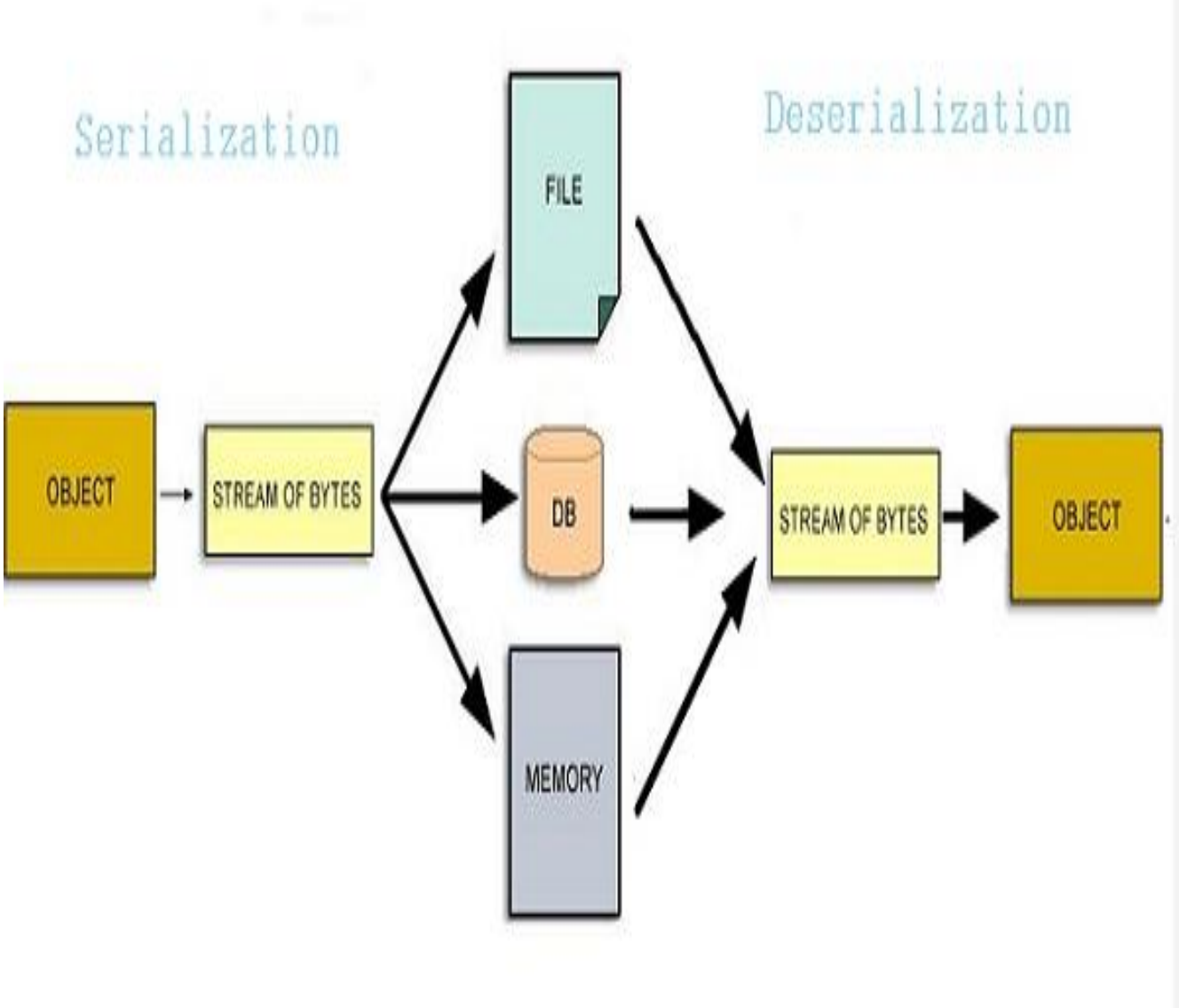


Рисунок 2.2 – Узагальнений алгоритм серіалізації

Комп'ютерні системи можуть відрізнятися за своєю апаратною архітектурою, ОС, адресації. Внутрішні бінарні представлення даних також відповідно змінюються в кожному середовищі. Зберігання та обмін даними між такими різними середовищами вимагає нейтрального для платформи та мови формату даних, який розуміють усі системи.

Після передачі серіалізованих даних з вихідної машини на машину призначення здійснюється зворотний процес створення об'єктів із послідовності байтів, що називається десеріалізацією. Реконструйовані об'єкти – це клони вихідного об'єкта.

Вибір формату серіалізації даних для програми залежить від таких факторів, як складність даних, потреба в читабельності людини, швидкість та обмеження місця для зберігання. XML, JSON, BSON, YAML, MessagePack і Protobuf – деякі часто використовувані формати серіалізації даних.

Комп'ютерні дані зазвичай організовані в структурах даних, таких як масиви, таблиці, дерева, класи. Коли структури даних потрібно зберігати або передавати в інше місце, наприклад, через мережу, вони серіалізуються.

Для простих лінійних даних (число або рядок) нічого робити. Серіалізація стає складною для вкладених структур даних та посилань на об'єкти.

Коли об'єкти вкладені в кілька рівнів, наприклад, у деревах, він згортається на рядбайтів, і достатньо інформації (наприклад, порядок обходу) включається для відновлення початкової структури дерева на стороні призначення. Коли об'єкти із посиланнями вказівників на інші змінні члена серіалізовані, посилаються об'єкти відстежуються та серіалізуються, забезпечуючи, щоб той самий об’єкт не був серіалізований більше одного разу. Однак усі вкладені об'єкти теж повинні бути серіалізованими.

Нарешті, серіалізований потік даних зберігається у послідовності байтів, використовуючи стандартний формат. ISO-8859-1 – популярний формат для 1-байтного представлення англійських символів та цифр. UTF-8 – світовий стандарт кодування багатомовних, математичних та наукових даних; кожен символ може приймати 1-4 байти даних у Unicode.

На рисунку 2.3 зображено приклад серіалізації структури у текстовий формат.

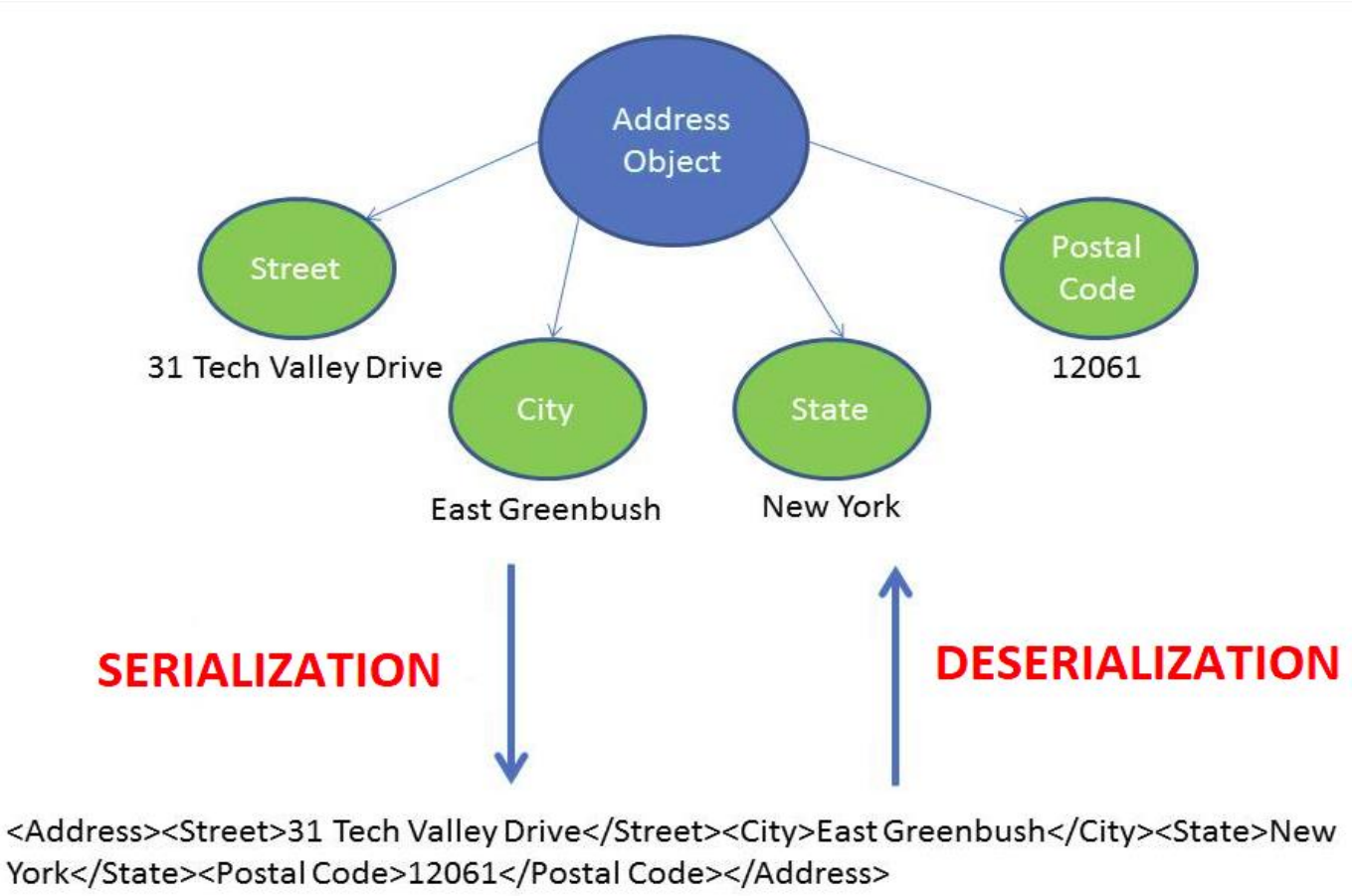


Рисунок 2.3 – Приклад серіалізації даних

Якщо визначати, які з форматів для яких цілей підходять краще, то можна визначити, що:

– швидкість – за даними Uber Engineering, бінарні формати швидші , ніж текстові формати, Google Protobuf має найкращі показники на сьогодні (рисунок 2.4}) а при стиснені даних показники будуть ще більші, проте для додатків, які не потребують інтенсивного використання даних або в режимі реального часу, JSON є кращим із-за читабельності та відсутності схем;

– розмір даних – це стосується фізичного простору в байтах після серіалізації, для невеликих даних стислі дані JSON займають більше місця в порівнянні з бінарними форматами, як Protobuf, і, як правило, бінарні формати завжди займають менше місця;

– корисність – людиночитаємі формати, такі як JSON, природно

переважніші перед бінарними форматами, для складних типів даних бібліотеки міжплатформової серіалізації дозволяють визначати структуру даних на схемах (для текстових форматів) або IDL (для бінарних форматів);

– сумісність або розширюваність – JSON є закритим форматом, XML є середнім за версією схеми, а зворотна сумісність (розширювані схеми) найкраще обробляється Protobuf.

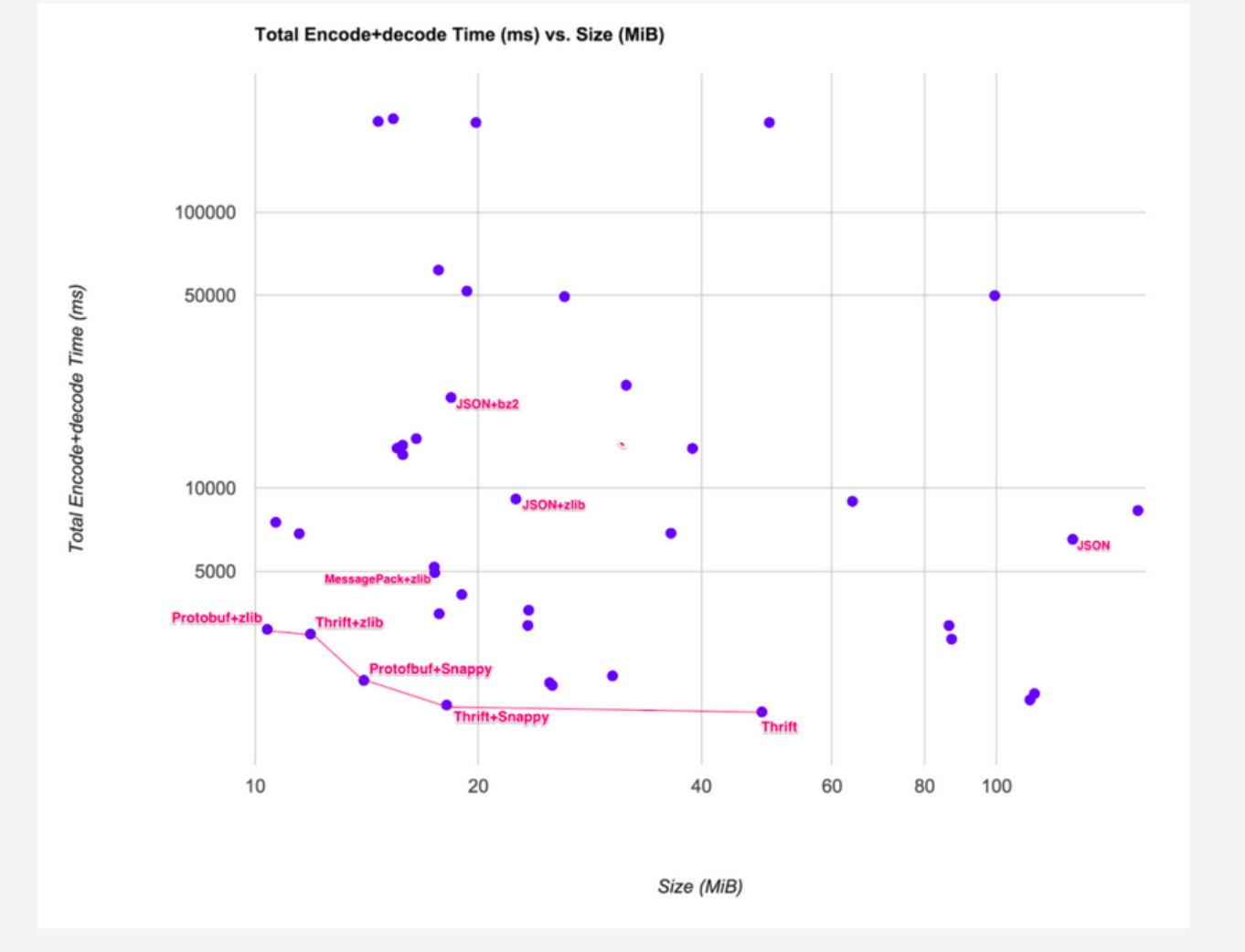


Рисунок 2.4 – Графік відношеня швидкості роботи Protobuf до розмірів вихідних даних

Навіть у Big Data серіалізація стосується перетворення даних у портативні потоки байтів. Але управління схемою – ще одна проблема, яку необхідно брати до уваги. Проблеми з узгодженістю даних, такі як пробіли або неправильні значення даних, можуть бути дуже дорогими, залучаючи великі зусилля щодо очищення даних.

Наступний важливий аспект – можливість легко розділяти та реконструювати дані (наприклад, MapReduce). JSON або XML можуть не працювати належним чином. Apache Hadoop має власний формат серіалізації на основі схеми під назвою Avro (рисунок 2.5), схожий на Protobuf. Схеми Apache також визначаються на основі JSON. Apache Hadoop використовує RPC для розмови з різними компонентами.

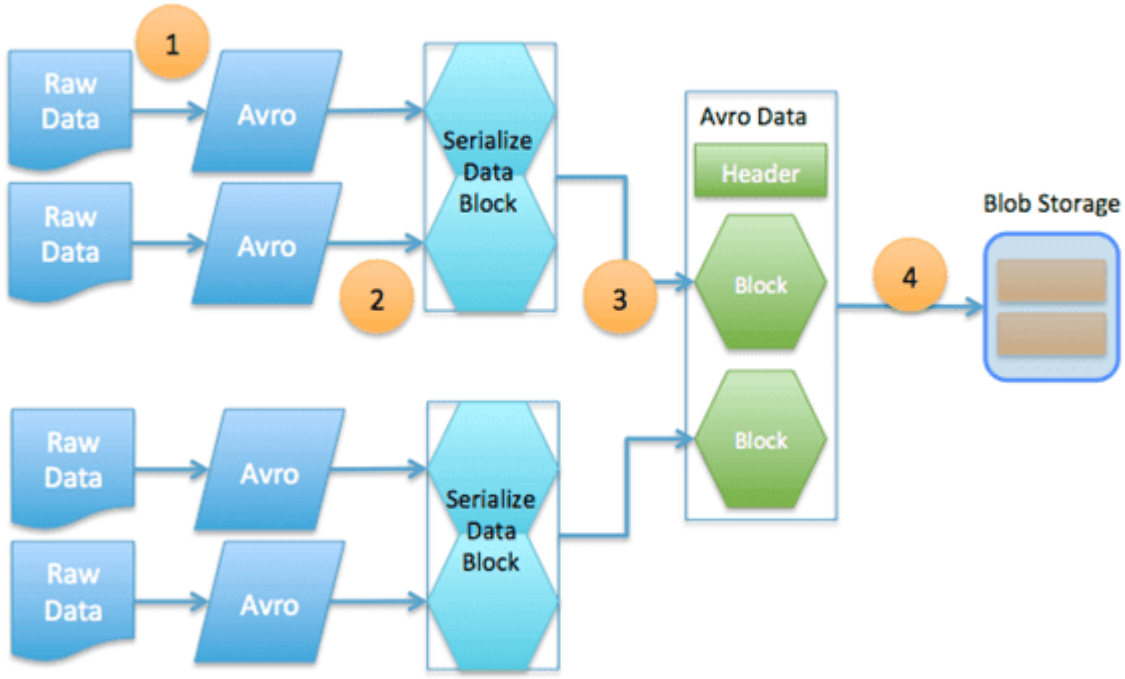


Рисунок 2.5 – Функціональна схема системи сереліалізації даних, побудованої з використанням Avro

Механізм серіалізації є важливою частиною платформи, оскільки не тільки дозволяє досягти агностичності, але і збільшити швидкодію та зменшити об’єм сховища даних.

2.3 Збереження даних

На відміну від клієнт-серверної архітектури, де існує централізоване сховище даних, децентралізовані мережі не можуть так працювати.

Тому дані зберігаються на кожній машині, яка підключена до централізованої мережі.

Що стосується розподіленої блокчейн-мережі, кожен учасник мережі підтримує, затверджує та оновлює нові записи.

Структура даних, в якій вони зберігаються, називається blockchain (рисунок 2.5).

Рисунок 2.6 – Схема blockchain структури даних

За задумом, блокчейн стійкий до модифікації даних. Це "відкрита, розподілена книга, яка може ефективно і оперативно фіксувати транзакції між двома сторонами". Для використання в якості розподіленого сховища, блокчейн, як правило, управляється одноранговою мережею, яка колективно дотримується протоколу для міжвузлового зв'язку та перевірки нових блоків. Після запису дані в будь-якомуданому блоці не можуть бути змінені заднім числом без зміни всіх наступних блоків, що вимагає консенсусу більшості мережі.

Проте, із часом, об'єм даних буде рости, це будет спричиняти зповілненя пошуку та оновленя цих даних у кожного клієнта мережі. Саме тому, для оптимального представлення даних використовують структуру даних під назвою "дерево Меркле" (рисунок 2.6).

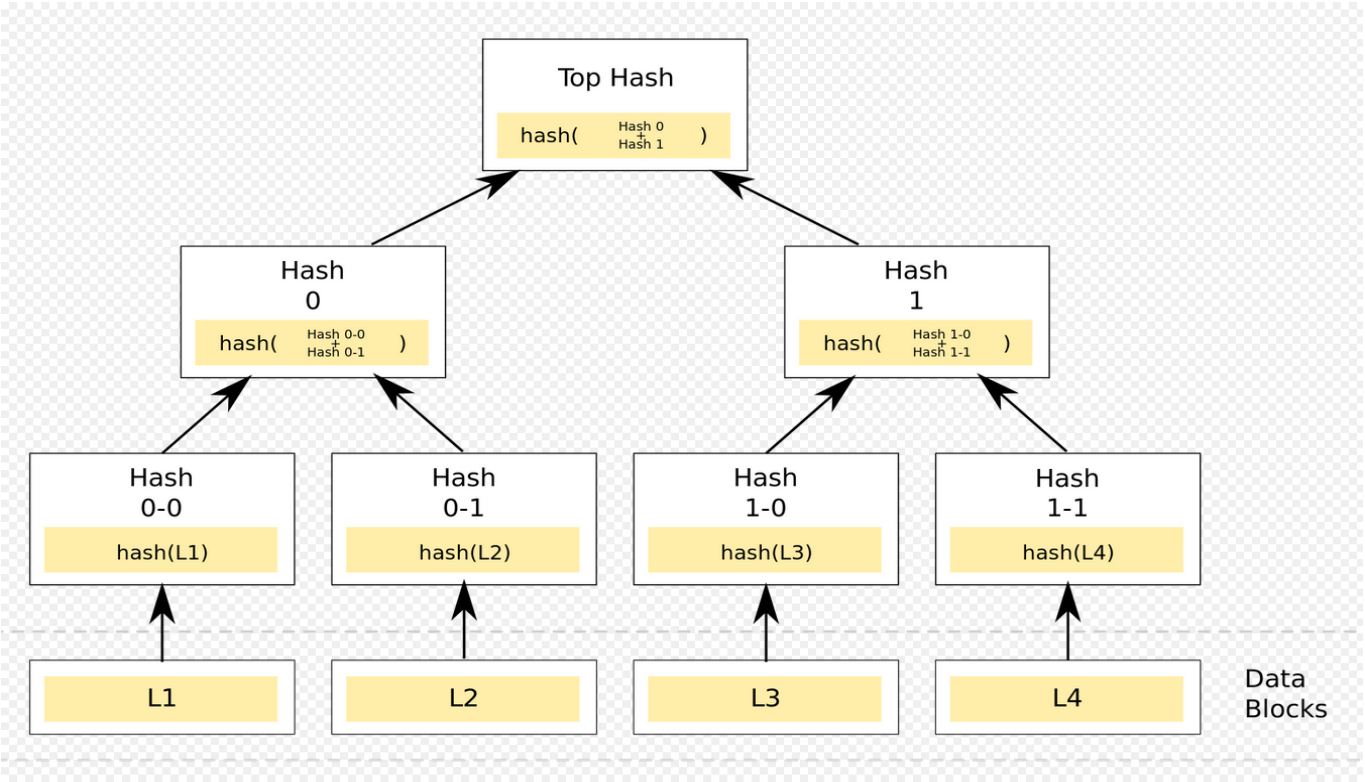


Рисунок 2.7 – Схема blockchain структури даних

Меркелеве дерево - це структура, яка дозволяє ефективно і безпечно перевіряти вміст у великому масиві даних. Ця структура допомагає перевірити узгодженість та зміст даних.

