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

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено

Завідувач кафедри

_____ О.В. Коваль

(підпис) (ініціали, прізвище)

“___”_____2020р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 121 Інженерія програмного забезпечення

на тему «Розробка апаратно – програмного комплексу управління розподіленою мережею сенсорів»

Виконав (-ла): студент (-ка) 4 курсу, групи ТВз-61

Лукирич Олег Сергійович _____

(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к. т. н., Ковальчук Артем Михайлович _____

(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доцент к. т. н., Баранюк Олександр Володимирович _____

(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень (бакалаврський)

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ

на дипломну роботу студента

Лукирича Олега Сергійовича
(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка апаратно – програмного комплексу управління розподіленою мережею сенсорів»
керівник роботи Ковальчук Артем Михайлович, к. т. н., доцент
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від «25» травня 2020р. №1168-с

2. Строк подання студентом роботи «10» червня 2020 р.

3. Вихідні дані до роботи мова Go; платформа: Go (середовище виконання), Raspbian (операційна система), Raspberry Pi Zero W (апаратна платформа).

4. Зміст роботи ідентифікувати проблеми клієнт-серверної архітектури для розподілених систем (систем сенсорів або подібних), змоделювати цільову систему та поставити до неї вимоги, проаналізувати проблеми клієнт-серверної архітектури та альтернативні методи, створити архітектуру для цільової системи на основі альтернативних методів (усуваючи проблеми клієнт-серверної моделі), розробити апаратно-програмне рішення за створеною архітектурою.

5. Перелік ілюстративного матеріалу мета та підстави для розробки, опис цільової системи, огляд типового рішення, огляд запропонованого підходу, застосовані методи, стек технологій, варіанти використання, загальна схема роботи, діаграми послідовностей, діаграма класів, висновки.

6. Консультанти розділів роботи

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

7. Дата видачі завдання «20» лютого 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	20.01.2020 р.	
2.	Вивчення та аналіз задачі	01.03.2020 р. – 20.03.2020 р.	
3.	Розробка архітектури та загальної структури системи	20.03.2020 р. – 25.04.2020 р.	
4.	Розробка структур окремих підсистем	25.04.2020 р. – 10.05.2020 р.	
5.	Програмна реалізація системи	10.05.2020 р. – 25.05.2020 р.	
6.	Оформлення пояснювальної записки	25.05.2020 р. – 10.06.2020 р.	
7.	Захист програмного продукту	10.06.2020 р.	
8.	Передзахист	10.06.2020 р.	
9.	Захист	17.06.2020 р.	

Студент _____ Лукирич О.С.
(підпис) (прізвище та ініціали,)

Керівник роботи _____ Ковальчук А.М.
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

В даній роботі досліджуються методи розробки розподілених систем, що використовують інформацію про середовище для прийняття узгоджених між вузлами рішень в умовах обмежених апаратних ресурсів та можливості збоїв. В якості відправної точки моделюється цільова система такого типу та ставляться до неї вимоги – надалі метою є аналіз та порівняння методів для її розробки, створення оптимальної архітектури, та відповідної реалізації.

Аналіз починається з типової для цільової системи клієнт-серверної архітектури, її недоліків та шляхів їх усунення, та врешті-решт переходить до однорангової архітектури на основі реплікованого кінцевого автомату з застосуванням консенсусу, акцентовано розглядаючи алгоритми консенсусу. Було створено архітектуру на основі Raft, та описано, розроблено і протестовано її реалізацію. Обрана архітектура має високу толерантність до збоїв окремих вузлів, автономність, мобільність, добру продуктивність та економність, та дозволяє використовувати переваги близького розміщення вузлів в мережі.

Роботу виконано на 67 аркушах, він містить перелік посилань на використані джерела з 19 найменувань. У роботі наведено 19 рисунків.

Ключові слова: peer-to-peer, розподілені системи, реплікований кінцевий автомат, консенсус, Raft, кіберфізичні системи.

ABSTRACT

This work presents a research on distributed systems that use environment information to make consistent decisions across all nodes – in spite of limited hardware and possibility of individual node failures. As a starting point, a target system of this kind is modelled, and the goal is set to analyze and compare the methods of implementation, synthesize them into an optimal architecture, and develop a prototype based on it.

The analysis begins from the typical for a such system client-server architecture, it's drawbacks and ways to improve on them. Gradually, it shifts towards a peer-to-peer architecture, based on replicated state machine with consensus, especially focusing on consensus algorithms. The chosen architecture is based on Raft, and the respective prototype has been described, developed and tested. The architecture has high tolerance to individual node failures, autonomy, mobility, good performance and resource efficiency, and enjoys the benefits of node proximity in the network.

This work is executed on 67 pages, contains 19 references to external sources, and 19 pictures.

Key words: peer-to-peer, distributed systems, replicated state machine, consensus, Raft, cyberphysical systems.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Розподілена система – група систем, де кожна окрема система виконує дії з огляду на поведінку інших, з метою функціонування як єдине ціле для зовнішнього спостерігача.

Peer-to-peer – принцип організації розподілених систем, за яким підсистеми є функціонально ідентичними та взаємозамінними, хоча й можуть виконувати різні ролі в різні часи.

Бек-енд – сукупність компонентів системи, що з точки зору даного компонента розглядаються як чорний ящик, що надає йому функціональність. Наприклад, з точки зору веб-інтерфейсу бек-ендом є веб сервер, база даних і т.д.

RPC – Remote procedure call. Принцип побудови прикладних мережевих інтерфейсів, за яким кожному елементу інтерфейсу відповідає звичайна функція, де клієнт кодує виклик такої функції з параметрами, відправляє серверу, а той у відповідь кодує значення, що та функція повертає.

ВСТУП

Управління в розподілених системах, зокрема кіберфізичних, займає все важливіше місце у сучасних комп’ютерних науках. Попит на відповідні технології має найрізноманітнішу природу: від побутових використань до воєнної інфраструктури. Сучасна цивілізація звикла сприймати здатність нарощувати обчислювальний потенціал за необхідності як належне, але це не може тривати вічно: прогрес у розробці інтегрованих комп’ютерів вже є не таким разючим як колись, виходячи на плато; щоб це компенсувати, можна покращувати методи ефективного використання великої кількості окремих комп’ютерів як одне ціле – горизонтального масштабування. Зростають також очікування щодо автономності, мобільності та надійності систем – характеристики, що можуть бути покращені розподіленням та децентралізацією.

Виробникам комп’ютерів стабільно вдається створювати дешевші, компактніші та масовіші продукти, відкриваючи доступ до практики з розподіленими апаратно-програмними комплексами спеціального призначення все ширшій аудиторії, стимулюючи любительську та навчальну розробку, бізнес сектор, цілі індустрії, та перетворюючи колись «сміливі» ідеї інтернету речей та розумного міста у звичну буденність. Як результат, у майбутньому слід очікувати велику кількість застосунків вузького призначення, часто обмежених апаратно, які регулюватимуть повсякденне життя, в рамках якого в тому числі постають проблеми, які можна розв’язати тільки одночасною фізичною присутністю системи в багатьох місцях та цілісною координацією її екземплярів. Тому, дослідникам обчислювальних наук та інженерам-програмістам слід звертати увагу на покращення методів розробки таких систем.

Дана робота приймає участь в цій екосистемі, пропонуючи практичний приклад застосування реплікованого кінцевого автомата та консенсусу в управлінні розподіленою мережею апаратно-програмних застосунків, що приймають узгоджені колективні рішення, керуючись інформацією про локальне середовище, в умовах

обмежених обчислювальних ресурсів та можливості збоїв окремих вузлів.

Основна суть роботи полягає у нестандартному дизайні, де замість превалюючої на сьогодні для подібних задач клієнт-серверної архітектури, перевага надається одноранговій архітектурі, яка має кращі характеристики в багатьох вимірах, що є наслідком усунення централізації.

1. ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд предметної області та мотивація рішення

Уявімо розподілену систему, що складається з фізичних пристроїв, які мають сенсори, що збирають показники навколишнього середовища, та уявімо, що кожен пристрій має якось діяти в залежності від цих показників. У простому випадку, контекст цієї дії повністю ізольований в рамках одного пристрою – наприклад, сигналізувати тривогу, якщо склад повітря містить достатньо високу концентрацію чадного газу. Іноді ж буває, що така дія вимагає кооперації між пристроями – вона є функцією від показників сенсорів їх усіх.

Розглянемо в якості прикладу розподілену систему охолодження повітря, де у кожного пристрою є термометр (для простоти відкинемо вологість та інші фактори) та охолоджувач. Уявімо, що у користувача є можливість встановити загальний ліміт енергоспоживання для усієї системи – може бути корисно, наприклад з точки зору економії.

Базова поведінка пристрою – чим вища температура на термометрі, тим більше витрачати енергії на охолодження. Але за існування загального ліміту енергоспоживання така поведінка є наївною і легко призведе до його перевищення; жоден алгоритм, що не покладається на комунікацію з іншими пристроями, враховуючи їхню ситуацію, тут не працює. Крім високого паралелізму, така система має враховувати можливість збоїв та мережевих неполадок: це робить задачу узгодження між пристроями бачення усіх показників сенсорів в кожен момент часу, що необхідно для прийняття рішення, складною.

На сьогодні, найпоширенішим підходом до цієї проблеми є використання клієнт-серверної архітектури: пристрої відправляють показники сенсора серверові, той на їх основі обраховує оптимальну стратегію, пристрої запитують її у сервера та

застосовують локально – не приймаючи жодних рішень самостійно. Це дозволяє обійти згадану складність, адже сервер є інтегральною системою (часто є просто процесом додатку ОС в єдиному екземплярі), в якій паралелізм легко контролювати і надійність засобів передачі інформації – близька до ідеальної; але й створює цілий ряд проблем через централізацію управління: від сервера залежить все. Крім технічних (наприклад, збій сервера означає збій усієї системи, чутливість до якості мережі) є ще й ряд гуманітарних наслідків, спричинених необхідністю підтримувати сервер (детальніше у п. 2.1).

Тому, завдання – розробити альтернативний дизайн, що матиме кращі характеристики, та прототип, що демонструє його у дії.

Для спрощення опису, аналізу та реалізації, та водночас застосовності результатів даної роботи до ширшого ряду прикладних проблем, вищезгаданий приклад з системою охолодження повітря далі оформлено у вигляді абстрактної моделі.

1.2 Модель проблеми та функціональні вимоги

Маємо $Q \in Z^+$ вузлів (пристроїв) $Node_i$, де $i \in Z^+, 0 \leq i < Q$ – порядковий номер вузла*)*. Кожен вузол має динамічний показник середовища I_i (локальний показник термометра), та параметр O_i (локальний ліміт енергоспоживання).

Вузли мають певний спільний ресурс константного розміру R (загальний ліміт потужності); O_i є часткою R. Кількість вузлів Q може бути змінною. Частота зміни I_i має вважатися стільки завгодно високою. Вузли об’єднані мережею, спілкуються між собою, виконуючи спільний протокол, метою якого є періодичне, якнайчастіше оновлення оптимальних значень O_i в залежності від значень I_i, Q та R наступним чином:

- $\sum_{i=0}^{Q} O_i \leq R$ – *Прим 1.1*

- O_i є прямо-пропорційним I_i, і має бути настільки великим, наскільки дозволяє R та поточне значення O_i інших вузлів.

Акт встановлення O_i має вважатися завершеним іншими вузлами тільки після підтвердження цього самим вузлом $Node_i$, факту отримання $Node_i$ інструкції встановити значення O_i недостатньо: допускається довільна затримка виконання цієї інструкції.

Якщо вузол вийшов з комунікації, останні відомості про його O_i мають вважатися актуальними іншими вузлами, допоки той не повернеться та комунікує інше значення.

Необхідно передбачити засоби побудови групи: створення нової групи, додавання вузлів до неї.

Вузлом має виконуватися журналювання подій (з часовою міткою) встановлення O_i у локальний файл, за результатами якого можна буде оцінити коректність його поведінки і системи загалом, перевіряючи, що умова з прим. 1.1 дотримається в кожен момент часу.

1.3 Нефункціональні вимоги

Рішення має мати форму прототипу апаратно-програмного комплексу: кожному вузлу має відповідати реальний фізичний пристрій зі встановленим програмним забезпеченням. Характеристики пристрою (архітектура процесора, тощо) мають відповідати реаліям інтернету речей. Відповідно, програмне забезпечення має задовольнятися ресурсами апаратної платформи, які мають очікуватися доволі обмеженими. Пристрої мають бути об’єднуваними бездротовою мережею та передбачати локальність розміщення (наприклад, знаходяться у одному приміщення офісного центру). Вимоги щодо безпеки інформації (шифрування, контроль доступів, тощо) не ставляться.

Для аналізу гуманітарних аспектів, припустімо що у системи є виробник, вона є продуктом, яким користується довільна кількість клієнтів різного масштабу. Виробник несе тривалу відповідальність за систему перед клієнтом – має її підтримувати. Відповідно, рішення має намагатися полегшити підтримку.