Дерево Меркле підсумовує всі транзакції в блоці, створюючи цифровий відбиток пальця всього набору транзакцій, тим самим дозволяючи користувачеві перевірити, включена чи ні транзакція в блок.

Дерева Меркле створюються багаторазово хешуючими парами вузлів, поки не залишиться лише один хеш (цей хеш називається кореневим хешом, або корінь Меркле). Вони будуються знизу вгору, з хешей окремих транзакцій (відомих як ідентифікатори транзакцій).

Кожен вузол листів – це хеш даних транзакцій, а кожен нелистовий вузол – хеш попередніх хешів. Мерклеві дерева є двійковими і тому потребують парної кількості листових вузлів. Якщо кількість транзакцій непарна, останній хеш дублюється один раз, створити парну кількість вузлів листів.

В даній роботі не будуть детально розглядатися підходи до збереження даних,

так як це добре досліджена тема.

2.4 Протокол консенсусу

Принциповою проблемою розподілених обчислювальних і багатоагентних систем є досягнення загальної надійності системи за наявності низки несправних процесів. часто вимагає від процесів узгодження певної вартості даних, яка потрібна під час обчислення. Приклади застосувань консенсусу включають, чи слід здійснювати транзакцію в базі даних, узгоджуючи особу лідера, центральну машинну реплікацію (генезис) та атомарну трансляцію.

Однак у процесі застосування технології blockchain виникає багато проблем і питань, серед яких велике питання – як розробити відповідний протокол консенсусу.

Консенсус блокчейна полягає в тому, що всі вузли підтримують однакове розподілене сховище. У традиційній архітектурі програмного забезпечення консенсус навряд чи є проблемою через існування центрального сервера, отже, інші вузли потрібно лише узгодити з сервером. Однак у розподіленій мережі, такій як блокчейн, кожен вузол є і хостом, і сервером, і йому потрібно обмінюватися інформацією з іншими вузлами, щоб досягти консенсусу. Іноді деякі вузли будуть в режимі онлайн або в режимі офлайн, а також з’являться деякі шкідливі вузли, що серйозно вплине на систему або знищить процес консенсусу. Тому відмінний консенсус-протокол може допустити виникнення цих явищ і мінімізувати шкоду, щоб не вплинути на кінцевий результат консенсусу.

Крім того, прийнятий системою протокол консенсусу також повинен бути придатним для типу блокчейн, який використовується системою. Загалом, можна виділити три основні типи blockchain: загальний блокчейн, консорціумний блокчейн та приватний блокчейн.

Кожен тип blockchain має різні сценарії застосування. Таким чином, прийнятий протокол консенсусу повинен відповідати вимогам конкретного сценарію застосування.

2.4.1 Алгоритм Proof of Work

PoW вибирає один вузол, щоб створити новий блок у кожному раунді консенсусу шляхом конкуренції з обчислювальної потужності. У змаганні вузлам-учасникам потрібно розв'язати криптографічну проблему. Вузол, який першим вирішить задачі, може мати право створити новий блок. Потік створення блоку в PoW представлений на рисунку 2.8.

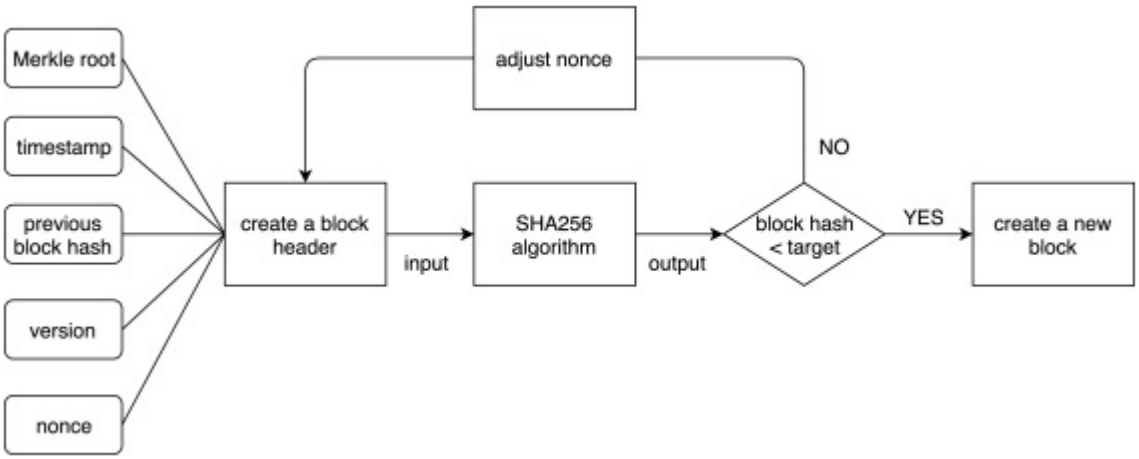


Рисунок 2.8 – Схема роботи PoW алгоритму

Розв'язати проблему, яку представляє PoW дуже важко. Вузлам потрібно постійно коригувати значення nonce, щоб отримати правильну відповідь, що вимагає великої обчислювальної потужності. Зловмисник може скинути один блок в ланцюжку, але в міру збільшення дійсних блоків ланцюга також накопичується навантаження, тому для скидання довгого ланцюга потрібна величезна обчислювальна потужність. PoW належить до протоколів консенсусу ймовірнісно-кінцевих, оскільки він гарантує можливу послідовність.

2.4.2 Алгоритм Proof of Stake

У PoS вибір кожного раунду вузла, який створює новий блок, залежить від утримуваного кола, а не обчислювальної потужності. Хоча вузлам все-таки потрібно

вирішити задачу SHA256, представлену формулою 2.1:

$$SHA512(timestamp,previoushash\ldots) < target * coin \quad (2.1)$$

Від PoW відмінність полягає в тому, що вузлам не потрібно багато разів коригувати нуль, натомість ключовим для вирішення цієї задачі є кількість ставок (монет). Отже, PoS – це енергозберігаючий консенсус-протокол, який використовує спосіб стимулювання внутрішньої валюти, а не витрачає багато обчислювальної сили для досягнення консенсусу. Схема алгоритму виконання PoS показана на рисунку 2.9.

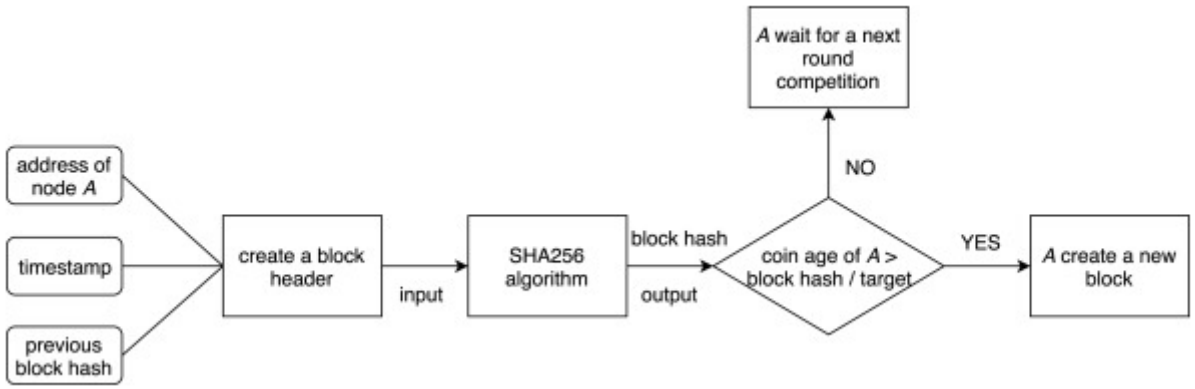


Рисунок 2.9 – Схема роботи PoS алгоритму

Як і PoW, PoS також є протоколом консенсусу ймовірнісно-кінцевих. PPcoin була першою криптовалют, яка застосувала PoS до блокчейн. У PPcoin, крім розміру ставки, вік монети також вводиться для вирішення задачі PoS. Наприклад, якщо ви тримаєте 10 монет протягом 20 днів, то ваш вік монет – 200. Після того, як вузол створить новий блок, його вік монет очиститься до 0.

Крім PPcoin, багато криптовалют використовують PoS, наприклад , Nxt, Odour.

2.4.3 Алгоритм Delegated Proof of Stake

Принцип DPoS – дозволити вузлам, які мають пакет голосів, обирати верифікаторів блоків (тобто, творців блоків). Цей спосіб голосування змушує

зацікавлені сторони надавати право створювати блоки для делегатів, яких вони підтримують, а не створювати блоки, таким чином зменшуючи їх обчислювальне енергоспоживання до нуля. Алгоритм роботи DPoS на рисунку 2.10.

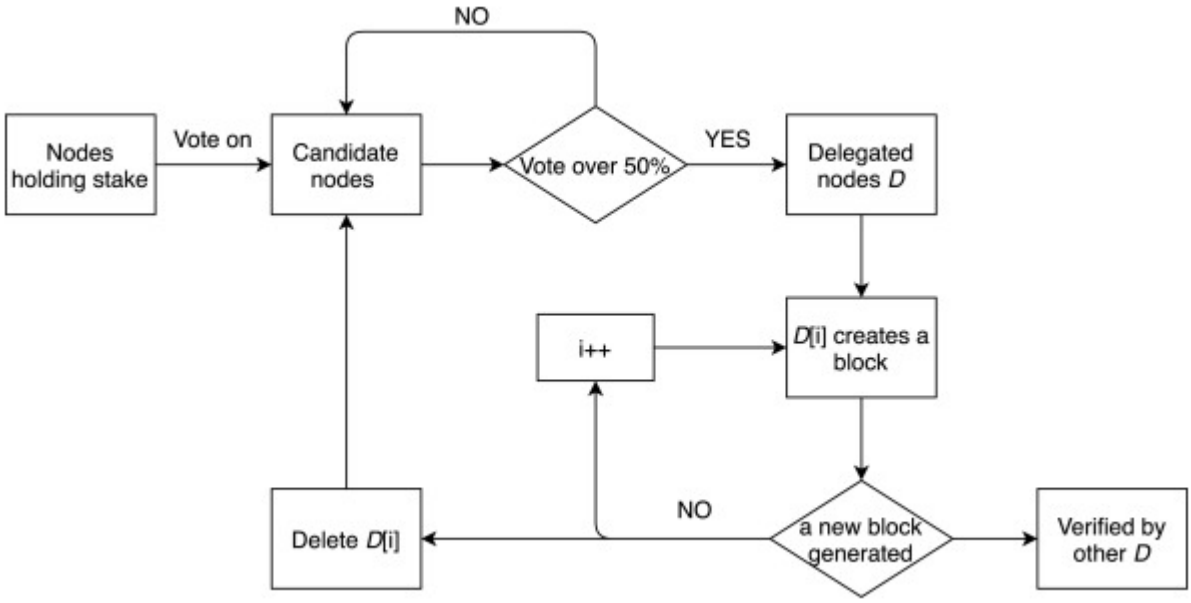


Рисунок 2.10 – Схема роботи DPoS алгоритму

DPoS – це як парламентська система, якщо делегати не зможуть генерувати блоки в свою чергу, будуть звільнені, а зацікавлені сторони виберуть нові вузли для їх заміни. DPoS максимально використовує голоси акціонерів для досягнення консенсусу справедливим та демократичним способом. Порівняно з PoW та PoS, DPoS – консенсус-протокол з низькою вартістю та високою ефективністю. Існують також деякі криптовалюти, які приймають DPoS, такі як BitShares, EOS. Нова версія EOS перетворила DPoS на BFT-DPoS (Byzantine Fault Tolerance-DPoS).

2.4.4 Алгоритм Practical Byzantine Fault Tolerance

PBFT – це візантійський протокол допуску відмов з низькою складністю алгоритму та високою практичністю в розподілених системах. PBFT містить п’ять етапів: запит, попередня підготовка, підготовка, виконання та відповідь. На рисунку 2.11 зображено, як працює PBFT.

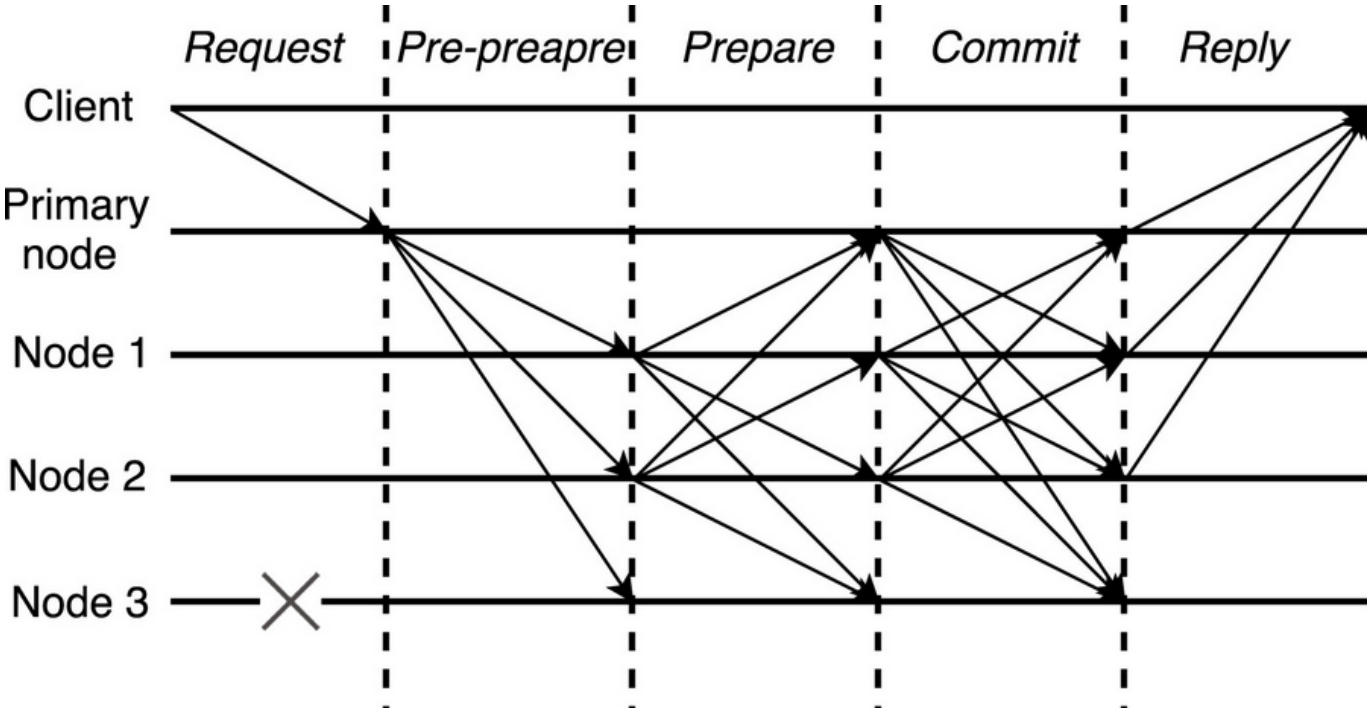


Рисунок 2.11 – Схема роботи PBFT алгоритму

Первинний вузол пересилає повідомлення, надіслане клієнтом, на три інші вузли. У випадку збою 3 вузла одне повідомлення проходить через п'ять фаз, щоб досягти консенсусу серед цих вузлів. Нарешті, ці вузли відповідають клієнту для завершення консенсусу.

PBFT гарантує, що вузли підтримують загальний стан і вживають послідовних дій у кожному раунді консенсусу. PBFT досягає мети міцної послідовності, таким чином, це протокол консенсусу абсолютної остаточності.

Новий протокол під назвою Stellar – це поліпшення PBFT. Stellar приймає протокол FBA (Федеративна візантійська угода), в якому вузли можуть обрати федерацію, якій вони довіряють, щоб провести процес консенсусу.

2.4.4 Порівняння алгоритмів консенсусу

Для того, щоб обрати, який з алгоритмів буде найкраще підходити для платформи, необхідно спочатку порівняти їх за певними критеріями (таблиця 2.1).

Таблиця 2.1 – Порівняння протоколів консенсусу

Властивості	PoW	PoS	DPoS	PBFT
Тип	ймовірнісно-кінцевий	ймовірнісно-кінцевий	ймовірнісно-кінцевий	абсолютно-кінцевий
Відмовостійкість	50%	50%	50%	33%
Споживання енергії	Велике	Низьке	Низьке	Відсутне
Масштабованість	Добре	Добре	Добре	Погано
Застосування	Публічне	Публічне	Публічне	Приватне

Як можна побачити, найоптимальнішого алгоритму для даної платформи не існує, проте PoS добре показує себе у відкритих системах, в той час як BPFT може себе показати у приватних блокчейнах.

2.5 Механізм обробки подій

Цикл подій – це програмана модель дизайну, яка чекає і пересилає події або повідомлення в програмі [9]. Цикл подій працює (рисунок 2.12), надсилаючи запит до якогось внутрішнього або зовнішнього "постачальника подій" (який, як правило, блокує запит, поки подія не надійшла), а потім викликає відповідного обробника подій ("розсилає подію").

Event-Driven підхід, може відокремлювати потоки від з'єднань, які використовують лише потоки для подій на конкретних зворотних викликах або обробниках.

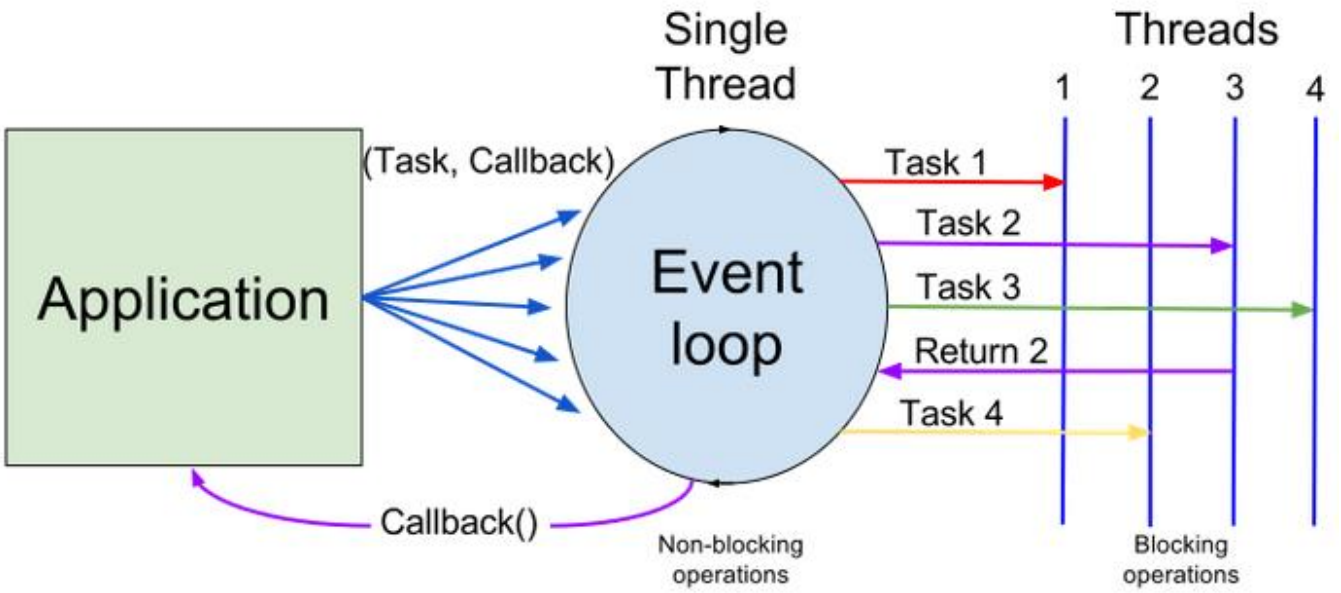


Рисунок 2.12 – Схема роботи Event Loop на чотирьох потоках

Керована подіями архітектура складається з творців подій та споживачів подій. Творець, який є джерелом події, знає лише, що подія сталася. Споживачі – це суб'єкти, які повинні знати, що відбулася подія. Вони можуть бути залучені до опрацювання події або можуть просто вплинути на подію.