Висновки до розділу 1

Було розглянуто клас систем, що колективно приймають рішення на основі інформації про локальне середовище. Було ідентифіковано недоліки типового підходу до побудови таких систем, та поставлено задачу їх усунути в рамках цільової системи. Було змодельовано цільову систему, поставлено до неї функціональні та нефункціональні вимоги.

2. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ТЕОРЕТИЧНИХ ТА ПРАКТИЧНИХ МЕТОДІВ

2.1 Типова архітектура – клієнт-сервер

Існує центральний сервер, $Node_i$ з певною частотою відправляє йому I_i та O_i, а сервер повертає O_{next_i} – бажане значення O_i для встановлення в майбутньому; мається на увазі, що отриманий O_{next_i} клієнт $Node_i$ встановить рано чи пізно, і надалі передаватиме його в якості O_i.

Може бути декілька варіацій клієнт-серверної взаємодії. Відправлення I_i та O_i, може здійснюватися двома окремими запитами, повернення O_{next_i} може бути здійснено окремим запитом без аргументів, а може бути об'єднано в з будь-яким запитом або запитами, що відправляють I_i та O_i.

Розрахунок $O_{next} = (O_{next_0} \dots O_{next_{q-1}})$ може відбуватися на сервері синхронно з запитами периферії, що передають локальні показники (запит провокує перерахунок і O_{next_i} повертається виключно після закінчення перерахунку; між такими запитами існуватиме конфлікт, що призводить до взаємного блокування), або асинхронно: сервер періодично перераховує O_{next} у фоні, фіксуючи результат з якого надалі братиметься O_{next_i} для повернення клієнту $Node_i$.

Надалі, розглядатиметься схема, де є всього один тип запиту. Для зручності, визначимо $NodeState_i = (i, I_i, O_i)$. Клієнт $Node_i$ передає серверу $NodeState_i$ та отримує O_{next_i}. Зведення взаємодії до одного запиту є корисним з точки зору продуктивності, порівняно з розбиттям на декілька запитів, адже це суттєво зменшує використання мережі, ресурсів сервера; мережеві затримки даватимуться взнаки менше. Також, один запит зручніше аналізувати ніж декілька. Така схема дозволяє обраховувати O_{next_i} синхронно чи асинхронно без змін API. Надалі матиметься на увазі асинхронна варіація розрахунку, адже вона є неблокуючою і вимагає значно

менше обчислень. Стан системи загалом характеризується $SystemState = \left((NodeState_0 \dots NodeState_{q-1}),\ O_{next}\right)$. Схема зображена на рис. 2.1.

«Рисунок 2.1 – Клієнт-серверна модель»

2.2 Недоліки

Основним недоліком такої архітектури є централізація. Проблеми з якістю роботи сервера безпосередньо відображаються на системі загалом. Якщо сервер припиняє обслуговування запитів, система в цілому не може досягати прогресу. Існує чутливість до якості мережі.

Загалом, недоліком є і сама необхідність мати справу з сервером як окремою сутністю: наприклад, така архітектура вимагає розробки сервера та розробки периферійних систем в окремому порядку, що може бути небажаним для виробника. Сервер необхідно підтримувати, що створює складність, ризики та витрати.

Системи, що мають підвищені вимоги до локальності та автономності, особливо важко сумістити з центральним контролером. Прикладом може бути флот дронів, що здійснює колективну автоматичну навігацію: пропускна здатність та затримка відіграє значну роль в комунікації; доступність сервера може бути проблемою в польових умовах. Хоч значно слабкіше, але це спостереження застосовне і до системи кондиціонування, яка наразі розглядається.

З точки зору користувача, сервер може бути або на боці виробника, або на його боці. В останньому випадку, це може бути або базова станція, або окремий пакет ПЗ

(on-premise), що користувач встановлює на власну обчислювальну інфраструктуру та конфігурує.

2.2.1 Сервер на боці виробника

Сервер на боці виробника – є зручним варіантом для користувача, але змушує першого підтримувати серверну інфраструктуру весь час існування продукту.
Для виробника це є значним ускладненням, та має негативні економічні наслідки: вимагає витрати на працю адміністраторів, електроенергію, телекомунікаційні послуги, оренду приміщення, придбання, ремонт та модернізацію обладнання, тощо. Це може бути полегшено за допомогою використання "хмари", але і тут є недоліки (наприклад, менше контролю, потенційна неможливість вдало розмістити сервери територіально для покращення мережевої комунікації), і загалом не завжди це є оптимальним вибором з точки зору бізнесу. Також, серверна інфраструктура є вразливою до людської помилки, природних факторів, проблем з зовнішньою інфраструктурою (міська електромережа, інтернет-провайдер, тощо).

Обчислювальні потужності сервера мають масштабуватися зі збільшення активних одиниць периферійних пристроїв, що використовують кінцеві користувачі: з їх збільшенням росте кількість запитів на сервер, та обсяг даних, що той має зберігати та на основі яких обчислювати O_{next}.

У даній варіації сервер має встановлювати відповідність між запитом периферійного пристрою та реальним приміщенням, що додає складність в розробці та конфігурації периферійних пристроїв (лягає на плечі виробника або користувача, або обох).

Ціна похибки тут висока, адже екосистема продукту повністю спирається на сервер.

2.2.2 Сервер на боці користувача

Міркування щодо сервера на боці користувача здебільшого дзеркальні. Не кожен користувач бажає чи може обслуговувати сервер, особливо індивідуальний побутовий користувач: on-premise варіація є складною технічно, в той час як базова станція є громіздкою: фізичний пристрій, з усіма наслідками. Хоч виробник і позбавляється від відповідних технічних ускладнень, це може бути не варто удару по загальній привабливості продукту (хоч тепер і дешевшого). Якість мережевої комунікації у випадку сервера на боці виробника загалом має вважатися принципово гіршою, адже у протилежному випадку є можливість розмістити сервер у фізичній близькості з периферією. Також, варіація з сервером на боці виробника у загальному випадку вимагає інтернет, на боці користувача – ні.

2.2.3 Реплікований бек-енд

Для підвищення стійкості до збоїв та покращення масштабування потужностей, сервер може бути реплікованим.

На практиці це означає використання можливостей реплікації бази даних, та збільшення кількості процесів власне серверного ПЗ. Основна складність полягає у реплікації бази даних, адже вона несе весь стан системи; серверне ж ПЗ в подібній архітектурі не повинне передбачати власний стан – відповідно, для його реплікації достатньо просто запустити програму та додати її до існуючої групи. Відповідна схема зображена на рис. 2.2.

«Рисунок 2.2 – Реплікований бек-енд»

Це означає, що новий процес має якось дізнатися адресу бази даних, а запити периферійних систем мають почати його досягати – це є завданням service discovery. Часто для цього обирають введення системи-диспетчера, яка знає адреси процесів сервера та бази даних, дозволяючи процесам сервера дістатися до бази даних. Додатково, така система може слугувати єдиним фасадом для периферійних систем, балансуючи обробку запитів між процесами сервера. Нові процеси сервера та бази даних мають опинитися в реєстрі диспетчера – це вимагає додаткової уваги, і передбачає мануальну роботу, тому бажано має бути автоматизованим. Service discovery також може бути реалізовано за допомогою конфігурації – початкового набору параметрів запуску ПЗ. Наприклад, програма сервера на старті вимагає параметр – адресу бази даних. Комбінація конфігурації та диспетчера можуть бути застосовані.

Процеси бази даних теж вимагають service discovery: як правило, ПЗ бази даних включає рішення для цього, але часто все одно доводиться робити надбудову, що задовольняє конкретні потреби проекту.

Наявність декількох вузлів бази даних підвищує толерантність до втрати окремих вузлів, адже передбачається, що функція бази даних все ще буде підтримуватися тими, що залишаються. Найчастіше, ПЗ бази даних передбачає автоматичне корегування поведінки кластера при втраті вузла, але іноді доводиться будувати таке рішення окремо.

Прим. 2.1: Множинність вузлів бази даних, здавалося б, має дозволяти горизонтально масштабувати потужності, розподіляючи обробку запитів поміж

вузлами – але є виключення. Багато класичних реляційних баз даних принципово на це не здатні: лише один вузол в момент часу обробляє запити (master), та виключно після обробки запиту надсилає до інших вузлів (slave) інструкції локально виконати ідентичні зміни. Пропускна здатність такої системи рівна пропускній здатності одного вузла. Прикладом є PostgreSQL.

Прим. 2.2: Ускладнюється реалізація асинхронної моделі розрахунку: за яких умов, і який процес (чи декілька процесів) сервера виконуватиме фоновий перерахунок O_{next}, що робити у випадку виходу його з ладу, та як переконатися, що між процесами не буде гонки щодо того, хто в даний момент має це робити – є неповним переліком проблем, що постають.

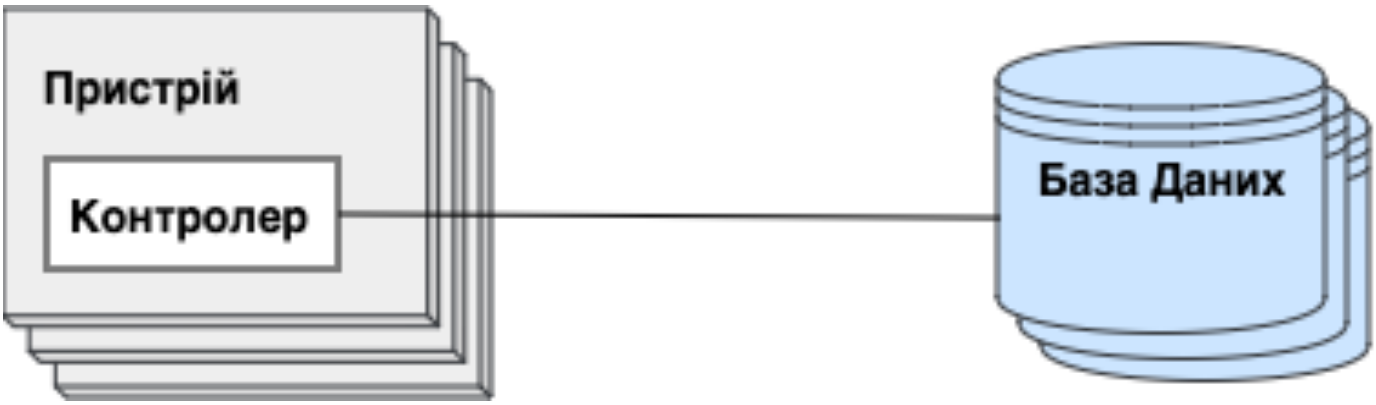
В якості покращення у варіації з сервером на боці виробника, можна розподіляти обробку запитів в залежності від приміщення, що відповідає запиту.

В попередньому аналізі серверу на боці виробника розглядалася лише multi-tenant модель щодо того, як надавати сервіс множині клієнтів: бек-енд є спільним, ідентифікатори клієнтів в запитах використовуються для логічного розмежування. Альтернативою є single-tenant: виробник створює одиницю бек-енду для кожного клієнта окремо. Цей підхід може полегшити або й зовсім усунути проблему масштабування – за рахунок ускладнення підтримки. Звісно, це все ще не працює для масштабних клієнтів, які мають багато активних одиниць периферії, наприклад, міжнародна мережа офісних центрів. Звісно, можна поділити такого клієнта за його локаціями (офісними центрами), але це призводить до ще більшого ускладнення, та заважає оперувати ресурсами клієнта цілісно (уявімо додаткові можливості системи, такі як інвентаризація). В загальному випадку, динамічне горизонтальне масштабування слід закладати.

Складність розробки та супроводу при розподіленій архітектурі бек-енду зростає, і останнє особливо виражається у варіації з сервером на боці користувача.

2.2.4 БД в якості сервера

Можна помітити, що функція сервера може бути розчиненою поміж базою даних та периферійними вузлами (рис. 2.3). Периферійні вузли могли б безпосередньо працювати з базою даних, за умови достатньої підтримки останньою складних атомарних операцій.



«Рисунок 2.3 – ПЗ пристрою спілкується з БД безпосередньо»

Синхронна варіація обробки сервером запиту $Request_i$ може бути трансформована у транзакцію БД, що ініціюється клієнтом з відповідними параметрами.

Для отримання ж більш бажаної асинхронної моделі розрахунку O_{next}, окремі клієнти мають брати на себе додаткову роль фонового обчислювача – постають проблеми, розглянуті у прим. 2.2.