Даний підхід дозволить максимально оптимально використовувати ресурси, тим самим підвищуючи швидкість встановлення консенсусу в системі.

2.6 Механізм зовнішнього доступу

Для того, щоб створювати додатки для користувачів, необхідний механізм, який надаватиме змогу отримувати данні системи ззовні, таким чином буде можливість підключати різноманітні клієнтські додатки із мінімальними зусиллями.

2.6.1 Технологія Remote Procedure Call

Віддалений виклик процедури (RPC) – це коли комп'ютерна програма

спричиняє виконання процедури (підпрограми) в іншому адресному просторі (зазвичай на іншому комп’ютері в спільній мережі), який кодується так, ніби це був звичайний (локальний) виклик процедури [10]. Тобто, програміст пише по суті той самий код, чи підпрограма є локальною для виконавчої програми, або віддаленою. Це форма взаємодії клієнт-сервер (абонент – клієнт, виконавець – сервер), що зазвичай реалізується через систему передачі повідомлень запит-відповідь (рисунок 2.13). У об'єктно-орієнтованій парадигмі програмування виклики RPC представлені вилученим методом виклику (RMI).

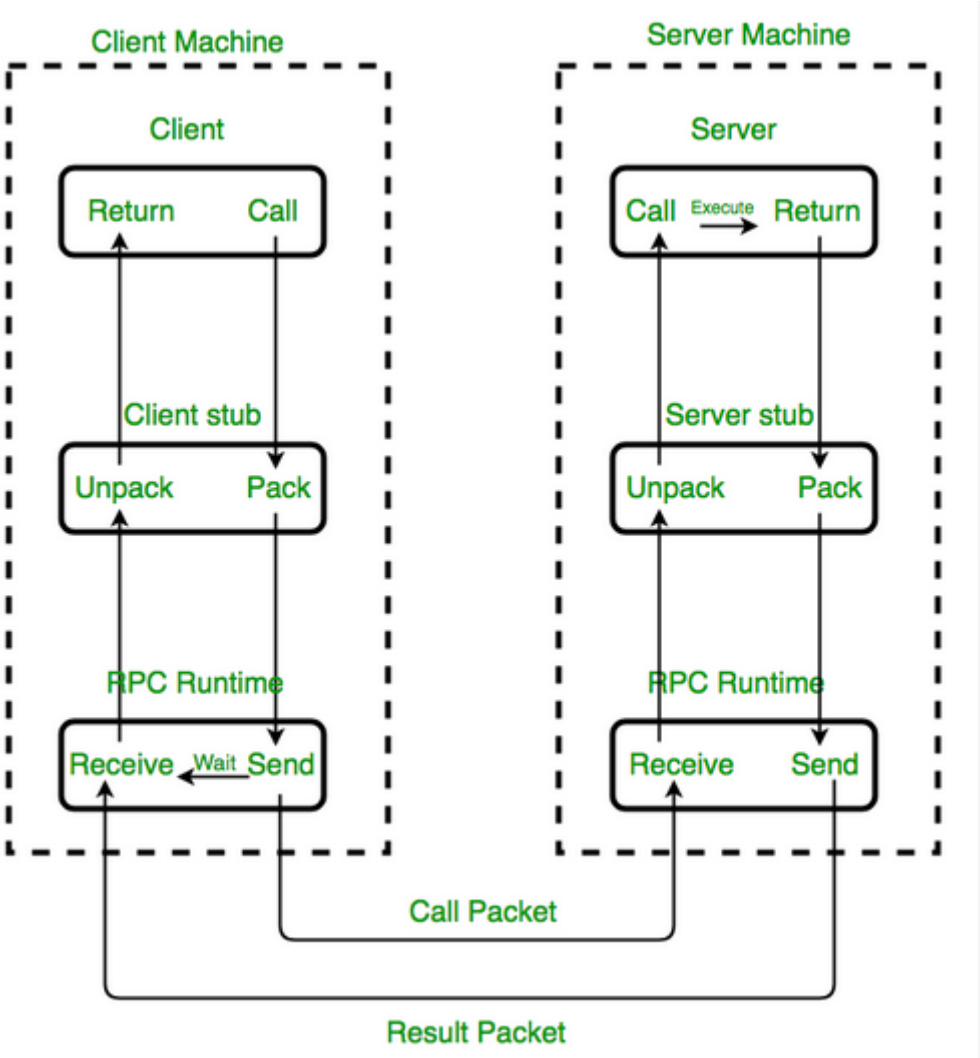


Рисунок 2.13 – Схема роботи виклику за допомогою RPC

Модель RPC передбачає рівень прозорості місцеположення, а саме те, що процедури виклику значною мірою однакові, незалежно від того, локальні вони або

віддалені, але зазвичай вони не є ідентичними, тому локальні виклики можна відрізнити від віддалених. Віддалені виклики зазвичай на порядок повільніші і менш надійні, ніж локальні, тому важливо розрізняти їх.

RPC – це форма міжпроцесорного зв'язку (IPC), оскільки різні процеси мають різні адресні простори: якщо на одній хост-машині вони мають чіткі віртуальні адресні простори, хоча фізичний простір адрес однаковий; якщо вони знаходяться на різних хостах, фізичний адресний простір відрізняється.

2.6.2 RESTful сервіс

REST – архітектурний стиль програмного забезпечення, який визначає набір обмежень, які будуть використані для створення веб-служб(рисунок 2.14) [11].

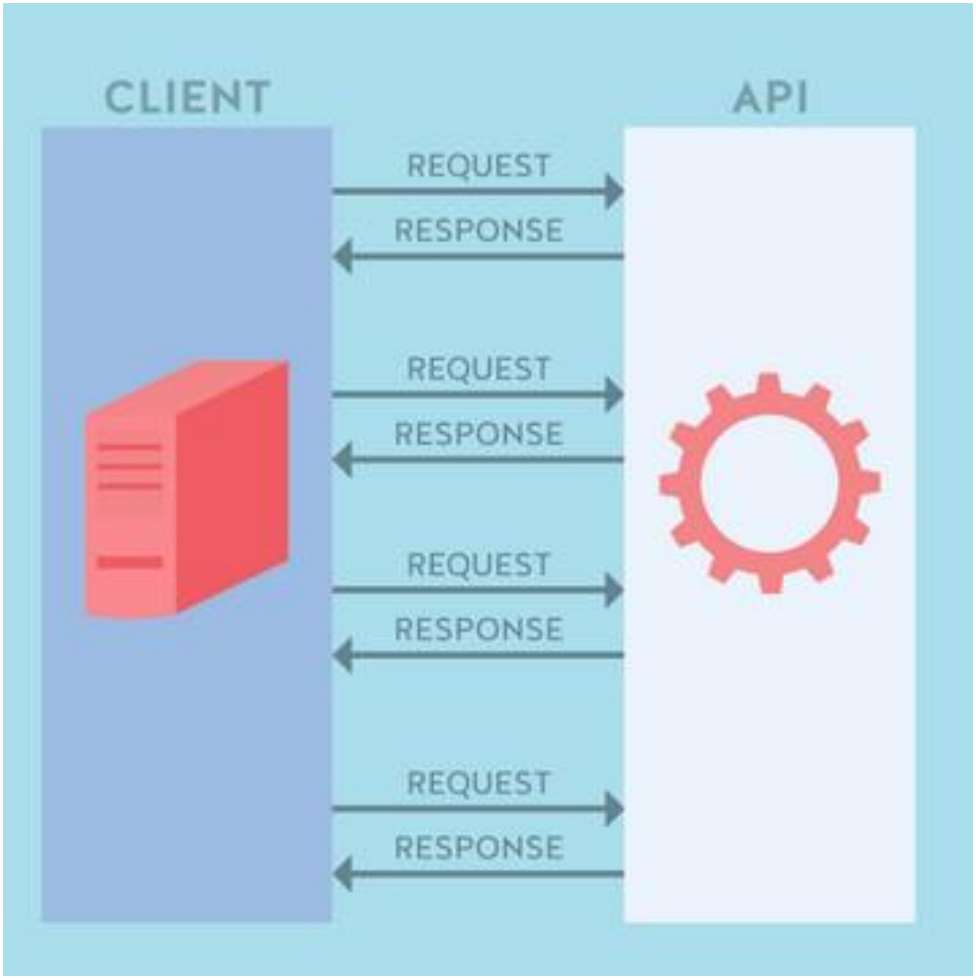


Рисунок 2.13 – Принцип роботи Restful API

"Веб-ресурси" вперше були визначені у Всесвітній павутині як документи або файли, визначені за їх URL-адресами. Однак сьогодні вони мають набагато більш загальне та абстрактне визначення, яке охоплює будь-яку річ чи сутність, які можна будь-яким чином ідентифікувати, назвати, звертатись чи обробляти в Інтернеті. У веб-службі RESTful запити, що надсилаються до URI ресурсу, викликають відповідь з корисним навантаженням, відформатованим у HTML, XML, JSON чи іншому форматі. Відповідь може підтвердити, що деякі зміни були внесені до збереженого ресурсу, і відповідь може забезпечити гіпертекстові посилання на інші пов'язані ресурси або набори ресурсів. Якщо використовується HTTP, як це найбільш часто, доступними операціями (методами HTTP) є GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS та TRACE.

3. РЕАЛІЗАЦІЯ ТА ТЕХНІЧНА ОПТИМІЗАЦІЯ

3.1 Формат серіалізації Dragnit

Dragnit – це бінарний формат на основі схем для ефективної сереалізації дерев даних. Він натхненний Google Protobuf [12], але простіший, проте має більш компактне кодування та кращу підтримку додаткових полів.

3.1.1 Особливості алгоритму

Основні особливості алгоритму:

– ефективна серіалізація загальних значень – кодування змінної довжини використовується для числових значень, де малі значення займають менше місця;

– ефективна серіалізація складних об'єктів – структурна функція підтримує вкладені об'єкти з нульовою серіалізацією накладних витрат;

– виявлення наявності необов'язкових полів – це неможливо в protobuf, особливо для повторних полів;

– лінійна серіалізація (читання та запис) – це операції одноканального сканування, тому вони ефективні в кеш-пам'яті та мають гарантовану часову складність;

– зворотна сумісність – нові версії схеми все ще можуть читати старі дані;

– сумісність вперед – старі версії схеми можуть читати нові дані, якщо копія нової схеми в комплекті з даними (нова схема дозволяє декодеру пропускати через невідомі поля);

– проста реалізація – API дуже мінімальний.

Серіалізатор підтримує такі типи:

– bool – значення, яке зберігає істинне або хибне, буде використовувати 1 байт;

– byte – беззнакове 8-бітове ціле число, очевидно, використовується 1 байт;

– int – ціле 32-бітне ціле значення, що зберігається з використанням кодування; змінної довжини, оптимізованого для зберігання чисел з невеликою величиною, буде використано не більше 5 байт;

– uint – ціле 32-бітне ціле значення, що зберігається з використанням кодування; змінної довжини, оптимізованого для зберігання малих невід’ємних чисел,уде використано не більше 5 байт;

– float – 32-бітове число з плаваючою комою, зазвичай використовує 4 байти; але для значення нуля використовує 1 байт;

– string – рядок з нульовим завершенням UTF-8, буде використаний принаймні 1 байт;

– T [] – будь-який тип може бути перетворений у масив, використовуючи суфікс [];

– enum – uint з обмеженим набором значень, які ідентифікуються за назвою, нові поля можна додавати до будь-якого повідомлення, зберігаючи зворотну сумісність;

– struct – складене значення з фіксованим набором полів, які завжди обов'язкові та записані в порядку, нові поля не можуть бути додані до структури, коли ця структура використовується;

– message – складене значення з необов’язковими полями, нові поля можна додавати до будь-якого повідомлення, зберігаючи зворотну сумісність.

На лістингу 3.1 зображена проста схема для Drargnit.

```
enum Shape {
    BIG = 0;
    MEDIUM = 1;
    SMALL = 2;
```

```
}
struct Color {
    byte r;
    byte g;
    byte b;
}
message Example {
    uint clientId = 1;
    Shape shape = 2;
    Color[] colors = 3;
}
```

Лістинг 3.1 – Проста схема серіалізації

3.1.2 Алгоритм роботи

Уданому розділі наведені приклади кодування кожної структури.

Для початку, нехай буде обрано просте повідомлення (лістинг 3.2 що складається із одного натурального числа у діапазоні (0-255):

```
message ChangeBitOperation {
    uint bitPosition = 1;
}
```

Лістинг 3.2 – Схема простого повідомленя із натуральним числом у якості даних

Якщо встановити значення, наприклад, в 1, то після серіалізації можна побачити наступну картирну (рисунок 3.1).

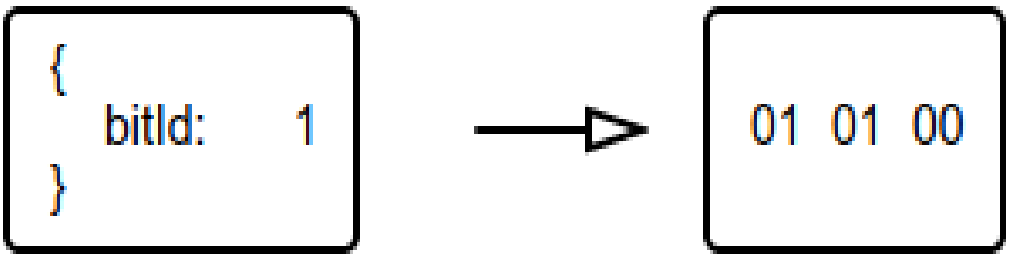


Рисунок 3.1 – Приклад роботи алгоритму серіалізації із натуральним числом

Щоб зрозуміти, як серіалізація працює, спочатку потрібно зрозуміти метод "varint".

Varint – це метод серіалізації цілих чисел за допомогою одного або декількох байтів. Менші числа займають меншу кількість байтів.

Кожен байт у varint, крім останнього байта, має найзначніший набір бітів (msb) – це вказує на те, що надходять ще байти.

Молодші 7 біт кожного байта використовуються для зберігання представленого ними комплексупредставлення числа в групах по 7 біт, найменш значущої групи спочатку.

Строкові рядки кодуються схожим методом, проте для кожного символу використовуеться ключ, ключем є номер поля у повідомлені.

Наприклад, нехай буде серіалізовано рядок "test" за схемою, наведеною у лістингу 3.3.

```
message StringOperation {
        string str = 1;
}
```

Лістинг 3.3 – Схема простого повідомленя із строкою у якості даних

На виході буде отримано значення, яке зображено на рисунку 3.2.

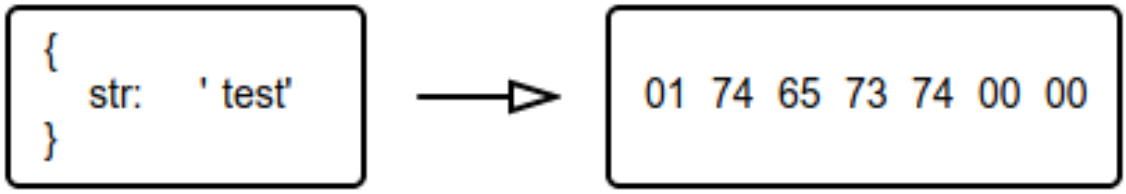


Рисунок 3.2 – Приклад роботи алгоритму серіалізації на строкових даних

Як можна побачити, перше значення, 01, і є ключ, тобто номер рядка у повідомлені, інші байти – саме слово. Для більше комплексних данних, завдяки оптимізаціям, можна досягнути значних оптимізацій використання ресурсів. Нехай, буде задана більш складна схема, приведена на лістингу 3.4.

```
enum Shape {
  BIG = 0;
  MEDIUM = 1;
  SMALL = 2;
}
struct Color {
  byte r;
  byte g;
  byte b;
}
message Example {
  uint clientId = 1;
  Shape shape = 2;
  Color[] colors = 3;
}
```

Лістинг 3.4 – Схема для сереалізації із користувацькими типами

В даній схемі можна побачити як і користувацькі типи, як struct та enum, так і вкладені данні у повідомлення.

Задана схема будет сереалізована даними, представленими на лістингу 3.5.

```
{
    "clientId": 21,
    "shape": "MEDIUM",
    "colors": [
      {
        "r": 122,
        "g": 127,
        "b": 0
      }
    ]
}
```

Лістинг 3.5 – Дані для сереалізації

Після серіалізації буде отримано такий потік байтів (рисунок 3.3)

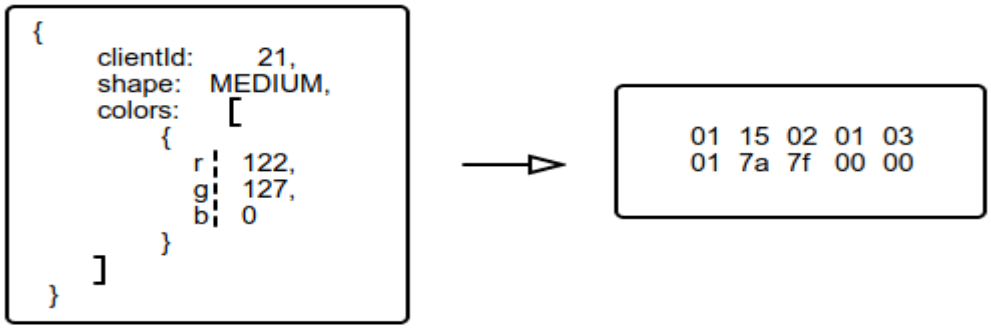


Рисунок 3.3 – Приклад роботи алгоритму серіалізації на складних структурах даних

Як можна побачити, алгоритм непогано оптимізує вкладені структури данних, зберігши по 1 байту на кожне одиничне значення, тобо 5 байтів усього.

Алгоритм також підтримує опціональні поля та обробку помилок у схемі. Це дозволяє двум конкуруючим схемам впізнавати один одну навіть при незначних відмінностях у схемах.

Для прикладу, у даних для ініціалізації, розглянутих вище (лістинг 3.4), зроблена помилка, замість поля "clientId" представлено "clientID", результат сереалізації можна побачити на рисунку 3.4.

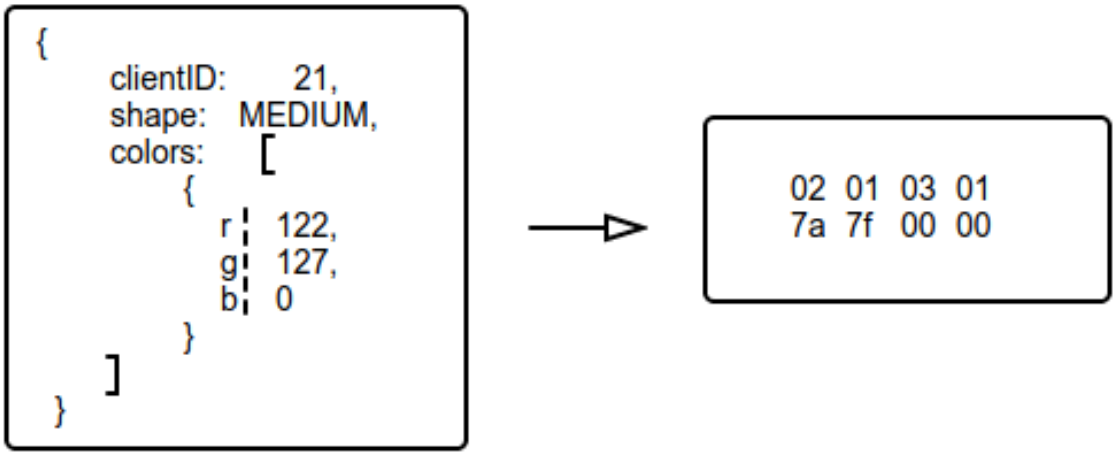


Рисунок 3.4 – Приклад роботи алгоритму серіалізації при помилках у даних

Як можна побачити, алгоритм просто проігнорував невалідне поле і просто закодував правильні дані, тим самим дозволяючи вирішувати конфлікти між схемами та невалідними пакетами даних.

3.1.3 Реалізація алгоритму

Даний алгоритм реалізований мовою Rust, так як вона ідеально підходить для реалізації низькорівневих речей, без великих затрат часу.

Rust – це системна мультипарадигмена мова програмування, орієнтована на безпеку, особливо безпечну багатопоточність.

Rust синтаксично схожий на C++, але розроблений для забезпечення кращої

безпеки пам’яті при збереженні високої продуктивності.

Дана мова була обрана із декількох переваг [13]. За задумом, код Rust не може мати загублені покажчики, переповнення буфера чи цілу низку інших помилок пам'яті. Будь-який код, який би спричинив це буквально, неможливо скомпілювати. Мова цього не дозволяє.

Найголовніше, що Rust досягає всіх цих гарантій безпеки пам’яті під час компіляції. Немає режиму виконання, що робить кінцевий код таким же швидким, як C/C++, проте він набагато безпечніший.

Rust дозволяє писати високопродуктивний код, проте не задумуватися про технічні моменти реалізації потоків чи управління пам’яті. Це дозволяє проектувати програмне забезпечення на іншому рівні абстракції.

Також, ще однією причиною для вибору, була дуже гнучка модульна система, яку підтримує інфраструктура Rust.

Для більшої зручності, алгоритм був реалізований в якості модуля бібліотеки, таким чинов він не був кросзалежним від інших модулів, його можна легко збирати та тестувати.

Якщо представити код бібліотеки простою структурною діагармаю, то він буде виглядати, як зображено на рисунку 3.5.

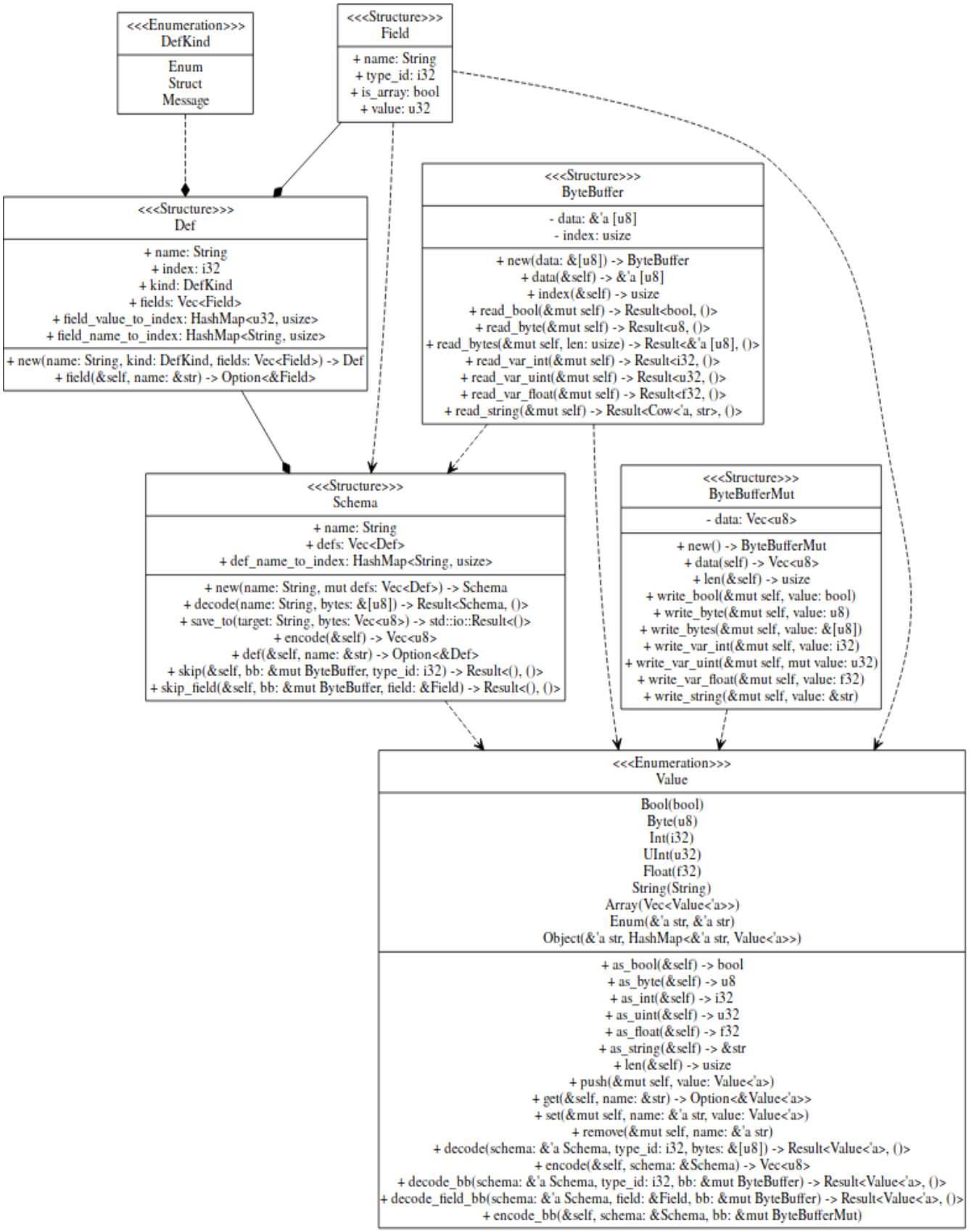


Рисунок 3.5 – Структурна схема модуля серіалізації

Алгоритм серіалізації простого натурального числа представлений в лістингу 3.6.

```
loop {
    let byte = value as u8;
    byte &= 127;
    value >>= 7;
    if value == 0 {
       write_chunck(byte);
       break;
    }
    write_chunck(byte|128);
}
```

Лістинг 3.6 – Частина коду функції сереліалізації натурального 32 бітного числа

У представлені алгоритму, кожне число розбивається на частини по 7 бітів, а старший, восьмий, біт представляє 1 або 0, в залежності від того, чи є частина останньою.

Реалізація десеріалізації зображена на лістингу 3.7).

```
let mut shift: u8 = 0;

    loop {

       let byte = read_byte()?;

       result |= (byte & 127) << shift;

       shift += 7;

       if ((byte & 128) == 0 || shift >= 35)
```

```
                break;

            }
```

Лістинг 3.7 – Частина коду функції десереліалізації натурального 32 бітного числа

Особливу увагу слід приділити методу skip, а точніше обробці вкладених структур, із структураа Schema (лістинг 3.8).

```
Message => {
    loop {
      let value = stream.read_var_uint()?;
      if let Some(index) = defined.value_to_index.get(&value)
        skip_field(stream, &defined.fields[*index])?;
      else
        return Err(());
    }
}
```

Лістинг 3.8 – Частина коду функції skip, відповідальна за обробку структур

Завдяки можливості ігнорувати невалідні дані, алгоритм дає змогу обробляти застарілі схеми і навіть пропускати схеми із надлишковою інформацією.

Такий функціонал є важливим для розподілених систем, оскільки, незважаючи на протокол консенсусу, існують субмережі, які мають різні ланцюги розподіленого сховища, проте їм необхідна можливість спілкуватися між собою.

Наприклад, існує дві компанії, вони використовують один і той же протокол передачі даних, проте мають різні версії протоколу. Завдяки серіалізації опційних полів, у них є така можливість, адже алгоритм може без конфліктів викинути поля, які були опціональними у більш новій версії протоколу передачі даних.

3.1.4 Порівняння з Protobuf

Перевірка буде виконана на схемі, представленій у лістингу 3.9.

```
message Person {
  string name = 1;
  int id = 2;
  string email = 3;
  enum PhoneType {
     MOBILE = 0;
     HOME = 1;
     WORK = 2;
  }
  message PhoneNumber {
     string number = 1;
     PhoneType type = 2;
  }
}
```

Лістинг 3.9 – Схема для тестування швидкодії

Тести виконуватимуться для 10000000 операціх серіалізацiЇ та десеріалізації.

Для серілазації отриманий результат зображений на лістингу 3.10.

```
BenchmarkSerializeToDragnit  10000000  230 ns/op  143 B/op  1 allocs/op
BenchmarkSerializeToProtobuf 10000000  197 ns/op  80 B/op   1 allocs/op\end
```

Лістинг 3.10 – Результат замірів роботи серіалізації алгоритмів

Для десеріалізації отриманий результат на лістингу 3.11.

BenchmarkSerializeToDragnit 10000000 711 ns/op 421 B/op 15 allocs/op

BenchmarkSerializeToProtobuf 10000000 461 ns/op 272 B/op 9 allocs/op\end

Лістинг 3.11 – Результат замірів роботи десеріалізації алгоритмів

Графік, отриманий в результаті тестів, зображений на рисунку 3.6.

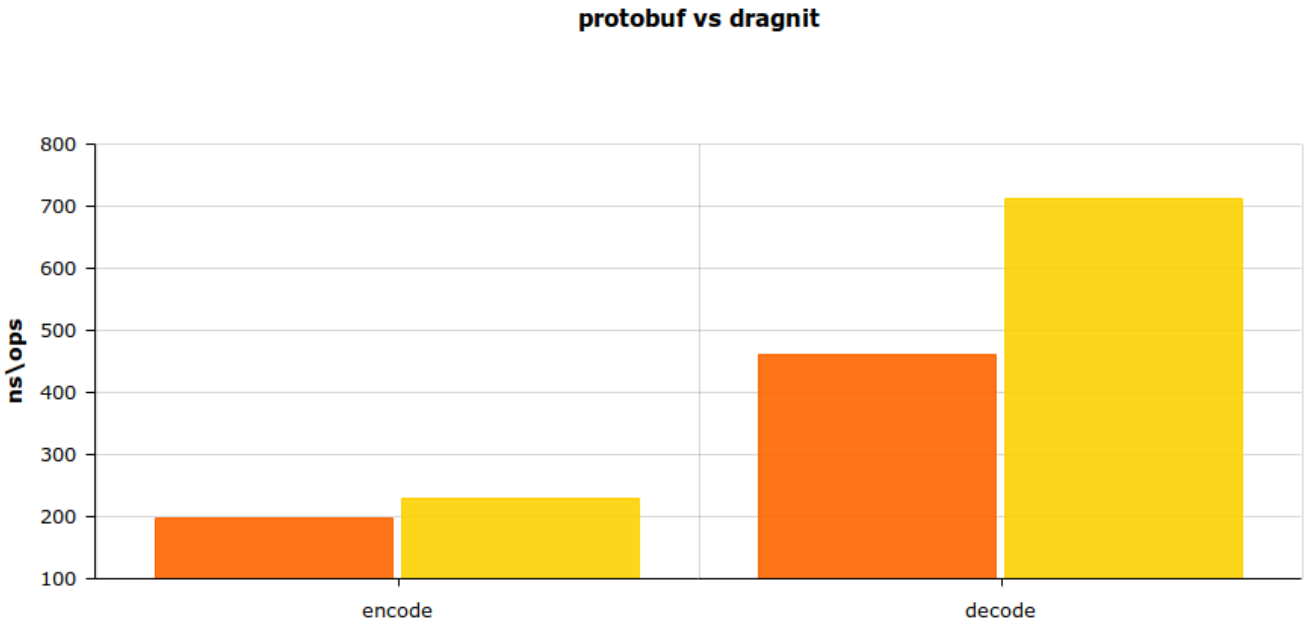


Рисунок 3.6 – Швидкість операцій (зліва – Protobuf, зправа – dragnit)

Як можна побачити, в десеріалізації dragnit сильно програє protobuf, проте це пов'язано із перевіркою на опціональні поля, а ця особливість є критично необхідною для платформи.

3.2 Програмна віртуальна машина

Програмна віртуальна машина необхідна для створення захищеної середи виконання криптографічних операцій, та створення агностичної платформи, яка зможе запускатися на будь-якій ОС.

Так-як потужність сучасного смартфона може позмагатися із простим ПК, то важливо, щоб платформа могла запускатись усюди, при цьому використовуючи мінімум ресурсів.

У розділі розглянуто реалізація простої стекової машини, парсера програмної мови, оптимізація її швидкодії та використання ресурсів, та реалізація базових криптографічних алгоритмів на створеній для віртуальної машини мові.

3.2.1 Архітектура

Обираючи із стекової та реєстрової машини, було обрану стекову, через її простоту, проте і значну швидкодію, ніж реєстрова.

Однією з важливих причин розроблення мов на основі стека є те, мінімалізм їх семантики дозволяє просту інтерпретацію та реалізацію компілятора, а також оптимізацію.

Отже, однією з практичних переваг такої парадигми є те, що вона дозволяє розробникам легко будувати над ними складніші речі та парадигми.

Серед переваг також можна назвати:

– час процесора – вартість часу на розподіл пам'яті в стеку практично безкоштовна, неважливо, виділяється одна чи тисяча цілих чисел, все, що потрібно, – це операція зменшення покажчика стека;

– витік пам'яті – під час використання стеку немає витоків пам'яті, це відбувається природно, без додаткових накладних витрат на вирішення цього питання, пам'ять, яку використовує функція, повністю звільняється при поверненні з кожної функції навіть при обробці винятків або використанні longjmp (відсутність посилань на підрахунок, збирання сміття, тощо);

– фрагментація – стеки також уникають фрагментації пам'яті природним шляхом, можна домогтися нульової фрагментації без будь-якого додаткового коду для вирішення цього питання, наприклад, пулу об'єктів або розподілу пам'яті на платформі;

– локальність – дані в стеку надають перевагу локальному збереженню, використовуючи кеш-пам'ять та уникаючи змін сторінок.

Віртуальна машина також реалізована мовою Rust.

Архітектура машини досить проста [14]. За значення стеку відповідає StackValue, зображений на рисунку 3.7.

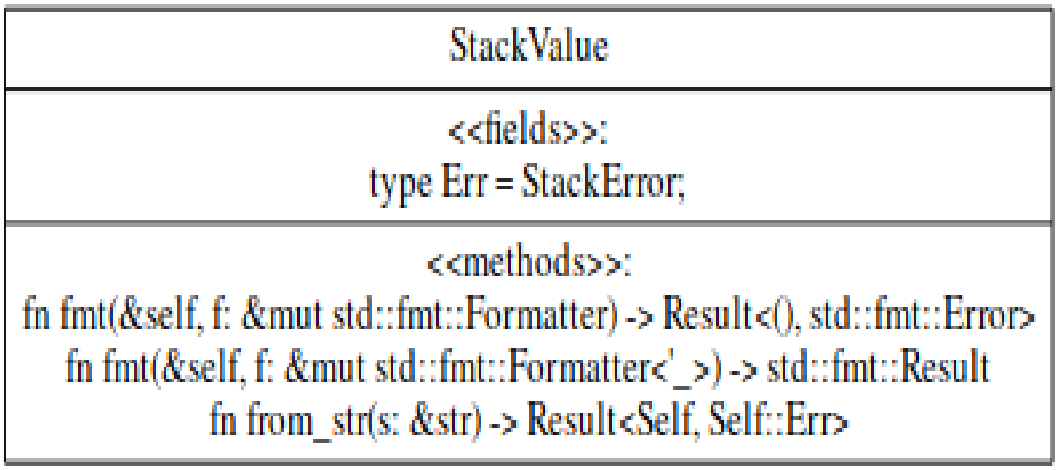


Рисунок 3.7 – Структура абстракції StackValue

Основою ж є структура Machine, який дозволяє виконувати операції 3.8.

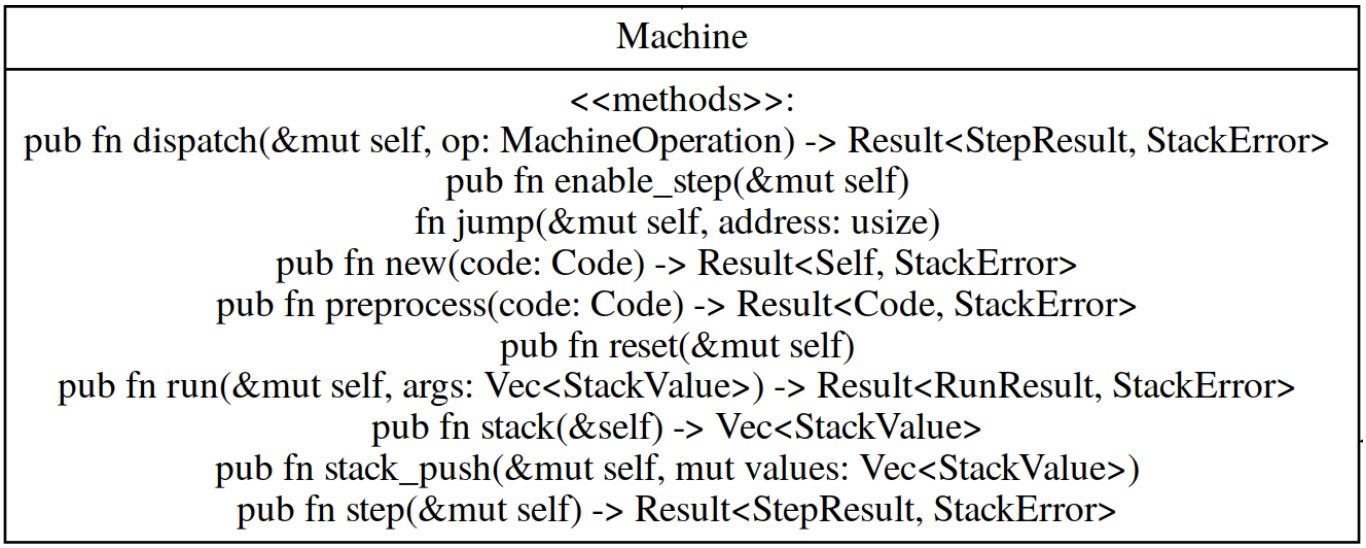


Рисунок 3.8 – Структурна схема абстракції віртуальної машини

Більш детально реалізація буде розглянута у наступних розділах.

Список операцій, які підтримує VM, перелічені у таблиці 3.1.

Таблиця 3.1 – Таблиця підримуємих операцій

Ім'я операції	Opcode	Сігнатура операції
LShift	<<	Push(Num(a << b))
RShift	>>	Push(Num(a >> b))
Plus	+	Push(Num(a + b))
Minus	-	Push(Num(b - a))
Multiply	*	Push(Num(a * b))
Divide	/	Push(Num(b / a))
ToInt	cast_int	Push(Num(a.parse::))
ToStr	cast_str	Push(String(format!(a)))
Println	println	Println(a)
Equals	==	Push(Bool(a == b))
Or	or	Push(Bool(a || b))
BitOr	|	Push(Num(a | b))
And	and	Push(Bool(a \&\& b))
Not	not	Push(Bool(!a))
Xor	xor	Push(Num(a \^{} b))
LessThan	<	Push(Bool(b < a))
LessThanOrEqualTo	<=	Push(Bool(b <= a))
GreaterThan	>	Push(Bool(b > a))
GreaterThanOrEqualto	>=	Push(Bool(b >= a))
Mod	%	Push(Num(b \% a))
If	if	Push(if cond { t } else { f })
Jump	jmp	Jump(a as usize)
Duplicate	dup	PushTwo(val.clone(), val)
Drop	drop	NA
Rotate	rot	PushThree(b, a, c)
LongRot	lrot	PushThree(b, c, a)

Продовження таблиці 3.1

ShortRot	srot	PushThree(a, c, b)
Fup	fup	PushFour(d,b,c,a)
Swap	swap	PushTwo(a, b)
SleepMS	sleep_ms	Sleep(a as u64)
Exit	exit	Stop(exit_code as i32)
Hex	hex	Push(String(format!(a)))
Stop	stop	Stop(0)
Read	read	ReadLn
Over	over	PushThree(b.clone(), a, b)
Call	call	Call(a as usize)
Return	return	Return

3.2.2 Огляд реалізації та роботи віртуальної машини

Для початку, потрібно розглягути одні з найважливіших функцій – токенізатор та парсер.

Вони реалізовані не через регулярні вирази, так як це було би ресурсовитратно. Натомість, токенізатор реалізований через PEG граматику.

Граматики розбиття (PEG) – це досить молоде відкриття у світі граматики та розбору. Вони були запропоновані Брайаном Фордом у 2004 році.

Здебільшого вони описують спосіб зчитування входів та деструкцію їх на правила замість пояснення, як створювати рядки, як це роблять інші граматики. Брайан Форд створив їх, маючи на увазі мови програмування, і як такі вони добре підходять для опису комп'ютерних/машинних граматик (DSL, JSON, граматика D, навіть самої граматики PEG).

Принципова відмінність між безконтекстними граматиками та граматиками вираження розбору полягає в тому, що оператор вибору PEG впорядкований. Якщо перша альтернатива успішна, друга альтернатива ігнорується. Таким чином, упорядкований вибір не є комутативним, на відміну від невпорядкованого вибору,

як у граматиках без контексту. Впорядкований вибір аналогічний операторам програмного забезпечення, який доступний у деяких мовах програмування логіки.

Наслідком цього є те, що якщо CFG транслітерується безпосередньо до PEG, будь-яка двозначність у першому вирішується шляхом детермінованого вибору одного дерева розбору з можливих синтаксичних аналізів. Ретельно вибираючи порядок, у якому вказані альтернативи граматики, програміст має великий контроль над тим, яке дерево аналізу буде вибрано.

Як булеві контекстні граматики без розбору, граматики вираження розбору також додають синтаксичні предикати. Оскільки вони можуть використовувати довільно складний підвираз, щоб "дивитись вперед" у вхідний рядок, не фактично використовуючи його, вони забезпечують потужну синтаксичну функцію пошуку та розбірливості, зокрема при упорядкуванні альтернатив не можуть вказати потрібне дерево розбору [15].

Вигляд абстракції стану токенізатора представлений на лістингу 3.12.

```
struct TokenizerState {
    prev_escape: bool,
    ignore_eol: bool,
    token: String,
    tokens: Vec<StackValue>,
}
```

Лістинг 3.12 – Структура стану токенізатора

Стан включає в себе поточний токен, список усіх токенів, які були до поточного та два прапори – прапор екранізації символів та прапор ігнорування строки (для коментарів).