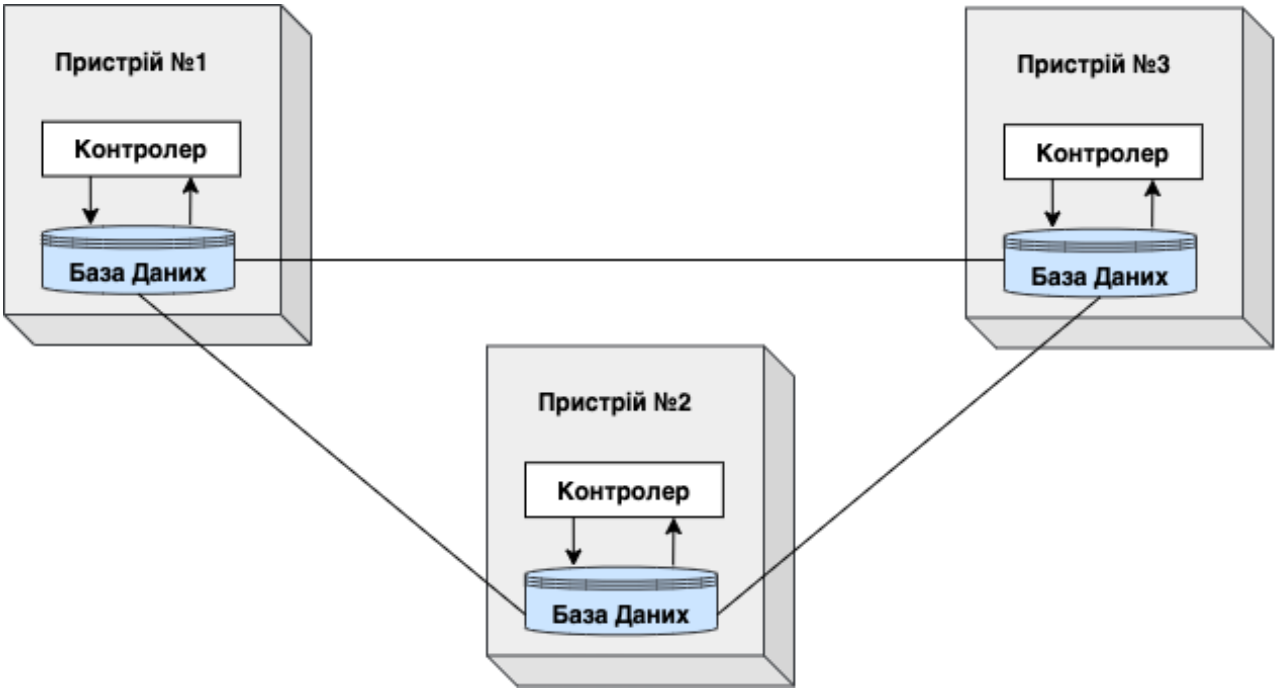
Загалом архітектура дещо спрощується через елімінацію проміжного сервера, хоч рішення на периферії і ускладнюється; але централізація залишається: тепер бек-ендом є база даних. Якщо ПЗ бази даних є достатньо функціональним і простим у використанні та підтримці, це може бути бажаним підходом.

2.3 Одноранговa архітектура

Альтернативою централізації є перенесення функції сервера та БД на периферійні вузли. Кожен периферійний вузол, умовно, стає водночас клієнтом, сервером, та базою даних; передбачається об’єднання їх у групу, що функціонує як одне ціле – така система є одноранговою.

2.3.1 На основі БД

Прямолінійним переходом від архітектури з п. 2.1 до такої схеми є встановлення серверу бази даних поруч з ПЗ периферії на периферійний вузол, де ПЗ периферії працює з локальним сервером бази даних за підходом п. 2.1.5. При цьому, об’єднання вузлів у групу означає об’єднання їх процесів БД у групу (рис. 2.4). Це все ще може означати високий рівень централізації, якщо обрана БД не є достатньо розподіленою, наприклад як у прим. 2.1.



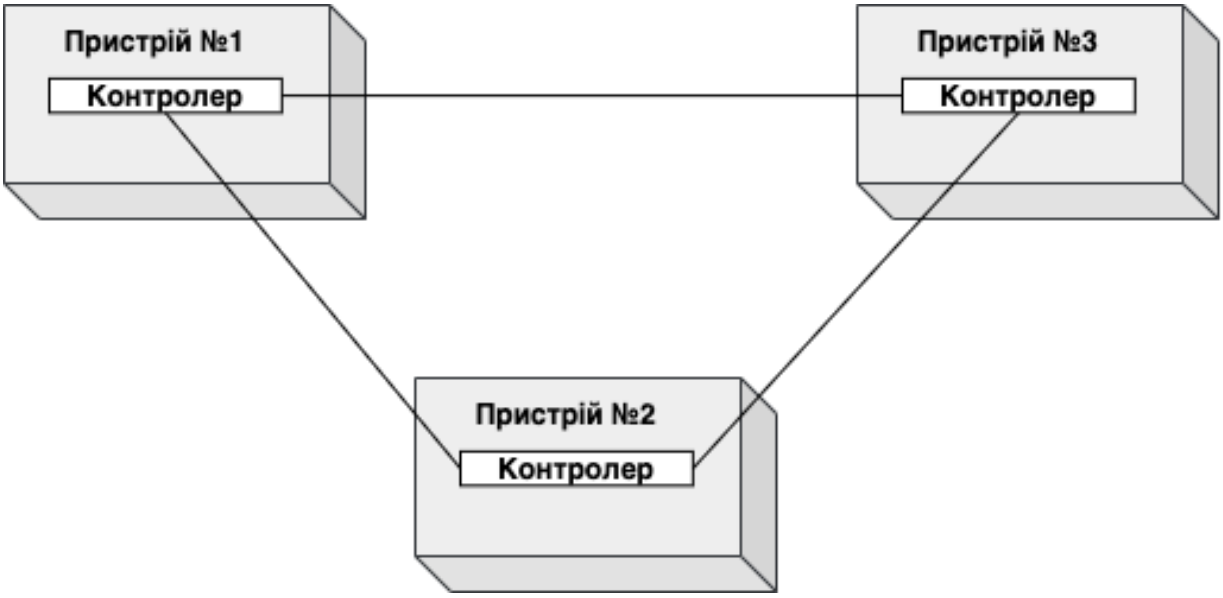
«Рисунок 2.4 – Ко-локація контролера та вузла БД в рамках пристрою»

В будь-якому випадку, мати окремий процес БД на периферійному пристрої може бути небажано, адже обчислювальні ресурси є обмеженими, і розподілена БД може вимагати значний їх обсяг. Існують й інші потенційні ускладнення: наприклад, ПЗ БД може бути несумісним з архітектурою процесора чи ОС пристрою – особливо актуально для вкрай апаратно-обмежених побутових систем.

Відмітимо, що для реалізації функції системи загалом, в БД достатньо зберігати лише актуальний запис $SystemState$, немає необхідності зберігати всю історію значень параметрів системи. Відповідно, БД тут слугує сховищем лише поточного стану, та механізмом узгодження змін цього стану. У випадку такої локалізованої системи, обсяг даних $SystemState$ можна очікувати значно меншим ніж у випадку централізованого бек-енду. Потенціал БД тепер використовується відносно малою мірою, тому можна розглянути розробку того аспекту системи, що вона представляє, з чистого листа – як модуля ПЗ вузла. Від такого модуля можна очікувати кращу ефективність, а його розробка може бути значно полегшена використанням існуючих фреймворків.

2.3.2 З нуля

Уявімо дизайн, де з точки зору ПЗ, кожен вузол є ідентичним, вони об'єднуються в однорідну групу, та, обмінюючись повідомленнями, слідують спільному протоколу (рис. 2.5).



«Рисунок 2.5 – Контролер пристрою повністю визначає поведінку системи»

Без зайвих деталей, програма вузла $Node_i$ полягає у відправленні іншим I_i та O_i, обрахунку $SystemState$ та встановленню $O_i = O_{next_i}$ відповідно до актуального $SystemState$. Алгоритм обрахунку $SystemState$ та встановлення $O_i = O_{next_i}$ можна уявити в якості простої незалежної програми, що може існувати поза контекстом децентралізованої системи. Можна сказати, що вона є кінцевим автоматом, який в кожному стані приймає набір $NodeState_i$ та переходить до наступного стану, де стан повністю описується $SystemState$. Цей автомат можна уявити як основу локальної поведінки вузла. Якщо всі вузли даватимуть на вхід своїм автоматам одні й ті ж аргументи, вони завжди знаходитимуться у когерентному стані – що і є власне метою. Таку модель називають *реплікованим (або розподіленим) кінцевим автоматом.*

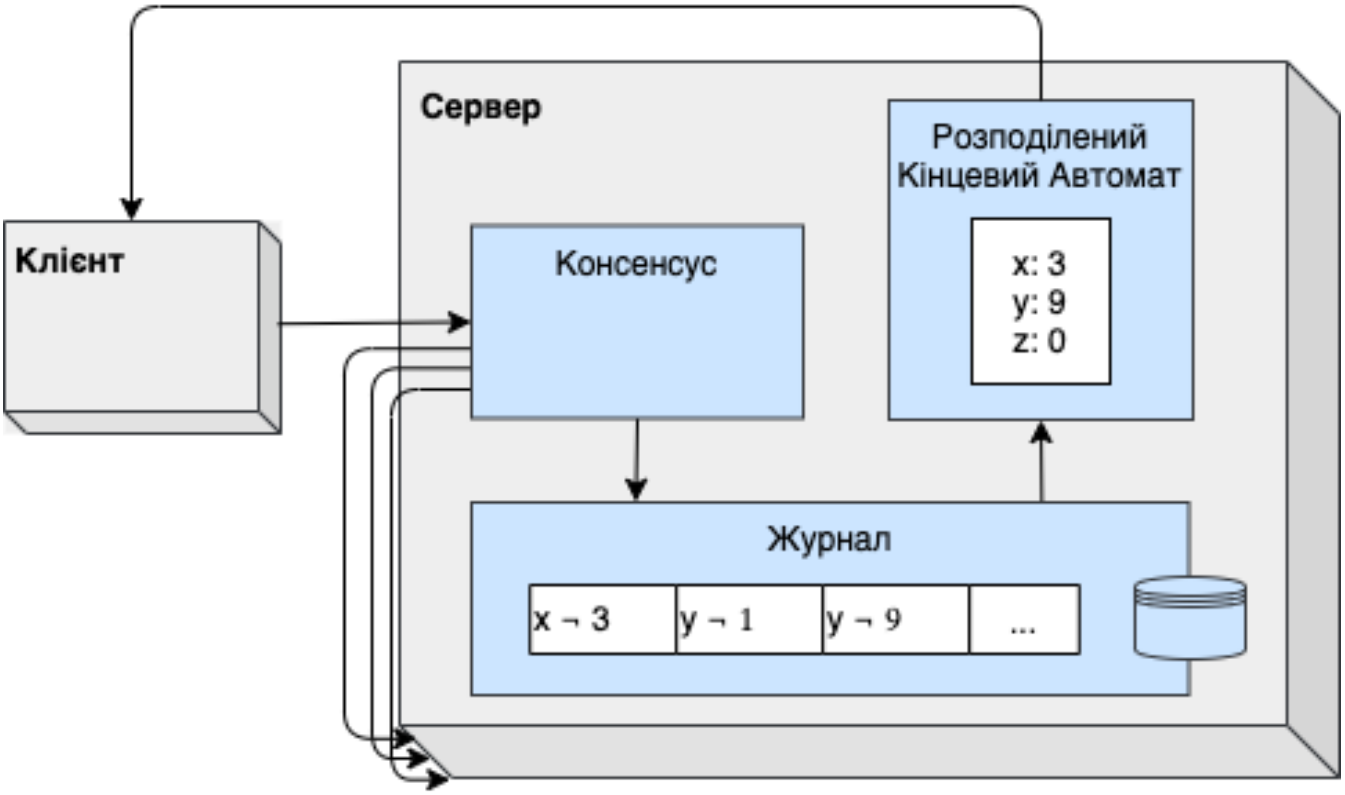
Залишається лише одна проблема: як гарантувати, що всі кінцеві автомати будуть завжди викликані з одними й тими ж параметрами? За нових умов, де немає зовнішнього контролера і комунікація безпосередня, кожен вузол може мати власне бачення стану системи, яке може відрізнятися від інших: які вузли є членами групи в даний момент, які значення локальних показників інших вузлів є актуальними. Ситуація ускладнюється можливістю дисфункції комунікації: окремі вузли можуть

мати збої (наприклад, фізичне пошкодження), можуть виникати проблеми з мережею. Узгодження між вузлами картини стану системи в цілому в умовах можливості збоїв лежить у полі алгоритмів консенсусу.

Розробка такого ПЗ з чистого листа є складною задачею, але за умови існування готових системних рішень для повторного використання – фреймворків – може значно полегшуватися.

2.4 Реплікований кінцевий автомат

Реплікований (або «розподілений») кінцевий автомат [2] – модель системи, у якій кожен вузол виконує одну й ту ж детерміністичну (для одних і тих же параметрів завжди дає один і той же результат) програму, спілкуючись з іншими вузлами, з метою досягання *консенсусу* щодо аргументів для цієї програми. Автомати також можуть мати суто локальні параметри, як наприклад ідентифікатор вузла, та використовувати їх для логіки, як наприклад коли вузол бази даних обслуговує лише ту порцію даних, що відповідає його ідентифікатору – sharding. Стан автомата може впливати на подальшу комунікацію та подальші аргументи. Поштовхом для прогресу – обрання набору аргументів та переходу до наступного стану – може слугувати запит зовнішнього клієнта (наприклад, клієнтський додаток проводить транзакцію розподіленої бази даних) або ж запит може надходити зсередини (наприклад, група автопілотів пропонує одне одному оптимальний маршрут). Ілюстрація на рис. 2.6.



«Рисунок 2.6 – Архітектура розподіленого кінцевого автомату. Сервер – вузол системи, а клієнт може бути як зовнішній, так і внутрішній.»

Мотивацією застосування цієї моделі можуть бути підвищені вимоги до доступності, толерантності до збоїв, продуктивності, безпечності, або ж природа предметної області (наприклад, необхідність контролювати фізично окремі системи як групу). Прикладами є:

- Розподілена файлова система
- Кортеж безпілотних авто
- Блокчейн

Інфраструктура системи на основі розподіленого кінцевого автомата може включати:

- Реконфігурацію – додавання чи видалення вузлів з групи, зміна адрес чи ідентифікаторів вузлів, зміна системних налаштувань, тощо

- Журнал аргументів – містить всю необхідну інформацію для переходу від початкового стану до поточного стану. Може використовуватися для актуалізації нових вузлів. Наприклад, новий вузол блокчейн системи може виконати всю історію транзакцій (згрупованих у блоки) від початку (журнал), який він отримає (та валідує) від інших вузлів, щоб перейти в актуальний стан і почати приймати повноцінну участь.
- Контрольні точки – Якщо природа предметної області дозволяє, вузли можуть домовлятися про компресію журналу станів, досягаючи певних ключових етапів протоколу. Приклад компресії: якщо задача локального автомата – додавати аргумент до локального лічильника, то запис журналу є доданком, і декілька таких записів можна перетворити в один, що містить суму доданків, без втрати інформації.
- Передачу стану – щоб новому вузлу не доводилося обчислювати весь журнал аргументів для отримання актуального стану, інші вузли могли б передати йому стан в готовому вигляді. Наприклад, якщо мова про розподілену базу даних, то новий вузол, замість того щоб з самого початку виконувати всю історію команд з журналу (dump), міг би отримати від інших базу даних у бінарному вигляді. Для багатьох систем це є важливою оптимізацією.

2.5 Консенсус

Консенсус у розподіленій системі – досягнення між її вузлами згоди щодо певного значення; це може бути запис до бази даних, стратегія поведінки, тощо. Консенсус дозволяє системі діяти як одне ціле, де кожен окремий учасник має поведінку інших на увазі, та узгоджений з нею. Прикладами використання консенсусу є:

- Прийняття рішення, чи підтверджувати виконання транзакції бази даних

- Синхронізація годинників
- Оцінка статусу бортових систем
- Обрання лідера для подальшої координації

2.5.1 Загальні положення

Вузли обмінюються повідомленнями. Вузол може *пропонувати* значення іншим, і може *обирати* значення. Відповідно, алгоритми виділяють як мінімум ролі *пропонувач* (або *координатор*) та *обирач* (або *послідовник*). Алгоритм може передбачати єдиного (*лідер*) або багатьох координаторів. Цикли пропозиція-обрання також називають *Раундами.* Вузол може перебувати у стані, в якому він не здатен коректно приймати участь – такий процес є *дефектним.*

Виділяють наступні властивості, що має мати коректний протокол консенсусу:

1. *Узгодженість* – всі не *дефектні* вузли мають *обрати* одне й те ж значення
2. *Скінченність* – кожен не *дефектний* вузол має рано чи пізно *обрати* якесь значення
3. *Цілісність* – значення, що *обирається* має бути *запропонованим* раніше якимось з не дефектних вузлів. Ця умова унеможливлює тривіальні рішеня, такі як де всі вузли завжди *обирають* константу «1» – це задовольняє перші дві властивості, але не має практичної цінності.

Також виділяють дві загальні ознаки коректності розподілених систем:

1. *Безпечність* – система ніколи не входить у такий помилковий стан, який неможливо виправити
2. *Рухливість* – система здатна досягати прогресу рано чи пізно

Узгодженість та *Цілісність* відповідають *Безпечності*, а *Скінченність* – *Рухливості*.

Однорідний консенсус – такий, що має підвищені вимоги до властивості узгодженість: обрані значення усіх вузлів мають бути однакові.

Є декілька типів *дефектності*:

1. *Збій-зупинка* – вузол виходить з ладу, припиняє роботу і більше ніколи не повертається.
2. *Збій-відновлення* – вузол тимчасово припиняє участь, і після довільної затримки повертається та продовжує виконання.
3. *Візантійська поведінка* – включає у себе типи вище, та додатково може означати зловмисну поведінку. Часто під цим поняттям мається на увазі довільна поведінка.

З темпоральної точки зору, комунікація може бути:

1. *Асинхронна* – допускається довільна затримка між відправленням повідомлення та його обробкою; не передбачається впорядкованість пар відправлення-обробка
2. *Синхронна* – пари відправлення-обробка впорядковані, затримка між відправленням та обробкою може бути обмеженою (*тайм-аут*)

Засоби комунікації можуть бути:

1. *Надійні* – повідомлення врешті-решт доставляються, не дублюються чи спотворюються
2. *Ненадійні* – будь-що може трапитися з повідомленнями

Деякі алгоритми розглядають специфічні підмножини ненадійності.

Деякі алгоритми значно покращують свої характеристики (і тому явно специфікують дану техніку), якщо вузли перш ніж надіслати повідомлення мережею журналюють цей намір у надійне локальне сховище. Це називається *логування наперед*, або *WAL – write-ahead logging.* Це допомагає реалізувати сценарії відновлення після збоїв на основі останніх відомостей про прогрес протоколу, як його бачать вузли локально.

Аналізуючи продуктивність алгоритму консенсусу, прийнято розглядати асимптотичну складність, що вимірюється кількістю повідомлень, якими необхідно

обмінятися для одного циклу, в залежності від кількості вузлів; наприклад, якщо складність алгоритму – $O(n^2)$, то n – кількість вузлів, а кількість повідомлень росте квадратично відносно кількості вузлів.

2.5.2 Результат FLP

Так званий результат FLP [3] – одна з ключових теоретичних робіт в області. Будо доведено, що консенсус за повністю асинхронної комунікації є неможливим, якщо хоча б один вузол є дефектним. В моделі системи для доведення є лише два можливих значення – 0 та 1, – допускається єдиний тип дефектності – збій-зупинка, а засоби комунікації вважаються надійними. Було продемонстровано, що така модель має варіант виконання що, порушує умову скінченності. Ключовою проблемою, що витікає з асинхронної моделі комунікації, є неможливість відрізнити процес, що зазнав збою та зупинився назавжди від процесу, що просто повільно працює. Цей результат дисциплінував міркування щодо консенсусу, встановивши ментальний ліміт.

2.5.3 BGP

Важливим моментом в ранній історії досліджень став розгляд BGP [6]. (Byzantine Generals Problem) – проблеми візантійських генералів. За цієї моделі, є група вузлів (генералів), один з них – координатор (командир), всі інші – послідовники (лейтенанти). Генерали намагаються координувати атаку, але деякі з них – зрадники (може бути і сам командир). Командир віддає накази кожному лейтенанту особисто, або «атакувати», або «відступати»; лейтенант може погодитися або відмовитися. Командир-зрадник може не віддати жодного наказу – тоді чесні лейтенанти мають обрати наказ за замовчуванням – «відступати». Отримані від

командира накази лейтенанти також передають одне одному додатково. Зрадники можуть поводитися як завгодно (візантійська поведінка): брехати про зміст наказу, не ретранслювати його зовсім, можуть вступати у змову, тощо. Передбачається повна з'єднаність командирів у мережі. Завдання протоколу зводиться до *інтерактивної консистентності*:

- Всі лояльні лейтенанти підкоряються одному й тому ж наказові
- Якщо командир лояльний, то кожен лояльний лейтенант має підкорятися його наказові

Іншими словами, лояльні генерали мають спілкуватися так, що результат не відрізнявся б від сценарію, де зрадників не існує зовсім. Такий протокол задовольняє загальні властивості коректного алгоритму консенсусу: цілісність, узгодженість, скінченність. Існують також певні допущення щодо системи комунікації:

- Надійність
- Отримувач завжди знає відправника повідомлення
- Відсутність повідомлення може бути поміченою

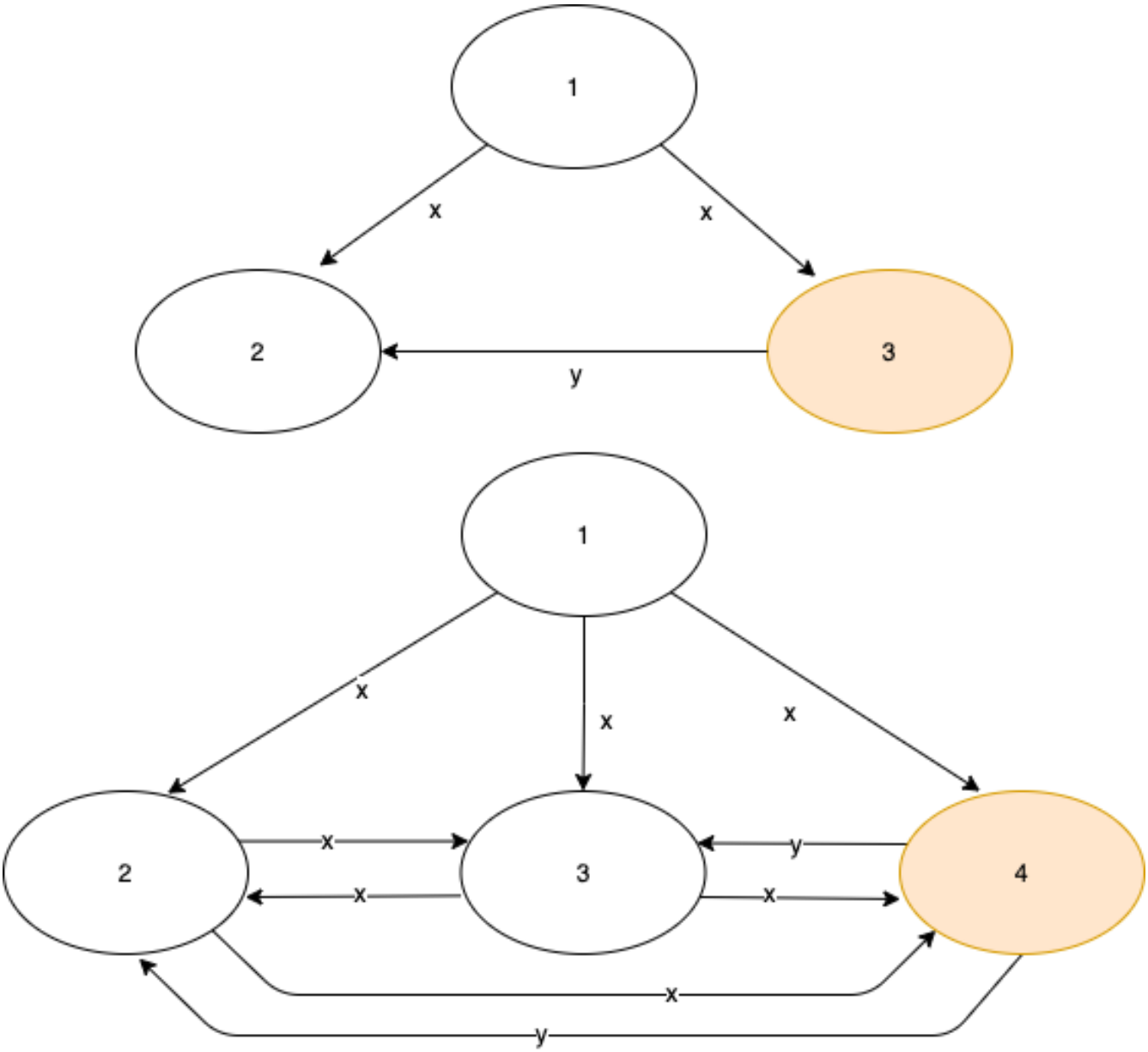
Виділяються типи повідомлень:

- *Вербальні* – неможливо надійно встановити первинне джерело та оригінальний зміст повідомлення – парафраз
- *Письмові* – Оригінальний зміст повідомлення та інформацію про його автора неможливо підробити. На практиці це означає використання криптографічної автентифікації (PKI, MAC)

В оригінальній постановці передбачається синхронна модель комунікації.

Було доведено, що якщо наказ є *вербальним*, рішення можливе, тільки якщо кількість лояльних вузлів більша ніж дві третіх від загальної кількості. Доведення полягає у демонстрації неможливості рішення з трьома генералами (рис. 2.7) та генералізації цього спостереження. Якщо ж наказ є *письмовим,* рішення існує для будь-якої кількості зрадників (щоправда, немає сенсу розглядати конфігурації, що

мають менше двох чесних генералів). Для обох варіантів – з автентифікацією і без – оригінальна робота пропонує алгоритми, які працюють, але є непрактичними через високу складність (кількість повідомлень для проведення одного раунду).