Токенізатор перебирає вхідний поток даних, зчитуючи зі строки кожен символ і заносячи його до поля 'token' поточного стану.

При виявлені спеціальних символів, таких як '#' (коментар) або '\' (екранізація),

токенізатор перевіряє поточний стан і вирішує, що із ним робити.

На лістингу 3.13 представлена частина функції, яка відповідальна за перевірку поточного стану токенізатора.

```
 if !token.is_empty() {
  let value = StackValue::from_str(&token)?;
  tokens.push(value);
  token.clear();
}
Ok(())
```

Лістинг 3.13 – Частина функції токенізатора, відповідальна за обробку стану

Віртуальна машина перевіряє валідність токена ще на стадії токенізації, а не предкомпіляції, тому це прискорює процес обробки помилок [16].

На лістингу 3.14 зображено простий алгоритм, написаний на мові віртуальної машини, який дозволяє просумувати числа із заданим інтервалом.

```
loop:
    over + dup println
    rot over over
    >= finish continue if jmp
    continue:
    rot rot
    loop jmp
    finish:
    stop
```

```
get:
println read cast_int return
```

Лістинг 3.14 – Приклад вихідного коду VM

Даний алгоритм простий і розглядатись його виконання не буде, він наведений лише для прикладу граматики.

Також, однією із особливостей є предкомпіляція коду, на основі граматики можна заздалегіть розібрати, слід виконувати код, чи ні.

Частина препроцесора, зображена на лістингу 3.15, необхідна, щоб заздалегіть визначати мітки і перетворювати їх в дерева переходів.

```
for (index, value) in program.iter().enumerate() {
    if let Label(ref s) = *value {
        let entry = labels_meta.entry(s).or_insert((vec![], vec![]));
        entry.0.push(index + 1);
    } else if let PossibleLabel(ref s) = *value {
        let entry = labels_meta.entry(s).or_insert((vec![], vec![]));
        entry.1.push(index);
    }
```

Лістинг 3.15 – Частина коду VM, відповідаюча за побудову дерева маркерів

Токінезований код перевіряється на присутність міток, усі дані записуються у хеш-таблицю метаданих переходів. Після цього, метадані перевіряються на правильність побудови і можливість переходів між існуючими маркерами, алгоритм перевірки приведений на лістингу 3.16

```
if val.0.len() > 1 { return Err(MultipleLabelDefinitions {
```

```
            label: (*key).into(),
            locations: val.0.clone(),
        });
    } else if val.0.is_empty() && !val.1.is_empty() {
        return Err(UndefinedLabel {
            label: (*key).into(),
            times: val.1.len(),
        });
    } else
        replacements.push((val.0[0], val.1));
```

Лістинг 3.16 – Частина коду VM, відповідаюча за перевірку правильності побудови маркерів

Для перевірки працездатності, для початку, реалізован простий шифр XOR, вихідний код наведений на лістингу 3.17.

```
loop:
    lrot
    dup
    lrot
    swap
    xor
    println
    swap
    1 -
    dup
```

```
    0
    <= finish continue if jmp
continue:
    swap
    loop jmp
finish:
    stop
```

Лістинг 3.17 – Код VM для шифру XOR

Функція приймає на вхід набір байтів (які можуть представляти текст у будь-якому кодуванні), довжину слова, та ключ, яким буде шифруватися повідомлення.

На виході буде отримано перевернутий масив байтів. Первернутий, тому що повна ротація стека дуже накладна операція, нативні функції зможуть перевернути слово швидше.

Для відладки коду можна скористатися дебагером, який вбудовано у VM, проте користувацький інтерфейс, для простоти і швидкості реалізований у якості консольного додатку.

Для перевірки, буде зашифровано слово "test" (74 65 73 74) за ключем 10.

На рисунку 3.9 зображена відладка стеку під час виконання:

```
rostegg@rostegg:~/masters-dissertation/stuff/crimthy-vm$ cargo run -- examples/xor 74 65 73 74 4
   Compiling crimthy_vm v0.1.0 (/home/rostegg/masters-dissertation/stuff/crimthy-vm)
    Finished dev [unoptimized + debuginfo] target(s) in 1.48s
     Running `target/debug/run examples/xor 74 65 73 74 4 10`
op:     PushThree(Num(4), Num(74), Num(10))
self.code[self.instruction_ptr - 1]:    Operation(LongRot)
self.stack:     [Num(74), Num(65), Num(73)]
self.return_stack:      []
...

op:     PushTwo(Num(10), Num(10))
self.code[self.instruction_ptr - 1]:    Operation(Duplicate)
self.stack:     [Num(74), Num(65), Num(73), Num(4), Num(74)]
self.return_stack:      []
...

op:     PushThree(Num(10), Num(74), Num(10))
self.code[self.instruction_ptr - 1]:    Operation(LongRot)
self.stack:     [Num(74), Num(65), Num(73), Num(4)]
self.return_stack:      []
...

op:     PushTwo(Num(10), Num(74))
self.code[self.instruction_ptr - 1]:    Operation(Swap)
self.stack:     [Num(74), Num(65), Num(73), Num(4), Num(10)]
self.return_stack:      []
...

op:     Push(Num(64))
self.code[self.instruction_ptr - 1]:    Operation(Xor)
self.stack:     [Num(74), Num(65), Num(73), Num(4), Num(10)]
self.return_stack:      []
...

op:     Println(Num(64))
self.code[self.instruction_ptr - 1]:    Operation(Println)
self.stack:     [Num(74), Num(65), Num(73), Num(4), Num(10)]
self.return_stack:      []
...
```

Рисунок 3.9 – Відладка виконання XOR шифра

На рисунку 3.10 зображений результат виконання операції:

```
     Running `target/debug/run examples/xor 74 65 73 74 4 10`
64
67
75
64
```

Рисунок 3.10 – Результат виконання XOR шифра

Для перервірки результату використаємо консольне середовище мови Python (рисунок 3.11).

```
rostegg@rostegg:~/masters-dissertation/stuff/crimthy-vm$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 74 ^ 10, 65 ^ 10, 73 ^ 10, 74 ^ 10
(64, 75, 67, 64)
>>>
```

Рисунок 3.11 – Перевірка правильності виконання алгоритму на мові Python

Отже, алгоритм реалізований на віртуальній машині працює коректно.

На рисунку 3.12 зображений результат продуктивності виконання XOR коду на VM:

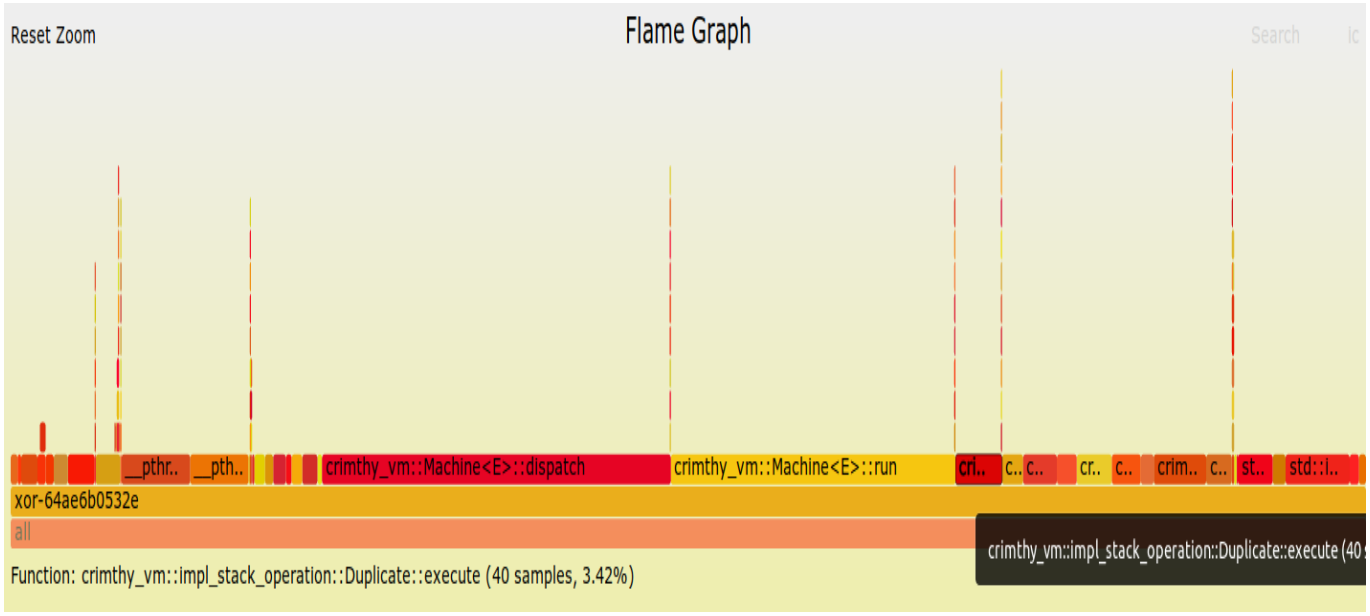


Рисунок 3.12 – Діаграма процесів VM при виконанні XOR алгоритму

Найбільше часу займають функції dispatch та run, це очевидно, так як вони являються основними функціями VM. Проте, дуже часто в алгоритмі використовується метод 'dup', це не погано, проте, можна подумати, чи є ще можливість оптимізувати алгоритм.

3.2.3 Реалізація та оптимізація алгоритму Adler32

Adler-32 – алгоритм контрольної суми, який був винайдений Марком Адлером у 1995 році і є модифікацією контрольної суми Флетчера. Порівняно з CRC, він більш швидкий. Adler-32 надійніший за Fletcher-16 і трохи менш надійний, ніж Fletcher-32.

Adler-32 використовується в бібліотеці компресії даних zlib, яка славиться своєю швидкістю, а швидкість є дуже цінним показником в децентралізованих мережах.

Чим швидше будуть перевірятися пакети на валідність, тим швидше буде розрішуватися консенсус у мережі.

Вихідний код алгоритму Adler32 представлений на лістингу 3.18.

```
loop:
    fup
    + 65521 swap
    % dup lrot
    + 65521 swap
    % swap rot
    1 - dup 0
    <= finish continue if jmp
continue:
    srot
    loop jmp
finish:
    srot swap
    16 << |
    hex println
    stop
```

Лістинг 3.18 – Вихідний код функції Adler32

Код стекової машини трохи складний для людини, якщо його інтерпретувати у мову С, як представлено на лістингу 3.19.

```
uint32_t adler32(const void *buf,
                 size_t buflength) {
  const uint8_t *buffer = (const uint8_t*)buf;

  uint32_t s1 = 1;
  uint32_t s2 = 0;

  for (size_t n = 0; n < buflength; n++) {
     s1 = (s1 + buffer[n]) % 65521;
     s2 = (s2 + s1) % 65521;
  }
  return (s2 << 16) | s1;
}
```

Лістинг 3.19 – Вихідний код функції Adler32 на мові С

Функція приймає на вхід набір байтів (які можуть представляти текст у будь-якому кодуванні) та довжину слова.

На виході буде отримано значеня хеш у 16-ічній системі. Для простоти відладки віртуальна машина може працювати із перетвореням даних у різні формати, проте для більшої оптимізації такі функції необхідно реалізовувати функціонал самої VM, так як це буде не тільки швидше, а й безпечніше.

Для перевірки буде дана проста послідовність байтів – (1, 2, 3), так як такий результат не важко порахувати, хешем послідовності буде значення 0x11.

На рисунку 3.13 зображена відладка стеку під час виконання.

```
rostegg@rostegg:~/masters-dissertation/stuff/crimthy-vm$ cargo run -- examples/adler_32 1 2 3 3 0 1
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/run examples/adler_32 1 2 3 3 0 1`
op:     PushFour(Num(3), Num(0), Num(3), Num(1))
self.code[self.instruction_ptr - 1]:    Operation(Fup)
self.stack:     [Num(1), Num(2)]
self.return_stack:      []
...

op:     Push(Num(4))
self.code[self.instruction_ptr - 1]:    Operation(Plus)
self.stack:     [Num(1), Num(2), Num(3), Num(0)]
self.return_stack:      []
...

op:     PushTwo(Num(65521), Num(4))
self.code[self.instruction_ptr - 1]:    Operation(Swap)
self.stack:     [Num(1), Num(2), Num(3), Num(0)]
self.return_stack:      []
...

op:     Push(Num(1))
self.code[self.instruction_ptr - 1]:    Operation(Mod)
self.stack:     [Num(1), Num(2), Num(3), Num(0)]
self.return_stack:      []
...

op:     PushTwo(Num(1), Num(1))
self.code[self.instruction_ptr - 1]:    Operation(Duplicate)
self.stack:     [Num(1), Num(2), Num(3), Num(0)]
self.return_stack:      []
...
```

Рисунок 3.13 – Відладка виконання Adler-32

На рисунку 3.14 зображений результат виконання операції:

```
rostegg@rostegg:~/masters-dissertation/stuff/crimthy-vm$ cargo run -- examples/adler_32 1 2 3 3 0 1
   Compiling crimthy_vm v0.1.0 (/home/rostegg/masters-dissertation/stuff/crimthy-vm)
    Finished dev [unoptimized + debuginfo] target(s) in 1.22s
     Running `target/debug/run examples/adler_32 1 2 3 3 0 1`
0x11
```

Рисунок 3.14 – Результат виконання Adler

На рисунку 3.15 зображений результат продуктивності виконання Adler коду на VM:

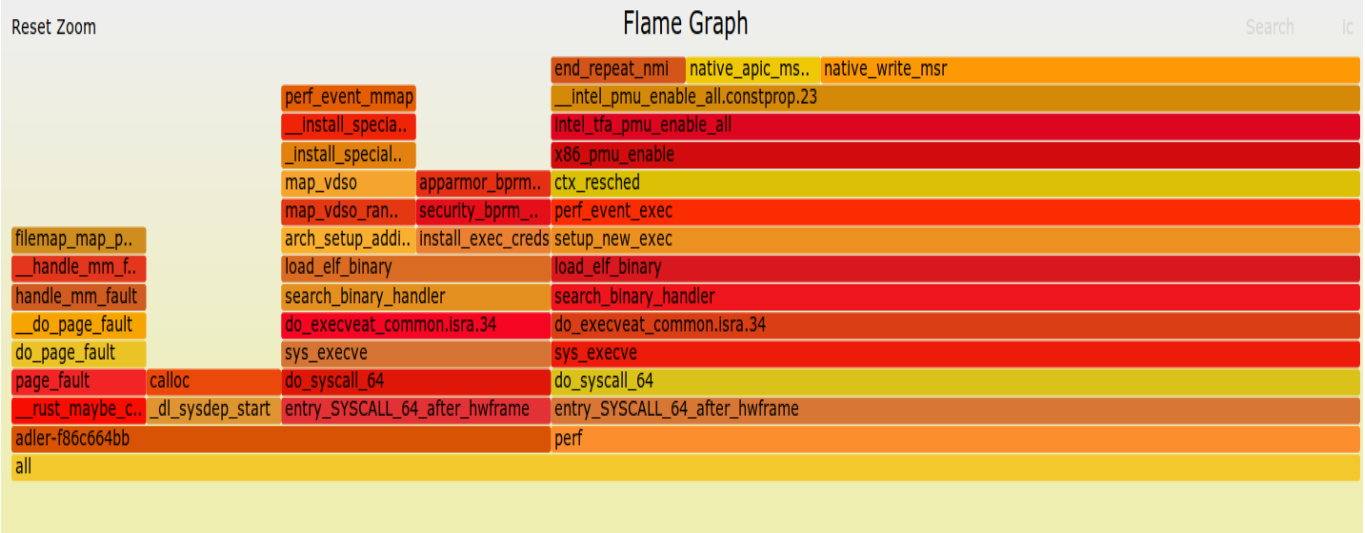


Рисунок 3.15 – Діаграма процесів VM при виконанні Adler алгоритму

3.2.4 Тестування швидкодії

Для тестування швидкодії реалізованих алгоритмів, буду взято 1000, 10000 та 100000 довільних наборів даних.

Для функції XOR, отриманий результат наведений на лістингу 3.20.

```
BenchmarkXOR    1000   180 ns/op  23 B/op  1 allocs/op
BenchmarkXOR    10000  201 ns/op  43 B/op  2 allocs/op
BenchmarkXOR    100000 215 ns/op  51 B/op  5 allocs/op
```

Лістинг 3.20 – Замір швидкодії функції XOR реалізованої на віртуальній машині

Для функції Adler32, отриманий результат наведений на лістингу 3.21.

```
BenchmarkAdler32    1000   223 ns/op  34 B/op  1 allocs/op
BenchmarkAdler32    10000  290 ns/op  67 B/op  1 allocs/op
BenchmarkAdler32    100000 415 ns/op  78 B/op  15 allocs/op
```

Лістинг 3.21 – Замір швидкодії функції Adler32 реалізованої на віртуальній машині

Хоча Adler32 показує просідання по швидкодії, при великих наборах даних, проте це можна уникнути шляхом оптимізації функцій.

В цілому, реалізації алгоритмів показують досить непоганий результат.

3.3 Механізм обробки подій

Для оптимального використання системних ресурсів, необхідний архітектурний підхід, який сможе ефективно управлять ресурсами, цьому не даючи системі можливості простоювати, використовуючи ресурси раціонально.

Для високонавантаженої децентралізованої платформи підійде архітектурний патерн Reactor.

Патер Reactor є однією з реалізації архітектури, керованої подіями. Простіше кажучи, він використовує єдину петлю подій, що блокує події, що виділяють ресурси, і передає їх відповідним обробникам та зворотним викликам.

Не потрібно заблоковувати I/O канали, доки обробники та зворотні виклики для подій реєструються для їх догляду. Події стосуються таких випадків, як нове вхідне з'єднання, готове до читання, готове до запису тощо. Ці обробники/зворотні виклики можуть використовувати пул потоків у багатоядерних середовищах.

Є два важливих компонента в архітектурі Reactor Pattern:

– reactor – реактор працює в окремій потоці, і його завдання полягає в тому, щоб реагувати на події вводу-виводу, направляючи роботу до відповідного обробника, це як телефонний оператор у компанії, який відповідає на дзвінки від клієнтів і передає лінію відповідному контакту;

– handler – обробник виконує фактичну роботу, яка повинна бути виконана з подією вводу-виводу, подібно до фактичного співробітника компанії, з яким клієнт хоче поговорити, реактор реагує на події вводу-виводу шляхом відправлення відповідного обробника, обробники виконують неблокуючі дії.

Архітектурна схема Reactor дозволяє керувати подіями додатка для демультиплексування та диспетчеризації запитів на послуги, які доставляються до програми від одного або декількох клієнтів.

Один реактор буде продовжувати шукати події та інформуватиме відповідного обробника подій, щоб обробляти його, як тільки подія запускається.

Шаблон реактора – це схема дизайну для синхронного демультиплексування та порядку подій у міру їх надходження.

Він отримує повідомлення, запити та з'єднання, що надходять від декількох одночасно клієнтів, і обробляє ці повідомлення послідовно, використовуючи обробники подій. Мета схеми дизайну Reactor – уникнути поширеної проблеми створення потоку для кожного повідомлення, запиту та з'єднання. Потім він отримує події від набору обробників і поширює їх послідовно до відповідних обробників подій.

Для реалізації патерну, за основу був обраний файловий дескриптор Unix, проте для підтриманя агностичності платформи, також необхідно реалізовувати механізми для Windows систем та FreeBSD систем, так вони реалізують інші механізми.

3.3.1 Реалізація через Epoll

Epoll – це системний механізм ядра Linux для масштабованого механізму сповіщення про події вводу/виводу, вперше введений у версії 2.5.44 основної лінії ядра Linux [17].

Його функція полягає у відстеженні декількох дескрипторів файлів, щоб побачити, чи можливі I/O операції на будь-якому з них. Він призначений замінити старі системні виклики POSIX для досягнення кращої продуктивності в більш вимогливих додатках, де кількість активних дескрипторів файлів велика.

На відміну від старих системних викликів, які працюють за час O(n), epoll працює за O(1) час.

Так як патерн буде реалізовуватся на мові Rust, необхідно отримати доступ до нативних функцій epoll.

На лістингу 3.22 зображено обгортку над нативними функціями, яка дозволяє робити безпечні виклики та обробляти помилки.

```
extern {
        pub fn epoll_create(size: c_int) -> c_int;

        pub fn epoll_create1(flags: c_int) -> c_int;
```

```
    pub fn epoll_ctl(epfd: c_int,
                     op: c_int,
                     fd: c_int,
                     event: *mut Event) -> c_int;

    pub fn epoll_wait(epfd: c_int,
                      events: *mut Event,
                      maxevents: c_int,
                      timeout: c_int) -> c_int;
}
```

Лістинг 3.22 – Обгортка над нативними функціями epoll

Такий підхід дозволить безпечно користуватися системними викликами ядра Linux.

Отже, все що необхідно, це реалізувати простий алгоритм, наведений на рисунку 3.16.