«Рисунок 2.7 – Верхній малюнок демонструє неможливість рішення з трьома генералами (генерал 3 – зрадник): генерал 2 отримує конфліктуючі відомості і не може дійти до висновку. Нижній малюнок (генерал 4 – зрадник) демонструє можливість рішення за 4 генералів.»

Прим. 2.3: Пізніше було продемонстровано [7], що за умов асинхронної комунікації, навіть за письмових повідомлень, консенсус можливий тільки якщо $n \geq 3f + 1$, де f – максимальна кількість вузлів, що можуть мати візантійську поведінку, а n – кількість вузлів загалом. Нехай r – кількість відповідей від інших, що має отримати коректний вузол в рамках одного обміну повідомленнями, щоб гарантовано зійтися на відповідях коректних вузлів. З одного боку, $r \geq 2f + 1$, адже $f + 1$ коректних вузлів є мінімальною більшістю, щоб переважити f візантійських вузлів. З іншого боку, вузол може не отримати відповідь від f вузлів зовсім (адже якщо вони є візантійськими, допускається їх мовчання), де усі f вузлів можуть насправді бути коректними, але такими, що надто довго відповідають (можливо за асинхронної моделі). Тому, $2f + 1 \leq r \leq n - f \Rightarrow n \geq 3f + 1$.

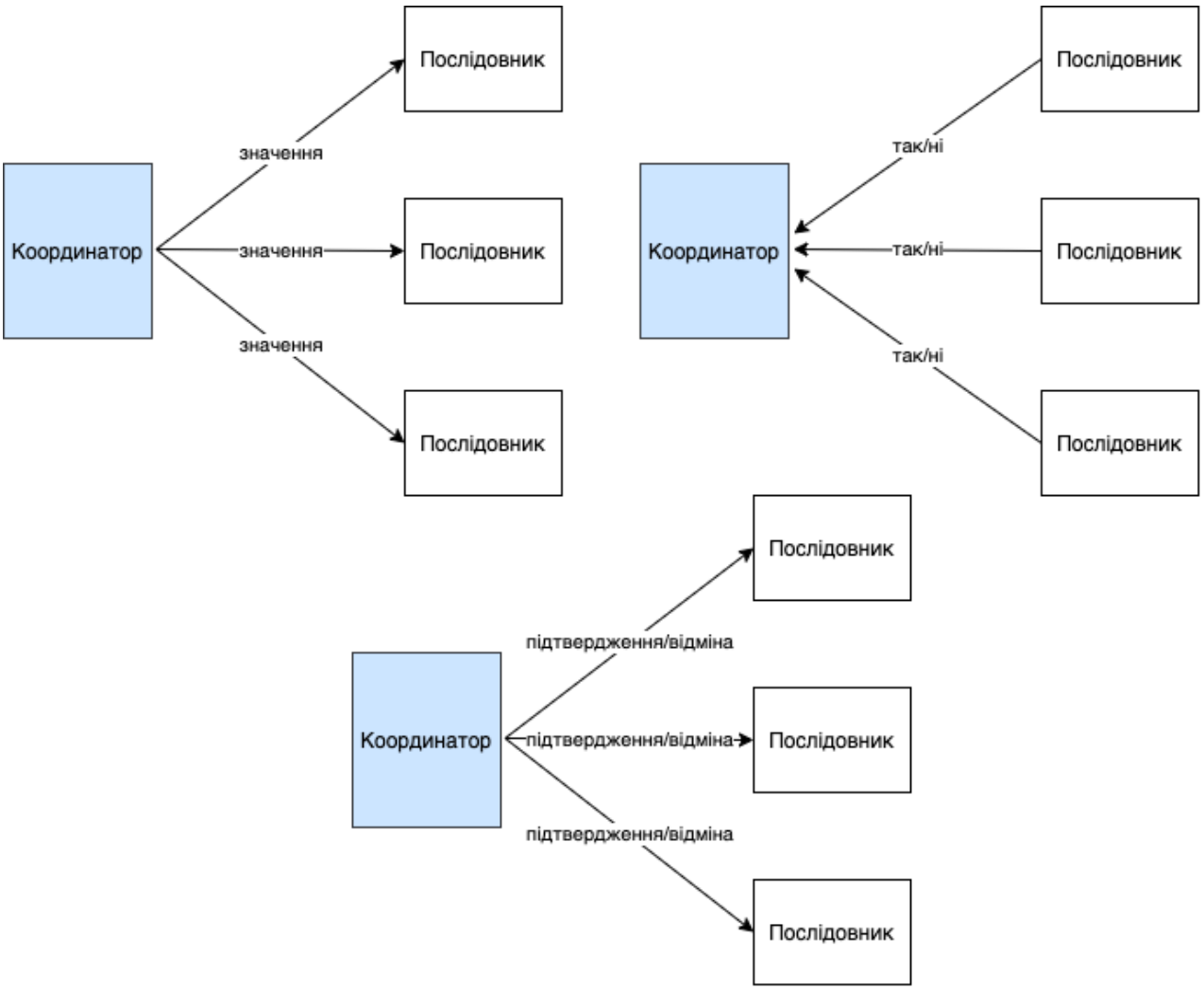
Здатність системи опиратися візантійській поведінці називається BFT (Byzantine Fault Tolerance). BFT-алгоритми є значною галуззю досліджень, було запропоновано ряд рішень, що суттєво покращують результат алгоритмів з оригінальної роботи, і таких, що частково підтримують асинхронну комунікацію. Застосування знаходиться в аерокосмічних системах, системах навігації, сенсорики, публічних блокчейнах, тощо – адже рішення моделі BGP означає здатність опиратися зловмисній або вкрай девіативній поведінці.

2.5.4 2PC

2PC [8], або «двохфазне підтвердження» – один з перших та найвідоміших протоколів консенсусу. Є прикладом однорідного консенсусу. Передбачається вузол-координатор. Протокол складається з двох кроків (рис. 2.8):

1. Координатор пропонує кожному учаснику значення, та збирає, очікуючи, їхні відповіді: «так» або «ні».

2. Якщо всі відповіли стверджувально – відправити усім підтверджувальне повідомлення, інакше – повідомити про відміну.



«Рисунок 2.8 – Кроки 2PC»

Протокол є простим та ефективним з точки зору кількості повідомлень в одному, що дорівнює кількості вузлів помноженій на три (хоча деякі варіації з підвищеною толерантністю до збоїв можуть досягати квадратичної складності в рамках протоколу відновлення), але на практиці доволі повільним через строго-синхронну модель: повний цикл має пройти перш ніж почнеться наступний, фази теж не можуть накладатися. Якщо проткол використовується, наприклад, для

розподіленої бази даних, а значення – транзакція, – то це може означати блокування ресурсів бази даних цією транзакцією (та блокування інших локальних процесів, яким ці ресурси треба) на час виконання циклу.

Алгоритм не є BFT і в базовому варіанті не передбачає толерантності до збоїв. За умови допущення дисфункції, навіть одного вузла, реалізація значно ускладнюється, особливо якщо проблема з координатором. За багато років 2PC в статусі популярного протоколу було придумано багато різних дрібних методик для специфічних сценаріїв збою: протоколи обрання нового координатора за умови виходу з ладу старого, протоколи прибирання неактивних вузлів з групи з можливістю їх повернення, застосування тайм-аутів, тощо. Додаткові заходи як правило вимагають WAL, та особливо обережного програмування прикладної логіки застосування запропонованого значення (наприклад, не робити невідворотні дії доки значення повністю не підтвердиться у фазі 2).

Прим. 2.4: Тим не менше, в загальному випадку досягнення консенсусу унеможливлюється за дисфункції. Якщо координатор виходить з ладу посеред фази 2, встигнувши відправити команду всього одному послідовнику (може бути і сам координатор), і той послідовник теж виходить з ладу, для системи в цілому тепер незрозуміло, чи координатор збирався підтвердити дію, і, якщо так – чи послідовник її виконав. Допустімо, що вузли здійснюють WAL, та є ідеальний протокол заміни координатора. Якби з ладу вийшов тільки координатор, щоб зрозуміти, чи збирався той підтвердити, новий координатор міг би опитати всі інші «живі» вузли як наче у фазі 1 і перейти до фази 2, повторюючи незавершену команду, але, в даному сценарії втрачений послідовник не зможе повторити відповідь з фази 1. Звісно, просто відміна прогресу тут не спрацює, адже зниклий послідовник міг отримати підтвердження і виконати дію, що як правило є невідворотною зміною локального стану. Тепер необхідно чекати доки або старий координатор, або зниклий послідовник відновить роботу, щоб їх опитати та надійно продовжити. Можливість для такого очікування є не завжди.

Навіть попри чутливість до збоїв, специфічні реалізації 2PC можуть давати достатню для варіанту використання відповідність критеріям коректності алгоритму консенсусу

2.5.5 3PC

3PC [9], або «трьохфазне підтвердження» – без сюрпризів, прямий нащадок 2PC, покликаний розв’язати проблеми останнього щодо толерантності до збоїв. Це досягається за рахунок додавання до 2PC третьої, підтверджувальної фази; окрім цього, міркування щодо 3PC ідентичні. Кроки 3PC:

1. Перший крок 2PC
2. Другий крок 2PC, але вузли тепер не виконують жодних невідворотних дій, лише дізнаються про те, яка дія має бути виконана у майбутньому: прийняття чи відмова
3. Координатор дає сигнал застосувати дію, що вузли дізналися на попередньому кроці

Протокол розв’язує проблему з прим. 2.4, адже тепер, не зважаючи на те, що дізнатися команду, яку віддав колишній координатор зниклому послідовнику все ще неможливо (доки хтось з них не повернеться і не буде опитаний), жодної невідворотної локальної дії зниклий послідовник не вчинить. Новий координатор в такій ситуації може просто обрати консервативну стратегію: повідомити вузли про відміну прогресу у проблемному раунді та продовжити з чистого листа.

Протокол вимагає від послідовників не виконувати жодних невідворотних дій аж до отримання підтвердження у третій фазі. Тим не менш, для покращення продуктивності прикладна реалізація (розподілений автомат) може виконувати максимум відворотних обчислень що передують невідворотним змінам, щоб у фінальний момент витратити якнайменше часу. Хоча, навіть попри техніки оптимізації, додаткова фаза значно вповільнює протокол.