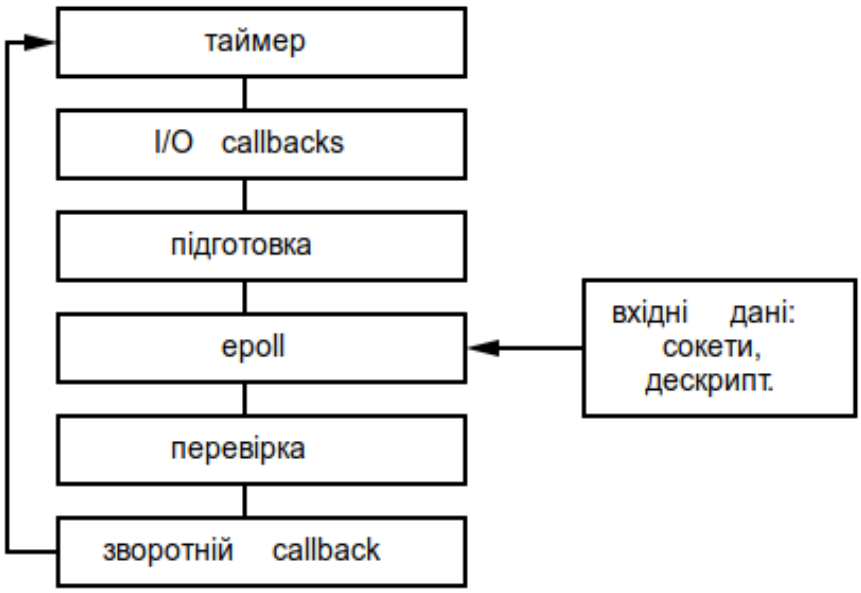


Рисунок 3.16 – Схема роботи патерна Reactor

Для тестів, в якості ресурсів, використовуються сокети, які обробляють прості

HTTP запити.

Для заміру ефективності, використана утиліта wrk, результати представлені на лістингу 3.23.

```
wrk -c100 -d1m -t4 http://127.0.0.1:30000 -H "Host: 127.0.0.1:3000" -H "Accept-Language: en-US,en;q=0.5" -H "Connection: keep-alive"
    Running 1m test @ http://127.0.0.1:3000
    4 threads and 100 connections
    Thread Stats   Avg      Stdev     Max   +/- Stdev
      Latency   523.52us   86.21us  18.64ms   92.15%
      Req/Sec    21.1k     1.95k   32.57k    72.54%
    10543532 requests in 1.00m, 1.40GB read
    Requests/sec: 175725.53
    Transfer/sec:     23.3MB
```

Лістинг 3.23 – Результат заміру швидкодії роботи epoll враперу

Із замірів можна побачити, що патер справді ефективно використовує ресурси, не даючи системі простоювати.

Також, патерн можна доповнити, запустивши його на мультиядерній системі, цим самим значно покращити продуктивність.

3.3.2 Event Loop

Для того, щоб покращити результати чистої обгортки над epoll, слід також реалізувати сховище для подій, де будуть зберігатися усі ще не викликані події. Саме для цього і необхідно реалізувати Event Loop [18].

Структурно, Event Loop виглядатиме, як зображено на рисунку 3.17.

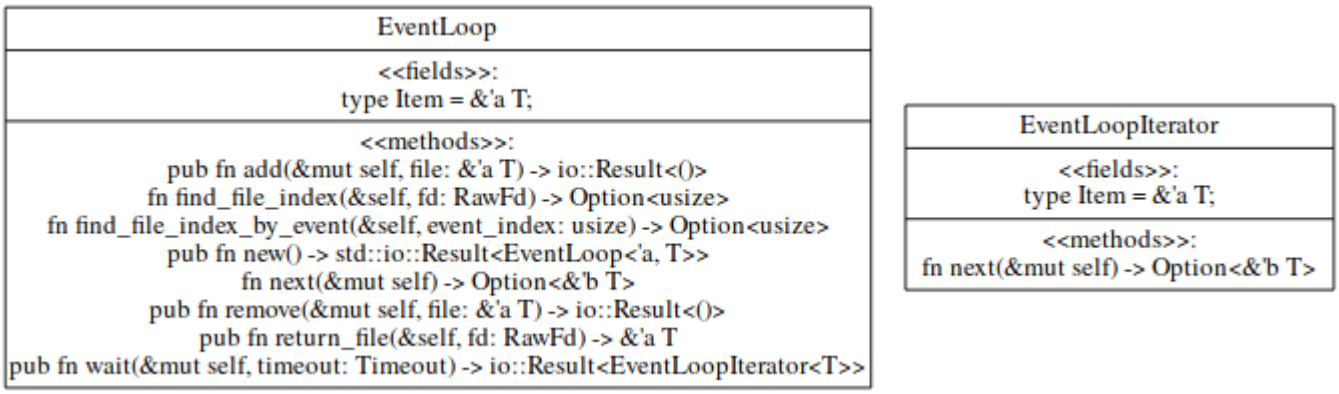


Рисунок 3.17 – Структурна схема Event Loop

Event Loop матиме свій вектор, до якого epoll, після спрацювання, буде додавати доступні дескриптори.

За допомогою EventLoopIterator, реактор буде перебирати ці події та обробляти.

Для заміру ефективності, використана утиліту wrk, результати представлені на лістингу 3.24.

```
wrk -c100 -d1m -t4 http://127.0.0.1:30000 -H "Host: 127.0.0.1:3000" -H "Accept-Language: en-US,en;q=0.5" -H "Connection: keep-alive"
    Running 1m test @ http://127.0.0.1:3000
    4 threads and 100 connections
    Thread Stats   Avg      Stdev     Max   +/- Stdev
      Latency   500.11us   84.11us  18.34ms   90.47%
      Req/Sec    20.5k     1.77k   31.21k    73.21%
    10765475 requests in 1.00m, 1.45GB read
    Requests/sec:  179424.58
    Transfer/sec:     24.16MB
```

Лістинг 3.24 – Результат заміру швидкодії роботи реалізації Event Loop через epoll

Як можна побачити, деякі параметри трохи покращились, наприклад реактор став обробляти більше запитів, тому оптимізацію можна вважати вдалою.

3.4 Протокол консенсусу Honey Badger Byzantine Fault Tolerant

Дивовижний успіх криптовалют призвів до сплеску інтересів у розгортанні широкомасштабних, високоміцних, візантійських протоколів (BFT) для критично важливих програм, таких як фінансові транзакції [19].

Хоча загальноприйнятий підхід полягає у побудові синхронного протоколу, такого як PBFT (або його варіація), такі протоколи критично покладаються на припущення про мережевий таймінг і лише гарантують життєздатність, коли мережа веде себе як очікувалося. Очікується, що ці протоколи не підходять для цього сценарію розгортання.

Тому, було за основу протокола консенсуса було обрано HoneyBadgerBFT, перший практичний синхронний протокол BFT, який гарантує життєздатність, не роблячи жодних припущень про терміни. Рішення базується на новому протоколі atomicbroadcast, який досягає оптимальної асимптотичної ефективності. Така система може досягти пропускної здатності десятків тисяч транзакцій за секунду, і масштабує понад сотню вузлів у широкій мережі.

У HoneyBadgerBFT вузли отримують транзакції як вхідні дані і зберігають їх у своїх (необмежених) буферах. Протокол продовжується в "епохи", де після кожної епохи до заповненого журналу додається нова партія транзакцій. На початку кожної епохи, вузли вибирають підмножину транзакцій у своєму буфері та надають їх як вхід до екземпляра протоколу рандомізованої угоди. Наприкінці домовленості про протокол вибирається остаточний набір транзакцій для цієї епохи. На цьому високому рівні такий підхід схожий на існуючі асинхронно-атомні протоколи атомного мовлення, зокрема на Cachinet, основи для широкомасштабної системи обробки транзакцій (SINTRA). Примітив дозволяє кожному вузлу запропонувати значення і гарантує, що кожен вузол видає загальний вектор, що містить вхідні значення принаймні N-2 правильних вузлів. Історично, щоб побудувати атомну трансляцію з цього примітиву – кожен вузол просто пропонує підмножину транзакцій з передньої своєї черги і виводить об'єднання елементів у узгоджений вектор. Однак є дві важливі проблеми[20].

– досягнення стійкості до цензури – вартість примітивів залежить

безпосередньо від розміру наборів транзакцій, запропонованих кожним вузлом. Оскільки вектор виводу містить щонайменше N множини, ми можемо, таким чином, підвищити загальну ефективність, гарантуючи, що вузли пропонують переважно нероз'єднані набори транзакцій, здійснюючи тим самим більше чітких транзакцій за одну партію за однакові витрати, тому, замість простого вибору першого елемента s з його буфера, кожен вузол протоколу пропонує рандомний вибір, таким чином, що кожна транзакція в середньому пропонується лише одним вузлом;

– практична пропускна здатність – хоча теоретична можливість асинхронних примітивів та атомарного віщання відома, їх практична ефективність не є відомою. Тому цікавим питанням є те, чи можуть такі протоколи досягати високої пропускної здатності.

Honey Badger BFT також забезпечує залишковість блоку та ефективно справляється із непередбачуваною поведінкою. Крім того, він пропонує кілька ключових переваг перед PBFT, в результаті чого процес консенсусу є ефективним та стійким до атаки. Ці покращення включають:

– швидкість блоку не потрібно налаштовувати на основі сценаріїв або припущень, вона відповідає справжній швидкості мережі;

– консенсус без лідерів – на відміну від PBFT, механізм консенсусу HBBFT не вимагає, щоб лідерський вузол пропонував транзакції, кожен вузол є пропонувачем і це виключає потенційні атаки, коли головний вузол може зупинятися на невизначений термін, приводячи до зупинки всю мережу;

– ефективне, безпечне розповсюдження повідомлень – на основі алгоритмів, Міллером та ін., Honey Badger використовує кілька методів для ефективного шифрування, розбиття та надсилання повідомлень невеликими шматками, це економить пропускну здатність і створює надзвичайно ефективний процес.

3.4.1 Архітектура модуля

Процес роботи алгоритму зображений на рисунку 3.18

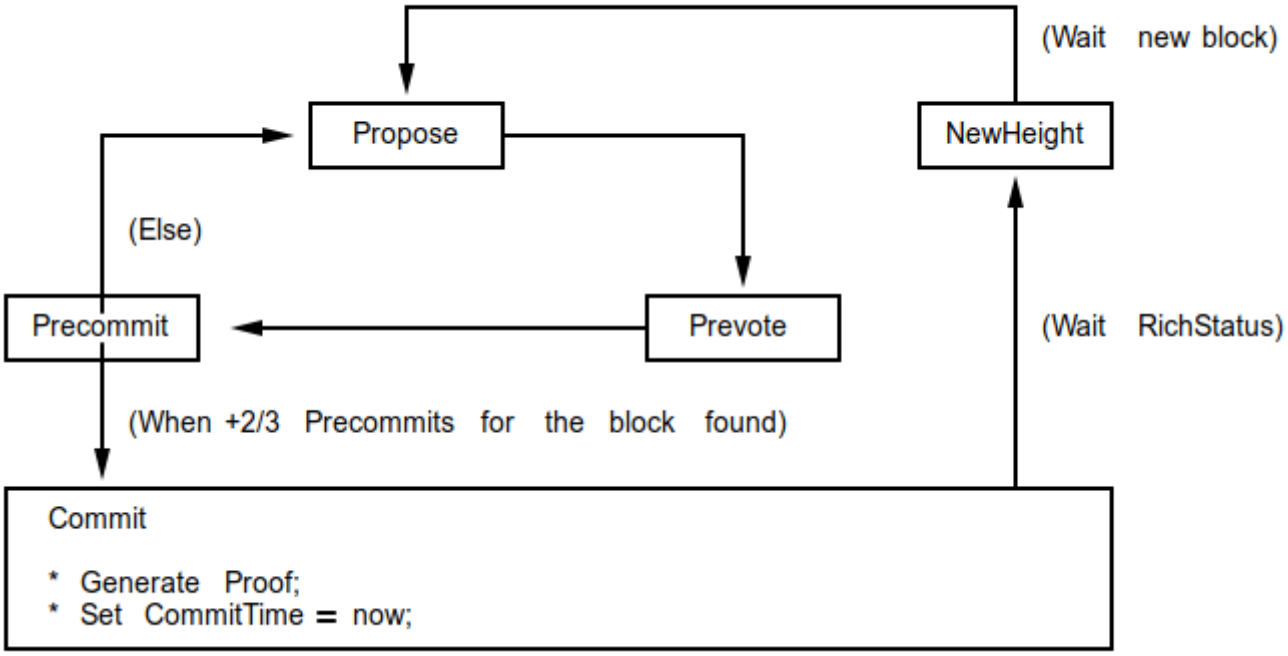


Рисунок 3.18 – Схема роботи алгоритма HBBFT

Так як алгоритм асинхроний, це дає ще більшу оптимізацію використання ресурсів і менший час простою.

Повна модель HBBFT складається з 4 основних частин:

– модуль консенсусу, модуль алгоритму консенсусу включає перевірку підписів, генерацію доказів, перевірку версій тощо;

– машина стану, машина стану HBBFT орієнтована на консенсусну пропозицію;

– транспортний модуль, мережа для модуля консенсусу для зв'язку з іншими модулями;

– місце збереження упереджувальних журналів HBBFT.

Метою HBBFT є повернення узгоджених партій транзакцій [21]. Він не завершує остаточну обробку блоку (наприклад, оновлення залишків рахунків,

виконання смарт-контрактів та підписання блоків) і не додає блоки до блокчейн. Це відповідальність програми, яку HBBFT використовує для спілкування з блок-ланцюгом (наприклад, Parity).

HBBFT розділений на модулі, і кожен модуль обробляє окрему частину процесу консенсусу.

Зв'язок між модулями, як правило, представлений у форматі "чорної коробки". Це означає, що модуль просто отримує вхід і забезпечує вихід. Модулю не потрібно знати, як працює інший модуль, лише який вихід очікувати та вхідний показник. Таке розділення проблем дозволяє легко оптимізувати та підтримувати бібліотеку.

Для того, щоб отримати повну картину про те, як працює HBBFT, необхідно змоделювати ситуацію.

У наступному сценарії, змоделюємо ситуацію, коли один користувач передає 1 токен іншому користувачеві:

Крок 1. Користувач увійдете у систему і використовуєте інтерфейс, щоб заповнити суму для надсилання та адресу гаманця іншого користувача.

Крок 2. Вузол блокчейна отримує цей запит на транзакцію через Інтернет-запит.

Крок 3. Вузол вводить цю транзакцію в HBBFT, ставлячи її в чергу (QHB); QHB розміщує цю транзакцію разом з іншими отриманими у черзі транзакцій, що очікують на розгляд.

Крок 4.Починається нова епоха; Випадковий процес визначає, які транзакції слід включити в наступний блок.

Крок 5. QHB готує список; У мережі є 21 валідатори, тому розмір списку становить 1/21 розмір блоку.

Крок 6. Перелік транзакцій подається до Honey Badger. HB шифрує список за допомогою порогової криптографії, створюючи скриптовану версію, яка містить транзакцію, але її неможливо прочитати.

Крок 7. Вклад подається до алгоритму підмножини.

Крок 8. Підмножина передає його в надійний мовний екземпляр, позначений ідентифікатором вузла (наприклад, Вузол 1); Це розподіляє внесок на кожен інший вузол мережі.

Крок 9. Як тільки кожен вузол отримав зашифрований внесок, вони знають, що всі інші правильні вузлитакож отримають цей внесок; Вони голосують "Y" у екземплярі угоди з іменем "Вузол 1".

Крок 10. Кожен вузол повертає Y для екземпляра договору, позначеного Вузол 1, тобто всі вони згодні з тим, що цей внесок повинен бути включений як частина наступного блоку.

Крок 11. Домовленість досягнута. Внесок повертається назад в стек.

Крок 12. Підмножина повертає внески HB 21 на 21 вузол, всі зашифровані.

Крок 13. В HB вся мережа співпрацює, щоб розшифрувати внесок; Кожен вузол отримує 21 список транзакцій, які розшифровуються. Ці списки включають оригінальну транзакцію.

Крок 14. Потім QHB робить об'єднання внесків і створює єдиний остаточний перелік транзакцій, про який узгодили всі вузли.

Крок 15. Цей остаточний список надсилається з QHB та повертається до клієнта програми.

Крок 16. Блокчейн виконує транзакцію і оголошує її як частину наступного блоку.

Крок 17.Токен передається іншому користувачеві, в блокчейні з'являється запис.

3.4.2 Симуляція та тестування мережі

Оскільки на практиці важко сказати, наскільки алгоритм буде ефективним в тій чи іншій ситуації, то необхідно змоделювати різні умови.

На рисунку 3.19 зображений результат симуляції для 10 вузлів.

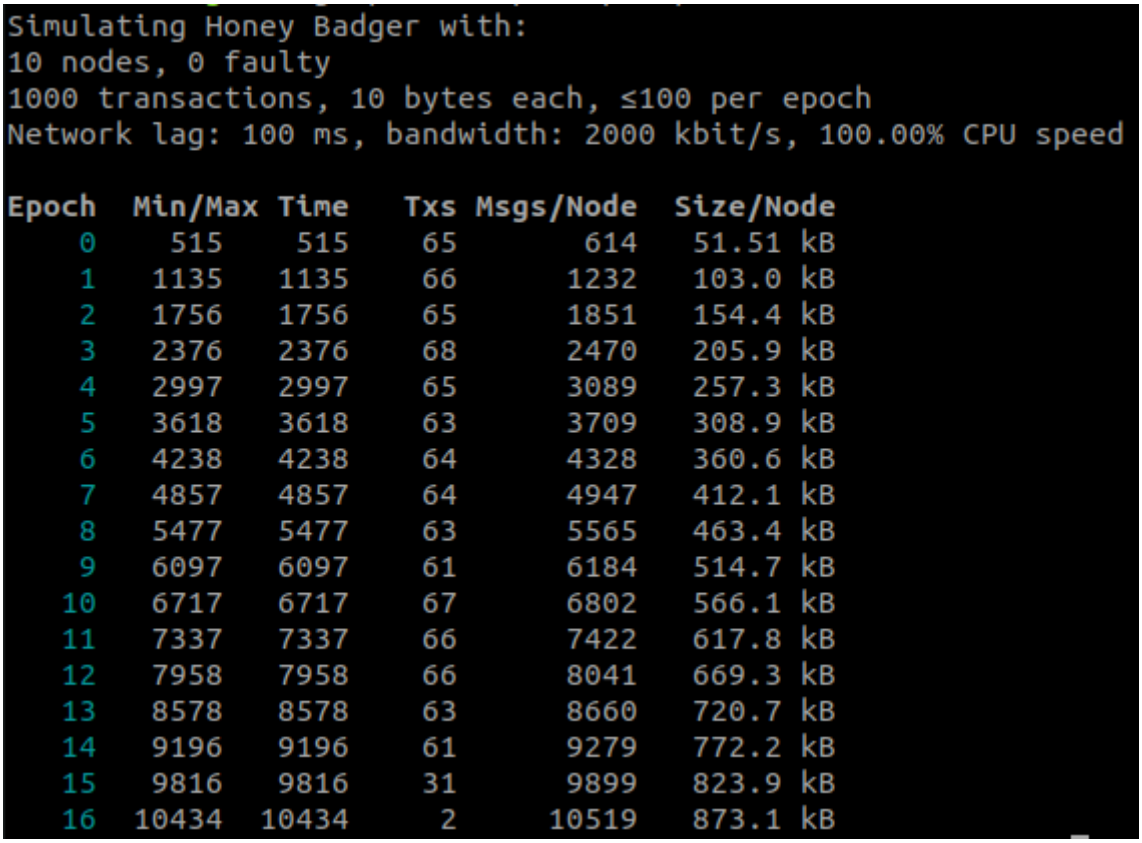


Рисунок 3.19 – Симуляція HBBFT мережі із 10 вузлів

Такий варіант є, практично, неможливим, так як мережа складається всього із 10 вузлів, кожен з яких має стовідсоткову відмовостійкість, тому дана симуляція необхідна лише для перевірки роботи алгоритму та заміру теоретичної швидкості.

Поясненя значень заміру представлені у таблиці 3.2.

Таблиця 3.2 – Поясненя значень тестового заміру продуктивності HBBFT

Значеня	Визначення
Epoch	Номер "епохи". У кожній епосі транзакції обробляються в партії імітованими вузлами у мережі.Пакет завжди виводиться однією частиною, з усіма транзакціями одночасно.

Min Time	Час в імітованих мілісекундах, поки перший справний вузол обробить пакет.
Max Time	Час в імітованих мілісекундах, поки останій справний вузол обробить пакет
Txs	Кількість транзакцій, які були оброблені за "епоху".
Msgs/Node	Середня кількість повідомлень, які обробляються вузлом.
Size/Node	Розмір транзакцій, які були оброблені за "епоху".

Отже, із вище зазначеного тесту, можна зазначити, що хоча він і проходив в ідеальних умовах, проте видно, що із плином часу алгоритм починає працювати швидше, якщо порівняти різницю в часі, між першою-другою "епохою" та другою-третьою, то видно, що час, на прийняття консенсусу зменшився, приблизно, на 33%.