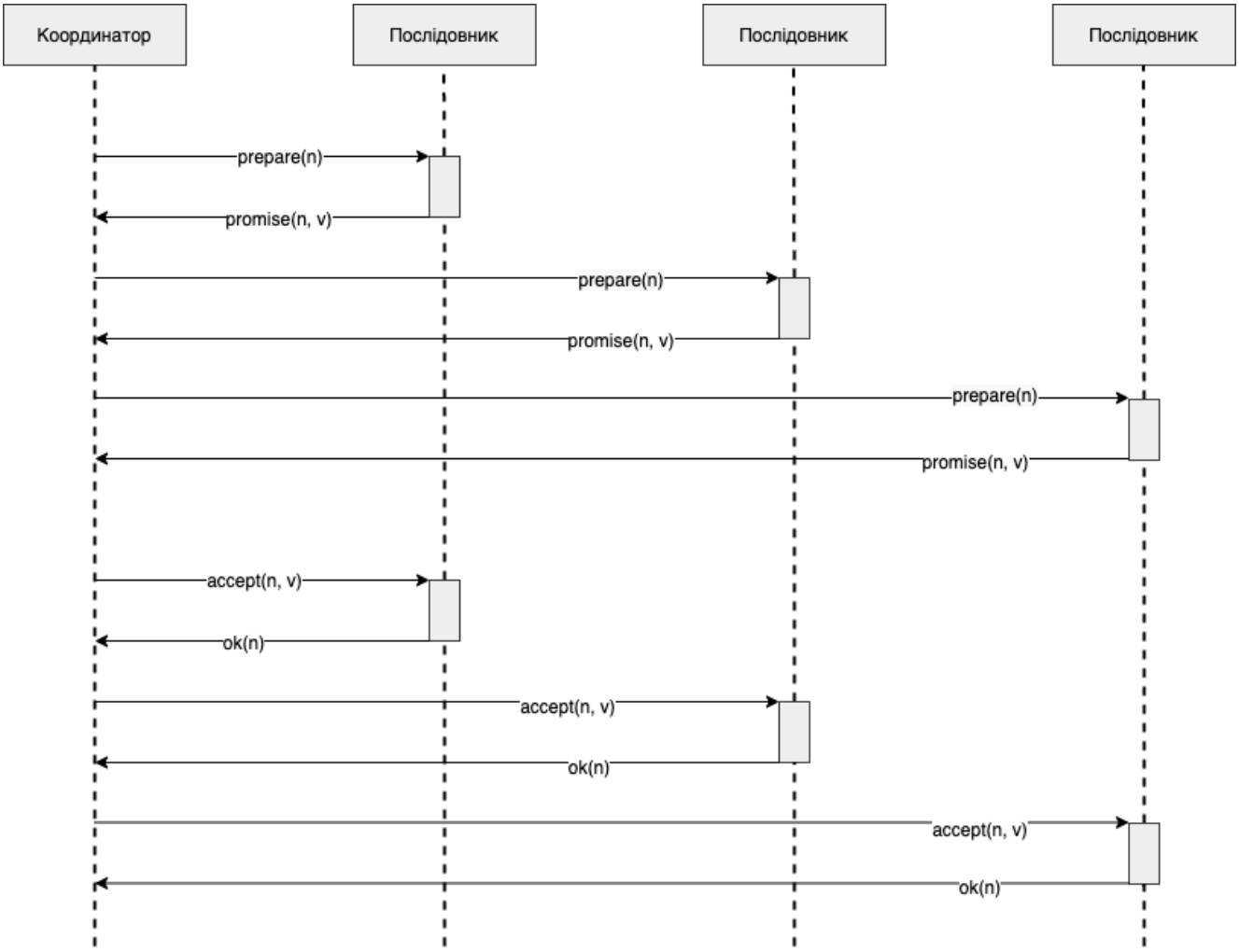
2.5.6 Paxos

Розглядаючи консенсус, неможливо ігнорувати Paxos [10][11] – ключовий класичний алгоритм, що першим запропонував практично-доцільне рішення в умовах збоїв та асинхронної комунікації. Алгоритм дозволяє збій-відновлення та збій-зупинку – не є BFT. Допускаються ненадійні засоби комунікації, але повідомлення не можуть бути спотворені. Так як консенсус неможливий за повністю асинхронної комунікації (результат FLP), Paxos гарантує безпечність, а рухливість в загальному є невизначеною: досягається за достатньо синхронізованої поведінки координаторів та адекватних затримок між відправленням та обробкою повідомлень – комунікація стає «достатньо синхронною»; на практиці часто це є задовільним. Вузол може виконувати будь-яку комбінацію ролей одночасно: координатор, послідовник та слухач; допускається декілька координаторів. Алгоритм дозволяє приймати рішення більшістю – не обов'язково усіма послідовниками. Пропозиція складається зі значення v та порядкового номеру n. Один повний цикл складається з (рис. 2.9):

1. Координатор обирає номер пропозиції *n* та відправляє команду *prepare* (приготуватися) з *n* всередині будь-якій більшості (наприклад, усім) послідовників.
2. Якщо послідовник отримує *prepare* з номером *n* вищим ніж у будь-якого *prepare*, на який він відповідав раніше, він відповідає координаторові обіцянкою *promise* не приймати більше пропозицій з номером нижчим за *n*, додаючи до відповіді пропозицію (*n* та *v*) з найвищим номером, що він вже встиг прийняти (якщо така є, інакше відповідь пуста). Якщо *prepare* має порядковий номер, нижчий за максимальний локально відомий, допускаються варіації: послідовник може просто проігнорувати команду і не відповісти, або відповісти, повідомляючи координаторові найвищий локальний порядковий номер (в деяких варіаціях координатор може спробувати повторити, тому йому треба актуалізувати свій порядковий номер), таким чином даючи йому знати,

що немає сенсу продовжувати дану пропозицію (координатор може знехтувати прогресом пропозиції у будь-який момент без загрози безпечності)

3. Якщо координатор отримує відповіді на *prepare* від більшості послідовників, він відправляє команду *accept* (прийняти) кожному з тих, що відповіли, додаючи до неї раніше обраний *n,* та *v*, що є будь-яким на розсуд координатора, якщо усі попередні відповіді на *prepare* були пусті, інакше – *v* з найвищим *n* серед усіх цих відповідей.
4. Коли послідовник отримує *accept* з номером *n*, він приймає його, якщо він тим часом не встиг відповісти на конкуруючий запит *prepare* (від іншого координатора) з вищим номером (пам’ятаємо: пообіцявши не приймати пропозиції з нижчим номером).
5. Отримавши підтвердження *accept* від більшості (відносно загальної кількості) послідовників, координатор вважає цикл завершеним.



«Рисунок 2.9 – Sequence діаграма нормального розвитку подій циклу Paxos»

Роль слухача полягає у слідкуванні за іншими з метою встановлення обраного значення (за принципом більшості). Порядкові номери *n* мають бути унікальними для кожного координатора, та у кожного координатора має бути механізм обрання номеру, вищого ніж у конкуруючої пропозиції іншого координатора: для цього вони мають користуватися множинами номерів, що не перетинаються, та такими, що для будь-якої пари множин, для будь-якого номеру з першої множини існує більший номер з другої множини. Опитування більшості послідовників у комбінації з тим, що в рамках одного циклу протоколу координатор або пропонує своє значення, якщо воно є першим відомим, або значення, вже запропоноване іншим координатором, є достатнім для гарантування унікальності рішення. Хоча, за умови високої конкуренції

між координаторами та необмежених затримок в комунікації (асинхронність) координатори можуть застрягати, намагаючись запропонувати одне й те ж значення, лише зі свого боку: гарантується безпечність, але рухливість є умовною. Якщо більшість accept була де факто прийнята послідовниками коли-небудь, рано чи пізно усі послідовники зійдуться на значенні, що містила відповідна пропозиція. Для зниження конкуренції пропонування прикладні реалізації часто обирають мати лише одного координатора – лідера, – використовуючи власне протокол (або будь-який інший протокол) для його обрання чи переобрання, якщо він стає недоступним. Також можна застосовувати евристики, що дозволяють координаторам поступатися одне одному. Збій-зупинка та збій-відновлення загалом не є проблемою за використання WAL.

Наданий опис оригінального алгоритму стосується лише одного циклу протоколу, і не розглядає континуум – пізніше для цього було розроблено декілька розширень, які колективно прийнято називати Multi-Paxos [12]. Існує широкий спектр варіацій Paxos, в тому числі BFT [13]. Його вплив спостерігається у багатьох сучасних фундаментальних алгоритмах, і досі він є одним з найпопулярніших алгоритмів консенсусу. Тим не менш, його прийнято вважати складним для розуміння та реалізації – звідси багатство інтерпретацій.

2.5.7 Raft

Raft [14] – новітній алгоритм, що фокусується на простоті та практичності. На відміну від попередньо розглянутих, він надає системне рішення для реалізації розподіленого автомату: специфікує поняття журналу та його реплікації, контрольні точки та компресію журналу, WAL, описує інтерфейс між консенсусом та локальним автоматом, та надає каркас для реконфігурації. Він також явно передбачає лідера, механізми його обрання та переобрання. Не є BFT, дозволяє збій-зупинку та збій-відновлення. Алгоритм передбачає синхронну модель, покладаючись на тайм-аути,

допускаючи невпорядкованість відправлення та обробки повідомлень. Вузол може бути лідером, кандидатом або послідовником. Кожен вузол має унікальний ідентифікатор. Все починається з обрання лідера (рис. 2.10), який в подальшому несе ексклюзивну відповідальність за реплікацію (або «пропагацію») журналу. Лідер обробляє запити (від зовнішніх клієнтів або зсередини групи), перетворює їх у записи журналу, та намагається пропагувати їх послідовникам. Час розділяється на строки (term); строк – порядковий номер періоду, що починається виборами, та, якщо лідера обрано, триває на час його правління, та закінчується початком нових виборів. Вузли пам’ятають номер останнього строку. Якщо послідовник достатньо довго не отримував звістки від лідера (тайм-аут), він інкрементує локальний номер останнього строку, переходить до стану кандидат, починає нові вибори з новим строком, та голосує за себе. Якщо кандидат отримує більшість голосів в рамках тайм-ауту виборів, він стає новим лідером. Тайм-аут виборів обирається кожним кандитатом по-своєму: до фіксованого базового значення додається випадкове значення у фіксованому інтервалі – це дозволяє рано чи пізно розв’язати потенційну гонку між декількома кандидатами, що намагаються стати лідерами.

«Рисунок 2.10 – Діаграма станів вузла Raft»

Запис журналу складається з:

- index – порядковий номер
- term – строк, що відповідає запису
- value – значення запису (аргумент для локального автомата)

Локальний стан вузла містить:

- id – ідентифікатор вузла (для простоти – його порядковий номер)
- currentTerm – локальний поточний term
- votedFor – якщо в поточному строці вузол проголосував за кандидата, містить його id, а інакше – нічого
- Журнал (список) записів
- commitIndex – index останнього підтвердженого запису журнала
- lastApplied – index останнього запису журнала, що був застосований до автомата

Додатково, стан лідера містить:

- nextIndex – масив, де порядковому номеру відповідає id вузла, а значенню – index локального запису, яку лідер планує відправити тому вузлові
- matchIndex – масив, де порядковому номеру відповідає id вузла, а значенню – index останнього запису, який лідеру вдалося пропагувати тому вузлові

Система повідомлень заснована на RPC. Існує два типи запиту:

1. RequestVote. Відправляється кандидатом іншому вузлу, щоб отримати голос за себе.

   Аргументи:

   - currentTerm кандидата
   - id кандидата
   - index останнього запису в журналі кандидата
   - term останнього запису в журналі кандидата

   Повертає:

- currentTerm запитаного вузла
- Чи голосує вузол за кандидата

Логіка:

- Якщо currentTerm кандидата менший за локальний currentTerm – відмова
- voterdFor непустий – відмова
- Якщо index останнього локального запису є вищим за той, що у запиті – відмова
- В інших випадках – голосувати за

2. AppendEntries. Відправляється лідером послідовникові для пропагації записів журналу. Також, за допомогою цього запиту лідер періодично дає про себе знати послідовникам – з цією метою він може передавати порожній список записів.

   Аргументи:

   - currentTerm лідера
   - id лідера
   - index останнього запису в журналі лідера (перед новими записами, що лідер цим запитом пропонує)
   - term цього запису
   - Список записів журналу, що пропагуються (може бути порожнім)
   - leaderCommit -- commitIndex лідера

   Повертає:

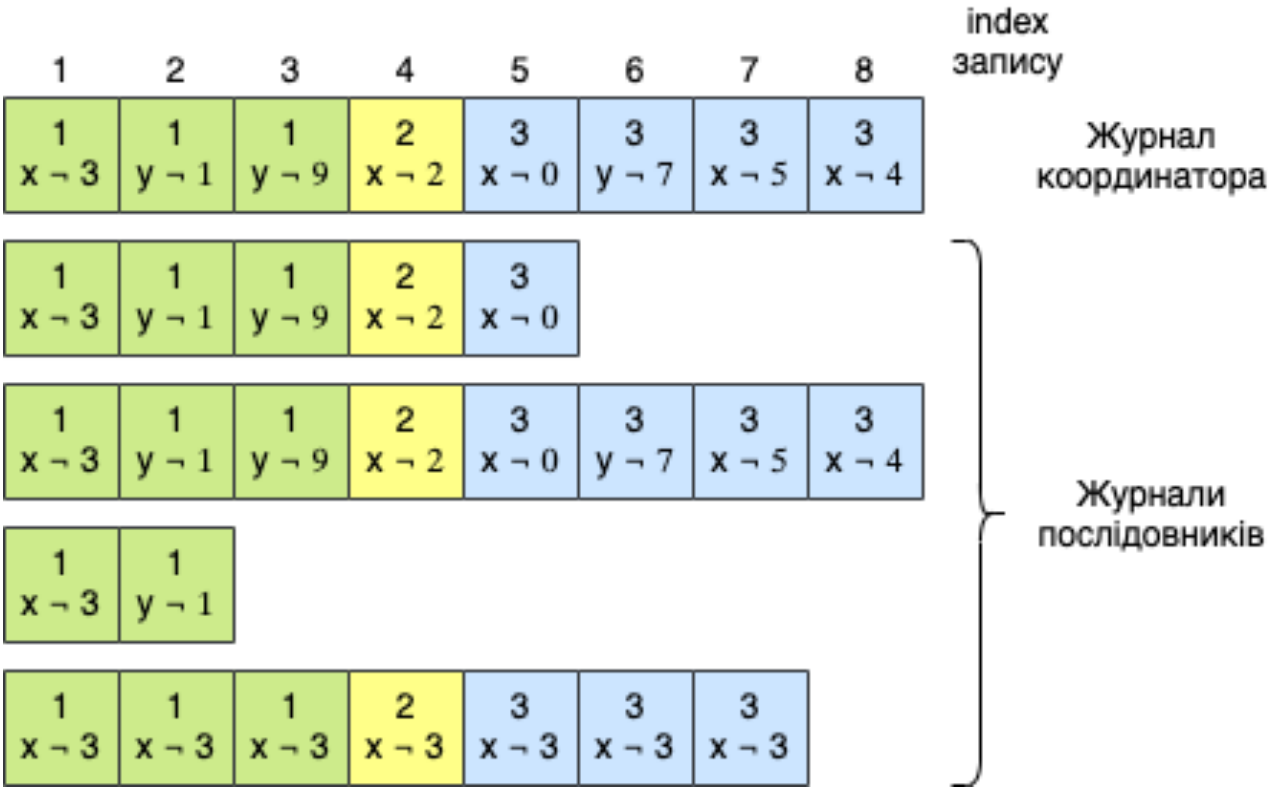
   - currentTerm запитаного вузла
   - Чи вдалося провести операцію

   Логіка:

   - currentTerm лідера менший за локальний currentTerm – відмова
   - У локальному журналі немає запису з заданим index та term, що дорівнює currentTerm – відмова

- Якийсь з локальних записів конфліктує (має той же index, але інший term чи value) з будь-яким з записом з запиту – видалити той локальний запис, та усі, що йому слідують
- Додати записи з запиту, що не містяться локально, в кінець журналу
- Якщо leaderCommit є більшим за локальний commitIndex, встановити в commitIndex мінімум серед leaderCommit та index останнього запису з запиту.