Далі змодельована мережа, де два вузли будуть несправні (рисунок 3.20):

```
Simulating Honey Badger with:
15 nodes, 2 faulty
1000 transactions, 10 bytes each, ≤100 per epoch
Network lag: 100 ms, bandwidth: 2000 kbit/s, 100.00% CPU sp

Epoch  Min/Max Time    Txs Msgs/Node  Size/Node
    0     521    521     58      1398   113.6 kB
    1    1150   1150     62      2808   228.4 kB
    2    1780   1780     58      4222   343.9 kB
    3    2409   2409     62      5633   458.5 kB
    4    3037   3037     60      7046   573.9 kB
    5    3666   3666     59      8459   689.3 kB
    6    4294   4294     61      9870   804.0 kB
    7    4923   4923     58     11282   918.9 kB
    8    5551   5551     63     12694   1.034 MB
    9    6180   6180     58     14108   1.150 MB
   10    6809   6809     56     15519   1.264 MB
   11    7436   7436     62     16929   1.379 MB
   12    8066   8066     59     18344   1.495 MB
   13    8695   8695     60     19757   1.610 MB
   14    9324   9324     61     21169   1.725 MB
   15    9953   9953     63     22581   1.840 MB
   16   10581  10581     37     23990   1.954 MB
   17   11209  11209      3     25400   2.067 MB
```

Рисунок 3.20 – Симуляція HBBFT мережі із 15 вузлів, де 2 вузли несправні

Алгоритм показав дуже гарний результат, навіть при наявності несправних вузлів, що простий BFT ніяк не може гарантувати.

Інша модель покаже, наскільки алгоритм справляється із підвищеням трафіку в мережі (рисунок 3.21).

```
Simulating Honey Badger with:
10 nodes, 0 faulty
1000 transactions, 10 bytes each, ≤500 per epoch
Network lag: 100 ms, bandwidth: 2000 kbit/s, 100.00% CPU speed

Epoch  Min/Max Time    Txs Msgs/Node  Size/Node
    0     521    521    324       614   64.30 kB
    1    1152   1152    321      1232   128.3 kB
    2    1783   1783    269      1852   192.8 kB
    3    2413   2413     86      2472   257.0 kB
```

Рисунок 3.21 – Симуляція HBBFT мережі із 10 вузлів, 500 пакетів кожної ери

Із результатів можна побачити, що алгоритм справився усього за 3 епохи, причому в дуже швидкому темпі.

Це значить, що алгоритм масштабований до високого трафіку и цілком може справлятися з великим навантаженням.

3.5 Протокол GOSSIP

Так як мережа децентралізована і постійно розширювана, то без discovery алгоритму не обійтись [22].

Переваги GOSSIP:

– масштабованість – оскільки загалом потрібна складність O(logN), щоб досягти всіх вузлів, де N – кількість вузлів, також кожен вузол надсилає лише фіксовану кількість повідомлень, незалежних від кількості вузлів у мережі, він не чекає підтверджень, і не вживає жодних дій відновлення, якщо підтвердження не надійде, тому система може легко масштабувати мільйони процесів;

– відмовостійкість – має можливість працювати в мережах з неправильним і невідомим зв’язком, існує багато маршрутів, за якими інформація може надходити від її джерела до місця призначення;

– відмовостійкий – жоден вузол не відіграє певної ролі в мережі, тому несправний вузол не завадить іншим вузлам продовжувати надсилати повідомлення;

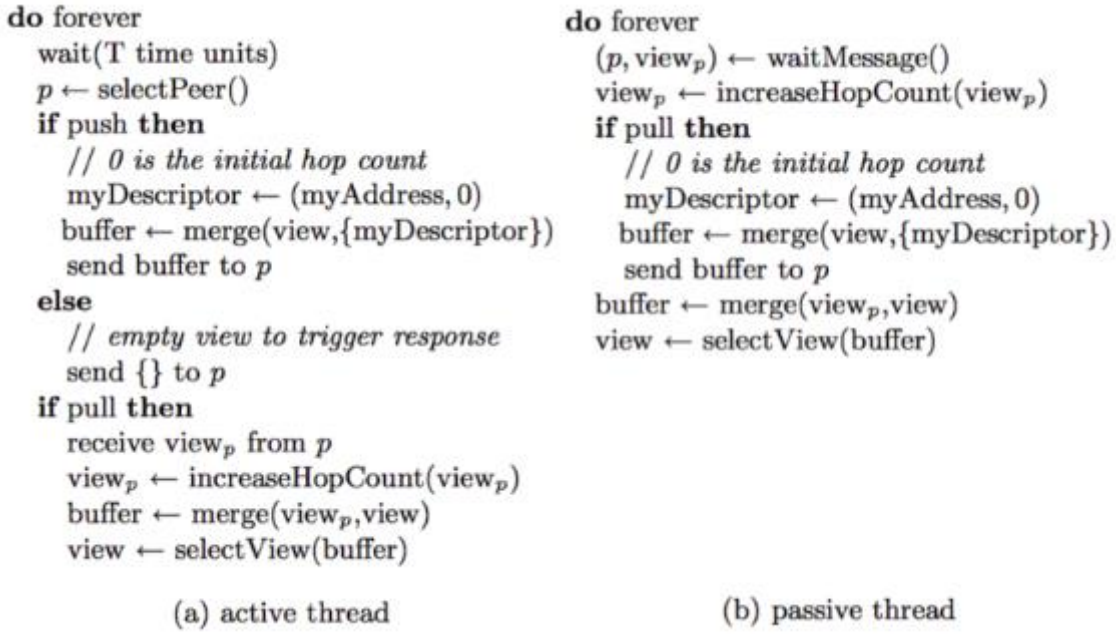
– кожен вузол може приєднатися або вийти, коли йому заманеться, не порушивши загальну якість обслуговування системи, вони не є надійними за будь-яких обставин;

– конвергентна консистенція – протоколи пліток досягають експоненціально швидкого поширення інформації і, отже, швидко переходять в експоненціально стан до глобально послідовного стану після настання нової події, за відсутності додаткових подій, поширючи будь-яку нову інформацію по всіх вузлах, на які буде впливати інформація протягом логарифмічного розміру системи [23];

– надзвичайно децентралізована – плітки пропонують надзвичайно децентралізовану форму пошуку інформації, і її затримки часто прийнятні, якщо

інформація насправді не буде використана негайно;

простота реалізації (рисунок 3.22).



```
do forever
  wait(T time units)
  p ← selectPeer()
  if push then
    // 0 is the initial hop count
    myDescriptor ← (myAddress, 0)
    buffer ← merge(view,{myDescriptor})
    send buffer to p
  else
    // empty view to trigger response
    send {} to p
  if pull then
    receive view_p from p
    view_p ← increaseHopCount(view_p)
    buffer ← merge(view_p,view)
    view ← selectView(buffer)
```

(a) active thread

```
do forever
  (p, view_p) ← waitMessage()
  view_p ← increaseHopCount(view_p)
  if pull then
    // 0 is the initial hop count
    myDescriptor ← (myAddress, 0)
    buffer ← merge(view,{myDescriptor})
    send buffer to p
  buffer ← merge(view_p,view)
  view ← selectView(buffer)
```

(b) passive thread

Рисунок 3.22 – Псевдокод простого алгоритму GOSSIP

3.5.1 Моделювання мережі з протоколом GOSSIP

Для початку, модель буде складатися із 20 вузлів, щоб переконатися, що алгоритм спроможний відкрити всі шляхи.

Кожен вузол буде запам'ятовувати до чотирьох сусідніх вузлів.

Такий підхід дозволить не створювати безліч відновлення реплікуваних даних, а надасть змогу відновлювати зв'язки шляхом порівняння реплік і узгодження відмінностей.

На рисунку 3.23 зображений стан кожного з вузлів.

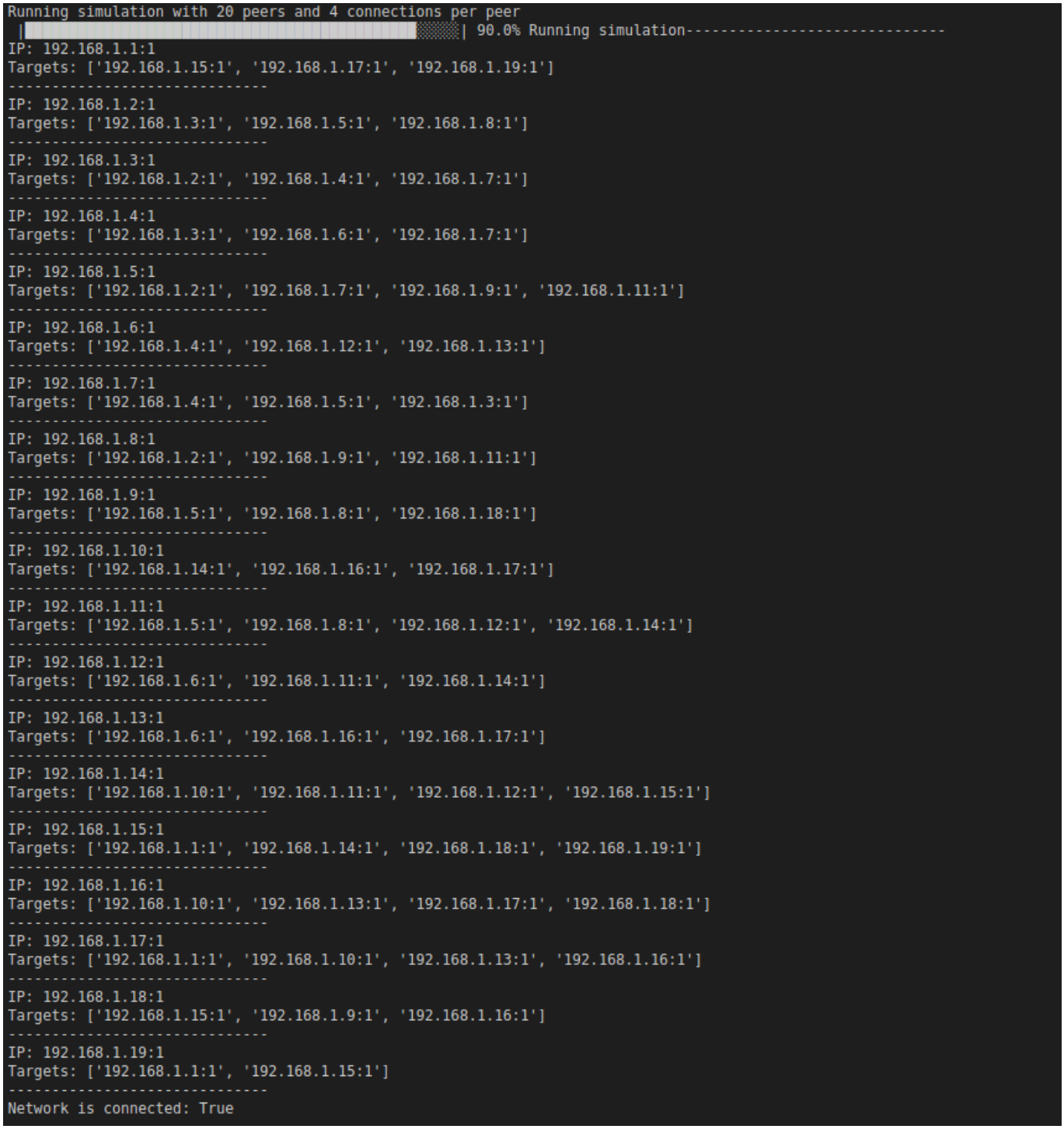


Рисунок 3.23 – Симуляція GOSSIP протокола на 20 вузлах

Із результатів можна зазначити, що кожен вузел має сусідів, тобто усі вони об'єднались у єдину мережу, проте не усі вузли мають рівно по 4 сусіди, це значить, що мережевий граф вдало оптимізувався.

На рисунку 3.24 зображений коефіцієнти кластерів кожного з вузлів.

```
Average shortest path length: 2.6198830409356724
------------------------------
Average bipartite clustering coefficient: 0.23684210526315788
------------------------------
Bipartite clustering coefficient:
{   '1': 0.3333333333333333,
    '10': 0.3333333333333333,
    '11': 0.16666666666666666,
    '12': 0.3333333333333333,
    '13': 0.3333333333333333,
    '14': 0.16666666666666666,
    '15': 0.16666666666666666,
    '16': 0.3333333333333333,
    '17': 0.3333333333333333,
    '18': 0,
    '19': 1.0,
    '2': 0,
    '3': 0.3333333333333333,
    '4': 0.3333333333333333,
    '5': 0,
    '6': 0,
    '7': 0.3333333333333333,
    '8': 0,
    '9': 0}
------------------------------
```

Рисунок 3.24 – Коефіцієнти кластерів для вузлів

Коефіцієнт "average_shortest_path_length" розраховується за формулою 3.1.

$$a = \sum_{s,t \in V} \frac{d(s,t)}{n(n-1)} \tag{3.1}$$

де V – це набір вузлів в графі, d(s,t) – це найкоротший шлях з вузла d до вузла t, а n – кількість вузлів графа.

Коефіцієнт "average_biparite_clustering_coefficient" розраховується за формулою 3.2.

$$C = \frac{1}{n} \sum_{v \in G} c_v \tag{3.2}$$

де n – кількість вузлів графа.

Як і очікувалось, деякі коефіцієнти рівні 0, так як мережа дуже розріджена, а кількість сусідів у вузлів не дуже велика.

Такий підхід дозволяє побудувати насправді росподілений підхід до виявленя вузлів у мережі.

Для більшої візуалізації результатів, побудовано граф (рисунок 3.25).

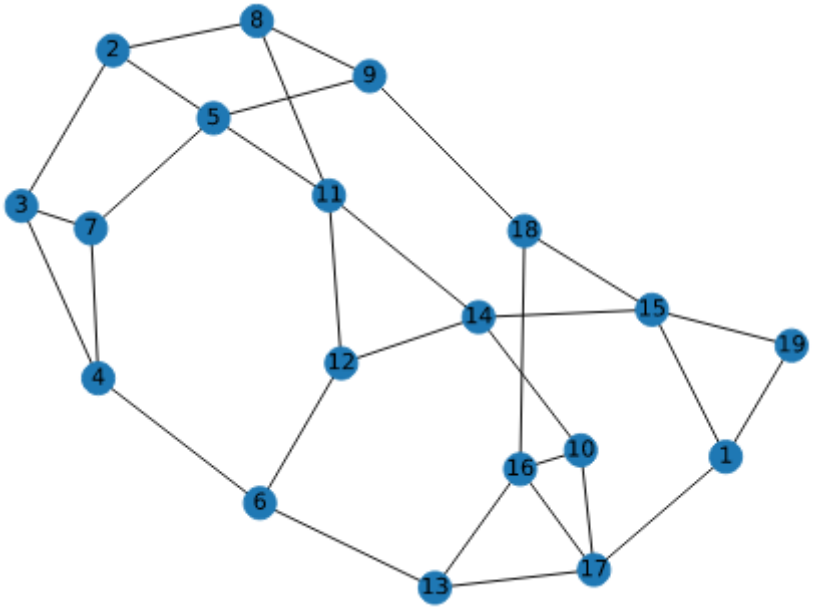


Рисунок 3.25 – Граф зв'язку вузлів у мережі

3.5.2 Оптимізація GOSSIP протоколу

Хоча протокол і показав себе із гарної сторони, проте кожен вузол довільно вибирає, на які вузли він буде посилатися [24]. Такий рандомізований підхід не самий кращий варіант, тому замість довільного вибору, необхідно ввести топології вибору:

– рядок – усі дійові особи розміщені в лінійному порядку, утворюючи лінію, де вузол по позиції і має сусідів по позиціях і + 1 і і-1, максимально можливі сусіди – 2, а мінімум – 1;

– недосконала лінія – розміщення вузла схоже на лінію, але в цьому випадку вузол (і) також матиме випадковий вузол як свого сусіда, крім передових (і

+ 1) та відсталих (i-1) сусідів, максимум можливих сусідів – 3, а мінімальний – 2;

– 2D – вузли розміщуються в 2D квадратній сітці, яка утворюється, якщо кількість вузлів є ідеальним квадратом якщо це не так, округляє о його до найближчого цілого числа, а потім розміщує решти вузлів на краях 2D сітки), максимальна кількість сусідів – 4, а мінімальна – 2;

– недосконалий 2D – те саме, що 2D, але кожен вузол має ще одного випадкового вузол як свого сусіда, таким чином, максимум і мінімум сусідів 4 і 3 відповідно;

– випадковий 2D – дійові особи розташовані випадковим чином у координатах x, y на [0–1.0] X [0–1.0] квадраті, два вузли пов'язані між собою, якщо вони знаходяться в межах .1 відстані від інших вузлів, це цікавий випадок, який має велику ступінь випадковості тому кількість сусідів не відома;

– 3D – вузли розміщені в тривимірному кубі, максимум сусідів 8 і мінімум 3;

– торус – це два виміри зі ступенем 4, вузли уявлені викладеними у двовимірну прямокутну решітку з n рядів і n стовпців, причому кожен вузол з'єднаний із своїми 4 найближчими сусідами та відповідними вузлами на протилежних краях, з'єднання протилежних країв можна візуалізувати, прокатуючи прямокутний масив у "трубу" для з'єднання двох протилежних країв, а потім згинаючи "трубу" в торус, щоб з'єднати інші два, пілкування може відбуватися у 4 напрямках, +x, -x, +y та -y;

– повна – у повній топології вузол має всіх інших вузлів своїм сусідом.

На графіку 3.26 зображено відношеня часу (в мілісекудах) до кількості вузлів.

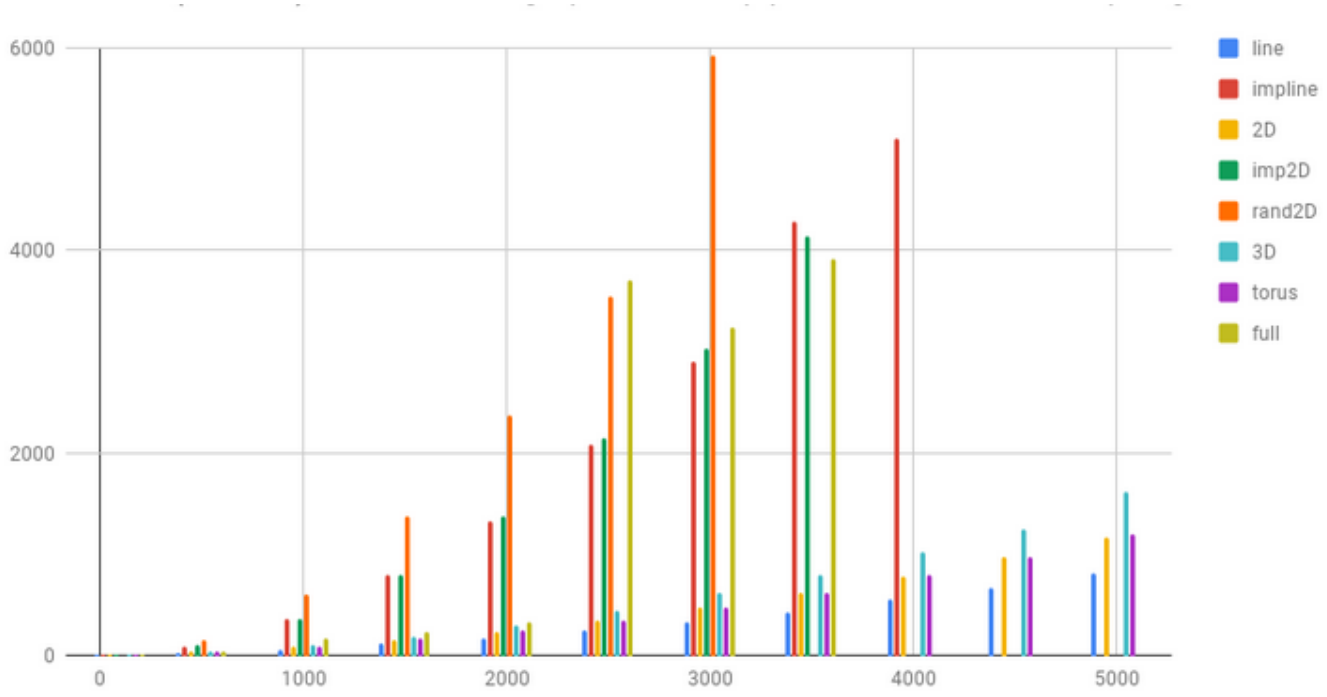


Рисунок 3.26 – Графік продуктивності алгоритма, залежно від топології

Із графіка видно, що хорошими топологіями для протоколу пліток є торус, 2D, лінія та 3D для великої кількості вузлів. Для менших значень суб'єктів повна мережа також дає хороші значення конвергенції.

Там, де торус, 2D і 3D спрацьовують добре для великої кількості N за рахунок балансу між кількістю сусідів і асинхронною взаємодією вузол-сусід.

3.6 Реалізація тестового додатку

Для демонстрації можливостей, найкращим рішеням буде створення простого групового чату.

Чат складається з веб-клієнта (рисунок 3.27), який підключається до платформи через веб-сокет. Клієнту автоматично видається ідентифікатор, і він може необмежено спілкуватись.

Рисунок 3.27 – Інтерфейс групового чату

На рисунку вище показано приклад, як декілька клієнтів спілкуються між собою у реальному часі. Це можливо завдяки використанню WebSocket, це комп'ютерний протокол зв'язку, що забезпечує повнодуплексні канали зв'язку через одне з'єднання TCP.

Завдяки цьому, клієнтські додатки можуть отримувати усю актуальну інформацію із мережі у реальному часі із незначними затримками.

Додаток показує простий спосіб реалізації групового чату, використовуючи компоненти платформи.

Для простоти сприйнятя, блоки виглядають людино читаємими, в JSON форматі (рисунок 3.28)

```
{
    "index": 0,
    "parentHash": "0",
    "ownHash": "4736da8abed4f7db7156afc3676993258165344c2e6337db8a921823784c1378",
    "timestamp": 1575361947387,
    "data": "{\"date\":\"2019-12-03T08:32:27.387Z\",\"author\":\"#Blockchain\",\"msg\":\"init message\"}"
},
{
    "index": 1,
    "parentHash": "4736da8abed4f7db7156afc3676993258165344c2e6337db8a921823784c1378",
    "ownHash": "d2ead4f038107a809c341671b24972802eeeb63ac50737a1cd5b47c607b0a79f",
    "timestamp": 1575361977271,
    "data": "{\"date\":\"2019-12-03T08:32:57.267Z\",\"author\":\"tcxj0bB-24Y_ygjMa2hJCj_qALtFJDxa93BjMbCG\",\"msg\":\"hello world\"}"
},
{
    "index": 2,
    "parentHash": "d2ead4f038107a809c341671b24972802eeeb63ac50737a1cd5b47c607b0a79f",
    "ownHash": "338308ad38ec33aa10fd13ddc4442e85f125880a6f0eb83aa1fb49b84577635b",
    "timestamp": 1575361990089,
    "data": "{\"date\":\"2019-12-03T08:33:10.088Z\",\"author\":\"XvtI9OrvnLv-N-B-exFTl2YhIvs383JLg-lgDFR3\",\"msg\":\"its me\"}"
},
{
    "index": 3,
    "parentHash": "338308ad38ec33aa10fd13ddc4442e85f125880a6f0eb83aa1fb49b84577635b",
    "ownHash": "991a1e839c21bf5187b18cd839e6ae1933e33292777ad47ddc36c3e384f5c408",
    "timestamp": 1575361990091,
    "data": "{\"date\":\"2019-12-03T08:33:10.090Z\",\"author\":\"XvtI9OrvnLv-N-B-exFTl2YhIvs383JLg-lgDFR3\",\"msg\":\"its me\"}"
},
```


Рисунок 3.28 – Формат блокчейн блоків у JSON форматі

Серіалізовані дані за схемою (лістинг 3.25) мають вигляд, представлений на рисунку 3.29.

```
message Block {
  uint index = 1;
  string parentHash = 2;
  string ownHash = 3;
  string timestamp = 4;
  Data data = 5;
}
```

Лістинг 3.25 – Схема для сереліалізації даних додатка

```
rostegg@rostegg:~/masters-dissertation/stuff/blockchain-js-chat$ bat binary.data
       File: binary.data
   1   01 02 02 64 32 65 61 64 34 66 30 33 38 31 30 37 61 38 30 39 63 33 34 31 36 37 31
   2   62 32 34 39 37 32 38 30 32 65 65 65 62 36 33 61 63 35 30 37 33 37 61 31 63 64 35
   3   62 34 37 63 36 30 37 62 30 61 37 39 66 00 03 33 33 38 33 30 38 61 64 33 38 65 63
   4   33 33 61 61 31 30 66 64 31 33 64 64 63 34 34 34 32 65 38 35 66 31 32 35 38 38 30
   5   61 36 66 30 65 62 38 33 61 61 31 66 62 34 39 62 38 34 35 37 37 36 33 35 62 00 04
   6   31 35 37 35 33 36 31 39 39 30 30 38 39 00 05 64 61 74 65 20 32 30 31 39 2d 31 32
   7   2d 30 33 54 30 38 3a 33 33 3a 31 30 2e 30 38 38 5a 20 61 75 74 68 6f 72 20 58 76
   8   74 49 39 4f 72 76 6e 4c 76 2d 4e 2d 42 2d 65 78 46 54 6c 32 59 68 49 76 73 33 38
   9   33 4a 4c 67 2d 6c 67 44 46 52 33 2c 6d 73 67 20 3a 69 74 73 20 6d 65 00 00 01 01
  10   02 34 37 33 36 64 61 38 61 62 65 64 34 66 37 64 62 37 31 35 36 61 66 63 33 36 37
  11   36 39 39 33 32 35 38 31 36 35 33 34 34 63 32 65 36 33 33 37 64 62 38 61 39 32 31
  12   38 32 33 37 38 34 63 31 33 37 38 00 03 64 32 65 61 64 34 66 30 33 38 31 30 37 61
  13   38 30 39 63 33 34 31 36 37 31 62 32 34 39 37 32 38 30 32 65 65 65 62 36 33 61 63
  14   35 30 37 33 37 61 31 63 64 35 62 34 37 63 36 30 37 62 30 61 37 39 66 00 04 31 35
  15   37 35 33 36 31 39 37 37 32 37 31 00 05 7b 22 64 61 74 65 22 3a 22 32 30 31 39 2d
  16   31 32 2d 30 33 54 30 38 3a 33 32 3a 35 37 2e 32 36 37 5a 22 2c 22 61 75 74 68 6f
  17   72 22 3a 22 74 63 78 6a 30 62 42 2d 32 34 59 5f 79 67 6a 4d 61 32 68 4a 43 6a 5f
  18   71 41 4c 74 46 4a 44 78 61 39 33 42 6a 4d 62 43 47 22 2c 22 6d 73 67 22 3a 22 68
  19   65 6c 6c 6f 20 77 6f 72 6c 64 22 7d 00 00
```

Рисунок 3.29 – Формат серіалізованих блоків у HEX форматі

Даний приклад показав використання базових компонентів платформи для створення додатку.

4. РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

4.1 Опис ідеї проекту

Описані в магістерській роботі підходи до проектування платформи для створення децентралізованих додатків можна застосовувати для розробки спеціалізованих платформ та модифікації існуючих. За ідею для стартап-проекту була вибрана технологія масштабованої децентралізованої платформи з можливістю подальшої її модифікації. Узагальнення цієї ідеї можна побачити в таблиці 6.1.

Таблиця 4.1 Опис ідеї стартап проекту.

Зміст ідеї	Напрямки застосування	Вигоди для користувача
		1. Можливість розгортати додатки у децентралізованій мережі
		2. Зменшення витрат ресурсів для створення додатків

Даний проект носить як теоретичний, так і практичний характер, тому він може бути порівняний з сучасними платформами для створення децентралізованих додатків. Порівняння наведено у таблиці 4.2.

Таблиця 4.2 Визначення сильних, слабких та нейтральних характеристик ідеї проекту

Техніко-економічні характеристики ідеї	Потенційні товари та концепції конкурентів			Слабка сторона	Нейтральна сторона	Сильна сторона
	Даний проект	Ethereum	Hyper-ledger			
Здатність швидкого обміну даними	Присутня	Відсутня	Присутня			+
Широкий обсяг інструментів для розробки	Відсутня	Присутня	Присутня			+
Можливість створювати кастомні користувацькі інтерфейси	Присутня	Присутня	Присутня			+
Здатність модернізувати платформу	Присутня	Відсутня	Відсутня		+	

4.2 Технологічний аудит проекту

Проект представляє собою застосування та набір бібліотек, оскільки в ньому використовуються нові технології програмування якими володіє невелика

розробників, але проект цілком реалізуємий. Технології, у контексті яких використовується проект, можна побачити на таблиці 4.3

Таблиця 4.3 Технологічна здійсненність ідеї проекту

Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
Платформа для розроблення децентралізованих додатків	Javascript Python Rust	Технології наявні, їх потрібно об'єднати між собою	Вільні для використання

4.3 Аналіз ринкових можливостей запуску стартап-проекту

Попередня характеристика потенційного ринку стартап-проекту та характеристика потенційних клієнтів стартап-проекту представлені в таблиці 4.4 та таблиці 4.5 відповідно. Проводиться визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів.

Таблиця 4.4 Попередня характеристика потенційного ринку стартап-проекту

№	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	10-15
2	Загальний обсяг продаж, грн/ум.од	15000 грн/ум.од.
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу	Недискримінаційні якісні

5	Специфічні вимоги до стандартизації та сертифікації	Відстуні
6	Середня норма рентабельності в галузі (або по ринку), %	70%

Таблиця 4.5 Попередня характеристика потенційного ринку стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Потреба створення децентралізованих додатків	Як для персонального використання так і для корпоративних клієнтів	Важлива безпека даних; дотримання умов доступності	Легкість використання; низька ціна за послуги

Фактори загроз та фактори можливостей представлені в таблиці 4.6 та в таблиці 4.7 відповідно.

Таблиця 4.6 Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Отримання несанкціонованого доступу сторонніми особами	Хакерська атака може призвести до викрадення даних клієнтів	Не є проблемою оскільки, оскільки платформа побудована на принципі відкритих даних і вона є децентралізованою,

			тож кожен клієнт несе відповідальність за збереження своїх персональних
2	Відсутність ринку	Відсутність шляху збуту товару внаслідок помилкового орієнтування	Ретельний розгляд проблем потенційних клієнтів Консультації із спеціалістами
3	Програмні помилки та несправність системи	У процесі розробки виникли програмні помилки, які призводять до некоректної роботи додатків	Затримка релізу проекту, продовження моделювання процесів та вдосконаленя кодової бази

Таблиця 4.7 Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Отримання інвестицій	Отримання капіталу що необхідний для реалізації продукту	Не потребує капіталу для реалізації

2	Успішна маркетингова політика	В результаті проведеної маркетингової політики отримана висока зацікавленість користувачів	Продовження роботи, розробка нового функціоналу, який пропонують зацікавлені користувачі

Ступеневий аналіз конкуренції на ринку та аналіз конкуренції в галузі за М. Портером представлені в таблицях 4.8 та 4.9 відповідно. Конкурентний аналіз не спрямований на визначення можливостей, загрози і відшукування стратегічних невизначеностей, що можуть створюватися конкурентами, оскільки результати роботи знаходяться в відкритому доступі і питання конкуренції не постає.

Таблиця 4.8 Ступеневий аналіз конкуренції ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Чиста	Досить багато рішень для створення децентралізованих додатків	Інноваційність рішення, надання послуг вищого класу
Міжнародна	Багато конкурентів працюють на міжнародній арені	Зменшення охоплення ринку
Внутрішня	Боротьба між компаніями, які спеціалізуються на створенні децентралізованих додатків	Створення більш якісної платформи
Товарно-видова	Надання різних сервісів одного типу	Збільшення маркетингових компаній
Цінова	Використання цін для покращення економічних умов збуту	Результати дослідження безкоштовні

Не марочна	Не має конкретної прив'язки до марки	Співпраця з потенційно вигідними партнерами

Таблиця 4.9 Аналіз конкуренції в галузі за М.Портером

Прямі конкуренти в галузі	Ethereum, Hyperledger, Corda	Інтенсивна конкуренція, монополізація ринку збуту
Потенційні конкуренти	Можуть мати більш іноваційні рішення	Є вірогідність виходу у міжнародну арену
Постачальники	Відсутні	Відсутні
Клієнти	Клієнти потребують рішення для створення децентралізованих додатків	Клієнти диктують якість забезпечення сервісу, його доступність
Товари-замінники	Копіювання загального функціоналу	Пропонування вигідних умов

Обґрунтування факторів конкурентоспроможності представлений в таблиці 4.10

Таблиця 4.10 Обґрунтування факторів конкурентоспроможності

№	Фактор конкуренто спроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим
1	Унікальність сервісу	Розроблений продукт представляє унікальну платформу для створення децентралізованих додатків, завдяки реалізацій оптимальних алгоритмів, дозволяє легко створювати та разгортати додатки
2	Цінова політика	Отримання прибутку за рахунок розробки додаткового функціоналу та додатків на базі платформи

Порівняльний аналіз сильних та слабких сторін платформи представлений в таблиці 4.11

Таблиця 4.11 Порівняльний аналіз сильних та слабких сторін платформи

№	Фактор конкуренто спроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні із платформою						
			-3	-2	-1	0	+1	+2	+3
1	Унікальність сервіс	15						+	
2	Цінова політика	12		+					

SWOT-аналіз стартап-проекту представлений в таблиці 4.12

Таблиця 4.12 SWOT-аналіз стартап-проекту

Сильні сторони: Якість та довготривалість Безкоштовність	Слабі сторони Малий прибуток
Можливості Інвестиції Придбання платформи	Загрози Викрадення коду платформи

Альтернативи ринкового впровадження стартап-проекту представлені в таблиці 4.13

Таблиця 4.13 Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Розробка тестового демонстраційного проекту	Висока	6 місяців
2	Маркетингова кампанія для приваблювання користувачів	Низька	3 місяці

3	Співпраця з більш сильнішими конкурентами	Висока	Від одного року

Найбільш вигідною альтернативою є надання консультації з приводу проектування та розробки платформи для створення децентралізованих додатків.

4.4 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (таблиця 4.14).

Таблиця 4.14 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
Опенсорс розробники	Висока	80%	Присутня	Низькі бар’єри входу
Фінансові компанії	Середня	50%	Присутня	Високі бар’єри входу
Академічне застосування	Низька	30%	Відсутня	Низькі бар’єри входу
Які цільові групи обрано: опенсорс розробники, фінансові фірми та академічне застосування				

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (таблиця 4.15).

Таблиця 4.15 Визначення базової стратегії розвитку

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

Відкрита розробка у опенсорс товаристві	Вибірковий розподіл	Здатність протистояти конкурентам за рахунок швидкого розвитку, ефективна співпраця	Стратегія диференціації

Визначення базової стратегії конкурентної поведінки та визначення стратегії позиціонування представленів таблиці 4.16 та в таблиці 4.17 відповідно.

Таблиця 4.16 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	Ні	Ні	Розширення первинного попиту

Таблиця 4.17 Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
Надійність платформи Відповідність функціонала специфікації платформи Безкоштовний	Стратегія диференціації	Постійний розвиток платформи	Безкоштовний Інноваційний Простий

4.5 Розроблення маркетингової програми стартап-проекту

Визначення ключових переваг концепції потенційного товару представлені в таблиці 4.18.

Таблиця 4.18 Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Потреба в реалізації більш обширного набору бібліотек	Користувач зможе створювати більш гнучкі та оптимізовані додатки	Простота та зручність у використанні
2	Підвищення продуктивність платформи	Оптимізація та підвіщення швидкості роботи додатків	Інноваційність

Опис трьох рівнів моделі товару представлені в таблиці 4.19

Таблиця 4.19 Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>Товар за задумом</td><td colspan="3">Платформа для створення децентралізованих масштабованих додатків</td></tr>
<tr><td rowspan="5"></td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>Кількість</td><td></td><td>1 шт.</td></tr>
<tr><td colspan="3">Якість: стандарти якості написання технічної документації</td></tr>
<tr><td colspan="3">Пакування: відстунє</td></tr>
<tr><td colspan="3">Марка: Відсутня</td></tr>
</table>

Товар за підкріпленям	Тестовий програмний продукт
	Програмний продукт, технічна підтримка
За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності від копіювання відсутній	

Визначення меж встановлення ціни представлені в таблиці 4.20

Таблиця 4.20 Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової груписпоживачів	Верхня та нижня межі встановлення ціни на товар/послугу
0 грн	0 грн	0 – 3000000 грн/міс	0 – 0 грн

Формування системи збуту представлено в таблиці 4.21

Таблиця 4.21 Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Закупівля не здійснюється	Інформування користувачів шляхом публікації статей	Канал одного рівня	Селективна з використанням комбінованого каналу збуту

Концепція маркетингових комунікацій представлена в таблиці 4.22

Таблиця 4.22 Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Знайомі з існуючими рішенями та їхніми первагами	Веб-додатки	Інноваційність рішення, швидкість та простота роботи	Висвітлення переваг над конкурентами	Реклама на таргетних платформах та публікації статей

ВИСНОВКИ

З кожним роком на ІТ ринку зростає попит на розробку додатків. Для розробки програми доступні численні технічні стеки, та мови, але не існує однієї єдиної технолгії, яка надає найкращі результати. Кожна технологія пропонує свій підхід для створення додатків. Проте більшість сучасних додатків проектуються за клієнт-серверною архітектурою, а це накладає деякі обмеження на розробку, зокрема необхідність матеріальної підтримки серверної частини. Проте існує підхід до розробки, який дозволяє розгортати додатки без серверної архітектури, у децентралізованій мережі. В роботі вирішена актуальна проблема проектування та розробки платформи для створення децентралізованих додатків.

Реалізована платформа для створення блокчейн додатків є сукупністю програмних інструментів, використання яких призведе до економії часу на розробку додатків, знизить затрати ресурсів на розгортання до мінімуму та дозволить створювати високоефективні додатки, які не будуть поступатися загальноприйнятим клієнт-серверним.

Для досягнення мети дослідження, яка полягає у дослідження та реалізації концепцій для створення гнучкої платформи для розробки швидких децентралізованих додатків, були вирішені наступні завдання:

– розглянуто поняття про платформи для створення децентралізованих додатків;

– досліджений устрій спроектованої системи для розробки блокчейн додатків;

– описана реалізація запропонованих концепцій;

– наведені результати випробовування розробленого рішення;

– розроблено стартап-проект.

В ході виконання магістерської дисертації було зроблено такі висновки:

– для реалізації оптимальної платформи, вона має складатися із таких підсистем:

– підсистема протокола консенсусу, для швидкого обміну та збереження даних;

– підсистема протоколу пошуку вузлів, для стабілізації децентралізованої системи;

– підсистеми механізмів використання ресурсів додатками (виконання накладних I/O операцій);

– для оптимального використання ресурсів, слід використовувати:

– програмні віртуальні машини для виконання накладних за ресурсами операцій;

– механізми моніторингів станів файлових дескрипторів для ефективного виконання I/O операцій;

– впровадження платформи у практичне використання повинно супроводжуватися та підкріплюватися запровадженням відповідного стартап-процесу.

Загальним результатом розробки платформи для створення блокчейн додатків має стати реалізація відповідного стартап-проекту на базі отриманих результатів дослідження та виведення відповідного рішення на ринок інструментів для створення децентралізованих додатків. Розробка стартап-проекту допомогла краще осягнути рамки розроблюваного рішення, його недоліки та переваги, якими він повинен володіти. Це було закріплено у відповідних таблицях розділу.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) ethereum · GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ethereum

2) EthereumHotHot [Електронний ресурс] – Режим доступу до ресурсу: https://www.reddit.com/r/ethereum/

3) Hyperledger - Open source blockchain for business - IBM Blockchain| IBM [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/blockchain/hyperledger

4) Welcome to Corda ! — R3 Corda Master documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.corda.net/

5) Quorum | Software Built for Public Affairs [Електронний ресурс] – Режим доступу до ресурсу: https://www.quorum.us/

6) Simple Virtual Machine [Електронний ресурс] – Режим доступу до ресурсу: https://bartoszsypytkowski.com/simple-virtual-machine/

7) Building a stack-based virtual machine - DEV Community [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/jimsy/building-a-stack-based-virtual-machine-5gkd

8) oop - What are object serialization and deserialization? - Stack Overflow [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/questions/1360632/

9) The JavaScript Event Loop: Explained [Електронний ресурс] – Режим доступу до ресурсу: https://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/

10) Remote Procedure Call (RPC) in Operating System - GeeksforGeeks [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

11) RESTful API – (advanced) Projects [Електронний ресурс] – Режим доступу до ресурсу: https://www.linux-projects.org/documentation/rest-api/

12) Protocol Buffers|Google Developers [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/protocol-buffers

13) The Rust Programming Language · GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/rust-lang

14) Continuous Architecture / Pierre Pureur, Murat Erder // Elsevier, – October 2015. – с. 223–289.

15) Software Architecture Fundamentals / Gernot Starke, Andreas Rausch, Mahbouba Gharbi, Arne Koschel // dpunkt, – February 2019. – с. 194–224.

16) Streaming Architecture / Ted Dunning, Ellen Friedman // O'Reilly Media, – May 2016. – с. 62–73.

17) epoll: I/O event notification facility - Linux Man Pages (7) [Електронний ресурс] – Режим доступу до ресурсу: https://www.systutorials.com/docs/linux/man/7-epoll/

18) JavaScript Event Loop Explained - Frontend Weekly - Medium [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4

19) POA Network: How Honey Badger BFT Consensus Works - POA Network - Medium [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/poa-network/poa-network-how-honey-badger-bft-consensus-works-4b16c0f1ff94

20) Hbbft Audit Report | Cryptography | Application Programming Interface [Електронний ресурс] – Режим доступу до ресурсу: https://www.scribd.com/document/397772610/Hbbft-Audit-Report

21) poanetwork/hbbft [Електронний ресурс] – Режим доступу до ресурсу: https://www.gitmemory.com/poanetwork/hbbft

22) Gossip Protocol - Consul by HashiCorp [Електронний ресурс] – Режим доступу до ресурсу: https://www.consul.io/docs/internals/gossip.html

23) Gossip data dissemination protocol — hyperledger-fabricdocs master documentation [Електронний ресурс] – Режим доступу до ресурсу: https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html

24) gossip-protocol · GitHub Topics · GitHub [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/topics/gossip-protocol