Обидва типи запиту повертають currentTerm запитаного вузла – це дозволяє вузлові, що запитує, усвідомити, що він відстає від системи, оновити власний currentTerm, та, якщо він є лідером – перейти у статус послідовника. Якщо commitIndex > lastApplied, вузол інкрементує lastApplied, та застосовує локальний запис, що відповідає lastApplied, до свого автомата. Якщо index останнього журналу лідера ≥ nextIndex для послідовника – лідер відправляє йому AppendEntries починаючи з цього nextIndex; якщо вдалося – оновити відповідно nextIndex та matchIndex для цього послідовника; інакше – декрементувати той nextIndex та спробувати заново. Лідер встановлює у власний commitIndex index журналу запису, якщо його term дорівнює currentTerm, та якщо matchIndex для більшості вузлів ≥ index цього запису (рис. 2.11). Потенційна гонка між кандидатами щодо отримання лідерства розв’язується використанням рандомізованих тайм-аутів.



«Рисунок 2.11 – Журнали вузлів Raft. Зеленим виділені застосовані до автомату записи, жовтим – записи, що відповідають commitIndex лідера»

Raft має наступні інваріанти:

- Лише один лідер може бути обраним для певного строку
- Записи лише додаються в кінець журналу лідера, в жодному разі не модифікуються чи видаляються
- Якщо два журнали містять запис з одним і тим же порядковим номером та строком, то вони є ідентичними від початку до даного запису включно
- Якщо запис журналу був підтверджений у певному строці, він гарантовано буде присутнім у журналах будь-яких лідерів у подальших строках
- Якщо вузол застосував запис журналу з певним порядковим номером n, строком k та значенням v до локального автомату, жоден інший вузол ніколи не застосує до локального автомату запис з тими ж n та k і значенням $v` ! = v$

Протокол досягає консенсусу за умови коректної роботи кворуму вузлів. Інакше

досяганняв прогресу неможливе – при цьому, унеможливлюється і досягання прогресу, який би загрожував узгодженості системи.

Дизайн, оптимізований з огляду на типову архітектуру однорангових систем, робить Raft привабливим для використання у реальних застосунках.

2.5.8 PBFT

Одним з найдовершеніших на сьогодні BFT-алгоритмом загального призначення є PBFT [15] (Practical BFT). Він гарантує безпечність за асинхронної комунікації, та рухливість за умов часткової синзронізації (як Paxos). Передбачає письмові повідомлення. Консенсус можливий, якщо $n \geq 3f + 1$, де n – кількість вузлів та f – максимальна допустима кількість візантійських вузлів – це відповідає теоретичному ліміту для алгоритму з подібними припущеннями (прим. 2.3). Алгоритм специфікується в контексті розподіленого кінцевого автомату: описує журнал повідомлень (в т.ч. WAL), обробку клієнтських запитів, та взаємодію між консенсусом та автоматом. Час сегментується на строки (view), у кожного строку є координатор (може мати візантійську поведінку). В рамках одного строку може пройти декілька циклів консенсусу, цикли можуть накладатися. Вузли можуть перейти в наступний строк, щоб змінити проблемного координатора (передбачається спосіб визначення ідентифікатора координатора в залежності від строку); для цього також передбачається протокол узгодження контрольних точок журналу. Поведінка коректних вузлів в рамках одного циклу консенсусу (рис. 2.12) за умови коректного координатора (аспект автентифікації опущено для простоти):

Клієнт C (може бути й одним з вузлів), відправляє запит request (аргументи для автомата) координаторові.

Координатор додає до запиту порядковий номер n та поточний номер строку v та переправляє усім послідовникам – pre-prepare.

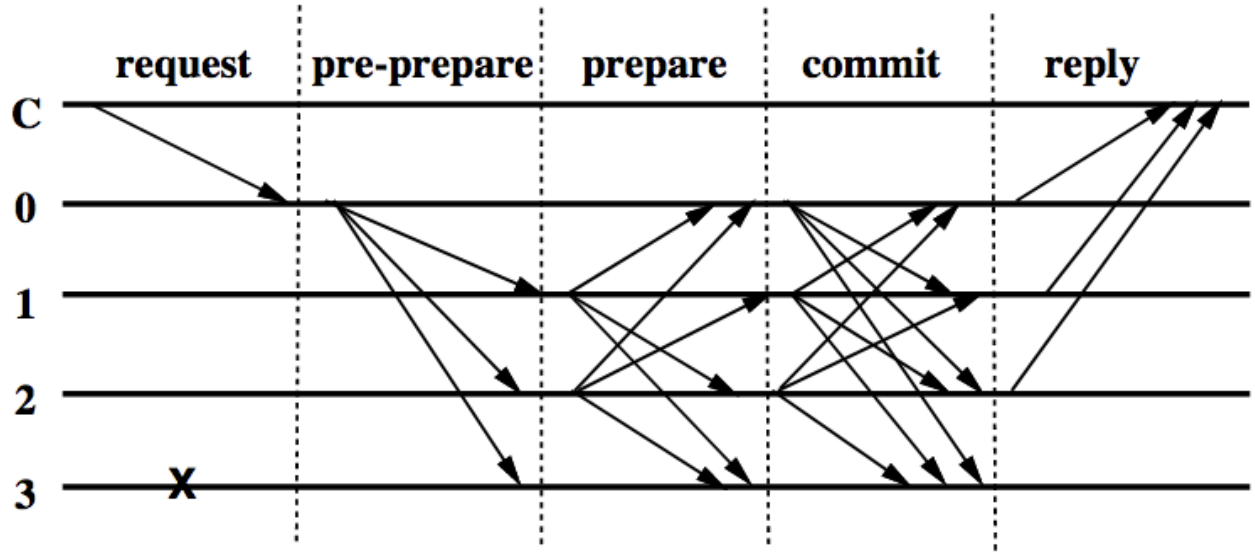
Якщо v відповідає локальному строку вузла, n задовольняє певний інтервал (передбачається спосіб визначення розумних рамок, що не дозволять зловмисному координатору вичерпати множину порядкових номерів), та якщо вузол не приймав pre-prepare для цих n та v раніше, він відправляє іншим prepare, що має той же вміст, що pre-prepare, але без request (можна його теж включати, але це не впливає на результат).

Отримавши $2f$ pre-prepare, вузол відправляє іншим commit з тим же вмістом.

Отримавши $2f + 1$ commit, вузол застосовує request до локального автомату, та повертає відповідь reply клієнтові C.

Отримавши відповідь від $f + 1$ вузлів, C може вважати request обробленим.

«Повернення відповіді» клієнту є умовністю: мається на увазі, що існує спосіб клієнтові дізнатися про факт застосування request до локального автомата.



«Рисунок 2.12 – Нормальний розвиток подій PBFT, вузол 3 – візантійський»

2.5.9 Конеснус в умовах публічності

Досі розглядалися методи, що передбачають закриті (приватні) системи: кількість вузлів під контролем, їх можна ідентифікувати. Публічність означає, що кількістю вузлів можна маніпулювати – це нівелює будь-які припущення, що базуються на кількостях коректних та некоректних вузлів.

Публічний блокчейн є прикладом розподіленого автомату з застосуванням консенсусу. Кінцеві користувачі відправляють вузлам транзакції (наприклад, намір переказати товаришу 1 одиницю цифрової валюти), вузли групують їх у блоки (аргументи для автомату), та пропонують одне одному; досягнувши консенсусу щодо наступного блоку, вони застосовують його до власних автоматів, змінюючи локальні бази даних відповідно до транзакцій блоку. Публічність означає, що будь-хто, у тому числі зловмисник, може приймати участь: система лише специфікує прикладний протокол (типи та зміст повідомлень, тощо), і будь-яка програма, яка вміє за ним спілкуватися (та може намагатися експлуатувати), може бути вузлом. Задача такого протоколу полягає у постановці правил, яким якщо слідуватимуть чесні вузли, та якщо їх є достатня (варіюється) більшість, система працюватиме коректно: з точки зору консенсусу (блоки обираються) та з точки зору автомату (не обираються зловмисні блоки). Необхідність додатково гарантувати коректність з точки зору автомату не дозволяє використовувати класичні алгоритми консенсусу у чистому вигляді, адже в кращому випадку ті обговорюють лише вплив консенсуса на автомат, і ніколи навпаки: немає можливості описати «погане» значення (наприклад, блок без транзакцій, що зловмисник пропагує, намагаючись затримати обробку реальних транзакцій користувачів – DoS) з точки зору вмісту – поняття, що визначається автоматом. Іншими словами, можна поводитися конструктивно на рівні консенсусу, поводячись деструктивно на прикладному рівні. В непублічній системі кількість вузлів можна контролювати та застосовувати криптографію для надійної їх ідентифікації, засновуючи на цьому припущення щодо толерантності системи до

девіативної поведінки – наприклад, можна представити проблему в категоріях BGP та застосувати BFT алгоритм. В публічній же системі, хоч криптографія і дозволяє асоціювати вузол з його поведінкою, зловмисник може створювати необмежену кількість вузлів, тому має бути спосіб відкинути такі вузли з розгляду, максимізуючи вагу чесних вузлів. Рішення, що для цього використовуються, базуються на наступних припущеннях:

1. Існує певний обмежений ресурс
2. Чесні актори не маніпулюють кількістю вузлів, що вони контролюють: одному актору відповідає один вузол
3. Кожен актор, для простоти моделі, має однакову частку ресурсу

За таких умов, нечесний актор може створювати скільки завгодно вузлів – це все ще не збільшує його частку ресурсу. Якщо система здатна оцінити співвідношення чесних та нечесних вузлів в термінах використання ресурсу, а не просто кількості вузлів, то класичні алгоритми знову стають застосовними. Досягнення цього вимагає синергії (або навіть злиття) між рівнем консенсусу та прикладним рівнем (розподіленим автоматом). Постають три глобальні проблеми:

1. Ідентифікувати ресурс
2. Верифікувати, що вузлові відповідає певна його частка
3. Забезпечити можливість акторів кооперативно сумувати власні частки з метою збільшення впливу на систему. Зокрема, максимізувати кооперацію серед чесних акторів.

Проблему 2 можливо розв’язати у технічно. Проблема 1 неодмінно змушує базуватися на припущеннях про реальний світ, адже лише в ньому поняття обмеженого ресурсу має сенс. Проблема 3 вимагає кооперації між чесними вузлами на прикладному рівні (алгоритмам консенсусу не відоме поняття ресурсу): вони мають концентрувати свій ресурс у спільному напрямку; так як ресурс знаходиться у полі фізичного світу, його використання вимагає дій у фізичному світі: дії у фізичому світі виконують люди.

Proof of Work (PoW) [16] – категорія алгоритмів, де ресурсом є обчислювальна потужність. Перші практичні рішення базувалися на PoW. Базисом для припущення щодо обмеженості ресурсу є обмеженість електроенергії та її прекурсорів, економічна обмеженість доступу до обчислювальної техніки, тощо. Існують математичні методи, що дозволяють поставити задачу W, обчислювальну складність розв'язання якої можна контролювати її параметрами (умовами), а складність її верифікації є невисокою (наприклад, Ethash). Блок в такій системі має містити у собі рішення подібної задачі, а її умови визначаються розподіленим кінцевим автоматом: детерміністично, базуючись суто на поточному стані, зокрема на вмісті останнього блоку (може бути декілька, якщо система передбачає ланцюг блоків у вигляді довільного направленого ациклічного графу, а не просто лінійного ланцюга) – це робить параметри і рішення задачі унікальними для ланцюга (підграфу) блоків. Цифрові підписи дозволяють надійно ідентифікувати автора блоку (що розв'язав задачу) та не дозволяють змінити блок. Здатність пропонувати блоки в такій системі і є власне доказом володіння часткою ресурсу. Помітимо, що ланцюги блоків можуть розходитися (на основі блоку А може бути побудовано блок Б і блок В, на основі блоку Б – блок Г, на основі В – блок Д, і так далі): це може відбутися внаслідок розділення вузлів мережевими неполадками, природнього паралелізму розподіленої системи, або ж внаслідок дій деструктивних вузлів. Довжина ланцюга блоків (від нульового до останнього) прямо-пропорційна обчислювальній складності його побудови, що прямо-пропорційна обсягам ресурсу, що необхідно витратити. Якщо чесні вузли будуть завжди слідувати найдовшому ланцюгові (пропонувати блоки на основі його останнього блоку), вони конвертують свої частки ресурсу у чисельну перевагу над деструктивними вузлами. Такий підхід працює навіть за мінімальної чисельної переваги чесних вузлів; жоден класичний BFT алгоритм не дає в подібних умовах такий сильний результат. Одним з недоліків є завжди присутня можливість появи нового довшого ланцюга, прийняття якого означає відміну прогресу виконання автомату до моменту, де локальний ланцюг перетинається з новим – відповідно, це

накладає додаткові вимоги на реалізацію автомату. Іншим недоліком є низька продуктивність та високе енергоспоживання через необхідність розв’язувати W.

Proof of Stake (PoS) [17] – сімейство алгоритмів, що в якості ресурсу використовує економічний статок вузла всередині системи: очікується, що він буде пропорційним статку актора у реальному світі. Метою цих алгоритмів є покращення продуктивності та енергоефективності у порівнянні з PoW. Методи, що застосовуються, різняться в деталях, але врешті-решт зводять проблему до BGP, далі застосовуючи (як правило, модифікований) BFT алгоритм (наприклад, PBFT).

2.6 Опції повторного використання

Реалізація розподіленого автомату та консенсусу з чистого листа є складною. Існує екосистема рішень, що можна повторно використати з метою полегшення розробки. Їх можна розділити на сервіси – системи, що під час виконання надають API для повторного використання прикладними клієнтськими системами, – та фреймворки – набори програмних компонентів на специфічній мові програмування, що надають каркас, який розширюється прикладною реалізацією.

Прикладом сервісу є розподілена база даних – прикладна система може користуватися нею як модулем, що надає консенсус. Бувають і сервіси, що спеціалізуються на консенсусі, такі як Zookeeper, Consul. Також, публічні блокчейни можуть використовуватися як консенсус-сервіси. Наприклад, Ethereum надає можливість в рамках транзакції створювати та виконувати програми довільної складності (будуть виконані в рамках розподіленого автомату) – смарт-контракти. Такій програмі виділяється база даних, що знаходиться всередині локальної бази даних кожного вузла, і клієнт може отримати до неї доступ через API вузла. Це дозволяє використовувати Ethereum та йому подібні як розподілену базу даних, та навіть повністю переносити на нього обчислення. Щоправда, обчислення в таких

системах мають ціну у внутрішній валюті, яка має ціну у реальному світі, а продуктивність є невисокою – зате підвищені гарантії безпеки, що є наслідком великої кількості чесних вузлів, які до того ж колективно підтримуватимуть існування клієнтських даних весь час життя системи. Існує ряд [18] прикладів застосування блокчейну для кіберфізичних систем, IoT.

Якщо ж метою є побудова системи, що не залежить від зовнішніх сервісів, то для багатьох алгоритмів (Paxos, Raft, PBFT, тощо) є фреймворки для різних мов програмування.

Висновки до розділу 2

Було досліджено теоретичні відомості та практичні підходи до рішення в контексті моделі проблеми. Типова клієнт-серверна архітектура має негативні наслідки через централізацію: вразливість до збоїв сервера, чутливість до якості мережі, необхідність підтримувати сервер, низька мобільність. Ситуація може бути покращена реплікацією сервера та БД, але це не усуває централізацію повністю – на відміну від однорангової архітектури. Простим способом перейти до неї від клієнт-серверної є ко-локація вузла розподіленої БД та прикладного ПЗ в рамках пристрою, але БД – система загального призначення, що як правило розрахована на значно вищі обсяги маніпуляції даними (та відповідно, використання обчислювальних ресурсів, що є обмеженими в умовах предметної області), ніж вимагає задача. Для розробки оптимальнішої з точки зору використання обчислювальних ресурсів однорангової системи з нуля, натуральною є архітектура на основі розподіленого кінцевого автомату. Вона передбачає застосування методів консенсусу. Вони різняться за припущеннями щодо цільової системи, тому важливо правильно їх визначити. Достатніми можуть бути і прості 2PC та 3PC, а якщо необхідні підвищені гарантії щодо толерантності до збоїв, можна звернутися до BFT алгоритмів, які також є значно

складнішими. Перевага протоколів з координатором, порівняно з протоколами без – у значно меншій кількості повідомлень, якими необхідно обмінятися для досягнення рішення; як наслідок, менші обсяги використання мережі, обчислювальних ресурсів, та збільшення швидкості. Відповідно, недолік – тимчасова централізація, що створює (мінливу, адже такі протоколи передбачають переобрання) вразливість. Також, координатора не можуть собі з легкістю дозволити системи, які не передбачають довіри між вузлами, адже ця роль є впливовою, що можна експлуатувати. Якщо мережа є достатньо якісною, можна припустити синхронну модель комунікації – це розкриває доступ до оптимальніших алгоритмів. Не обов'язково розробляти консенсус з нуля – можна використати сервіси, що його надають, як наприклад розподілені бази даних. Зокрема, публічний блокчейн можна використовувати як сервіс консенсусу, що має вийняткову тривалість та надійність: інформація, зареєстрована на ньому, потенційно може існувати стільки ж, скільки людська цивілізація, і підробити її ретроактивно буде практично неможливо – але за рахунок низької продуктивності та фінансових витрат. Якщо ж сторонні сервіси не вписуються в архітектуру, для полегшення розробки для багатьох алгоритмів існують фреймворки на різних мовах програмування.

3. ОПИС РІШЕННЯ

Мовою програмування було обрано Go. Go [19] є статично-типізованою мовою з підтримкою об’єктно-орієнтованої та функціональної парадигми, передбачає автоматичне управління пам’яттю, компілюється в ефективний машинний код цільової платформи, надає безпосередній доступ до представлення даних, має потужні вбудовані засоби паралельного програмування, добре стандартизовану, підтримувану та функціональну інфраструктуру щодо побудови проекту, тестування, управління залежностями, тощо. Підтримує крос-компіляцію, зокрема під ARM. Екосистема мови містить усі необхідні для розробки даного рішення бібліотеки. Мова є простою та високорівневою – при цьому дозволяє створювати ефективний код.

Реалізована система є одноранговою: реплікований кінцевий автомат з використанням консенсусу. Це робить її самодостатньою, мобільною, толерантною до збоїв за рахунок множинності та взаємозамінності вузлів, та дозволяє користуватися перевагами локального розміщення, покращуючи якість мережевої взаємодії.

В якості алгоритму консенсусу обрано Raft. Припущення Raft щодо прикладної системи є адекватними постановці задачі:

- Між вузлами передбачається довіра, адже система є закритою, тому лідер-орієнтованість Raft не становить загрози
- З точки зору мережі, передбачається, що вузли розміщуватимуться поруч, тому можна очікувати достатній для Raft рівень синхронності та надійності комунікації

Переваги ж Raft добре поєднуються з поставленою задачею:

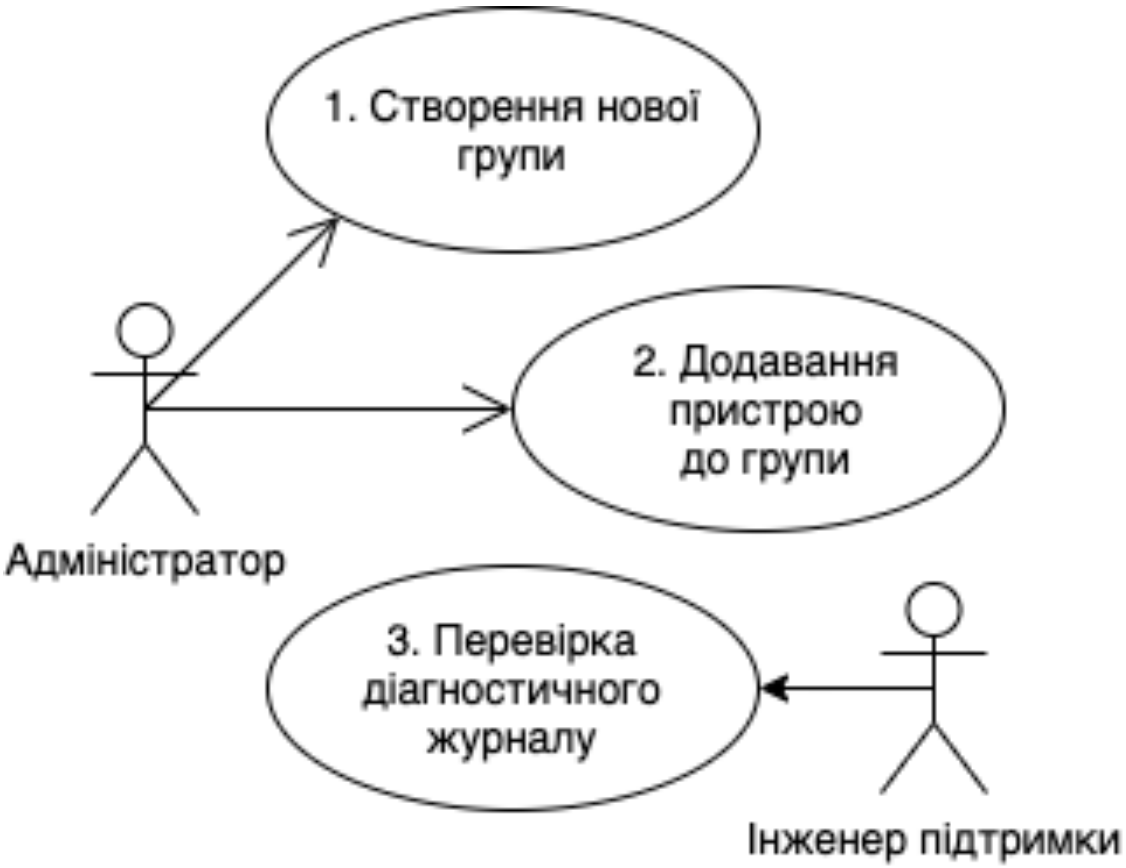
- Достатньо кворуму вузлів для коректної роботи
- Генерує малу кількість повідомлень, має добру продуктивність
- Надає системне рішення для розподіленого автомату

Використовується фреймворк на мові Go, що надає скелетну імплементацію

Raft, дозволяючи розширити поліморфні моменти, такі як реалізація автомату чи спосіб зберігання журналу. Журнал локально зберігається у базі даних BoltDB (БД ключ-значення бібліотечного типу, що базується на локальній файловій системі) він містить лише останній запис про актуальний стан системи $SystemState$, їх історія не зберігається: логіка компресії журналу в момент досягнення контрольної точки полягає у простому відкиданні всіх записів, крім найновішого. Була розроблена реалізація автомату для даного фреймворку – модуль «FSM», – та код, що керує модулем «RAFT» – центральним об'єктом даного фреймворку, що являє собою локальний вузол протоколу Raft.

Вузол має публічний інтерфейс – модуль «API» – на основі JSON-RPC поверх HTTP, реалізований засобами стандартної бібліотеки Go. Дозволяє адміністратору (кінцевому користувачу) керувати топологією системи. Також API використовується вузлами для передачі локальних показників середовища лідерові. Методи API:

- StartCluster() – (відповідає прецеденту 1 на рис. 3.1) приймаючий вузол має створити нову групу з собою у якості лідера. Якщо приймаючий вузол вже є членом групи – помилка.
- AddNode(id, address) – (відповідає прецеденту 2 на рис. 3.1) приймаючий вузол має додати до своєї групи вузол з ідентифікатором id та мережевою адресою address. Обробляється лідером, якщо приймаючий вузол не є лідером, він переправляє цей запит лідерові. Якщо лідер наразі не відомий – повертає помилку.
- AcceptNodeState(nodeState) – вузли використовують цей метод для передачі власних показників середовища $NodeState_i$ лідерові. Метод повертає помилку, якщо приймаючий вузол не є лідером.



«Рисунок 3.1 – Діаграма прецедентів»

Модуль «CALC» відповідає за обчислення $SystemState$, модуль «ENV» – за взаємодію з фізичним середовищем: знімає показники I та встановлює O. Модулі API, CALC, ENV, RAFT працюють в окремих потоках. Вузли можна безпечно зупиняти та перезавантажувати – вони відновлять свій стан з локально збережених даних та повернуться до комунікації з іншими, актуалізувавши свій стан як наслідок.

Загальна схема роботи системи (рис. 3.1):

1. Адміністратор керує структурою кластера, використовуючи методи API StartCluster, AddNode.
2. Лідер пропагує оновлення $SystemState$ через модуль RAFT. Модуль FSM кожного вузла отримує новий $SystemState$, використовує його щоб встановити $O_i = O_{next_i}$ модулеві ENV. Модуль ENV вузла N_i відправляє лідеру

$NodeState_i$, де I_i генерується випадково, за допомогою API. Лідер передає його до свого модуля CALC, який використовує дані $NodeState_i$ для обрахування

$$O_{opt_i} = \frac{RI_i}{\sum_{k=0}^{Q} I_k} \quad (3.1)$$

та потім

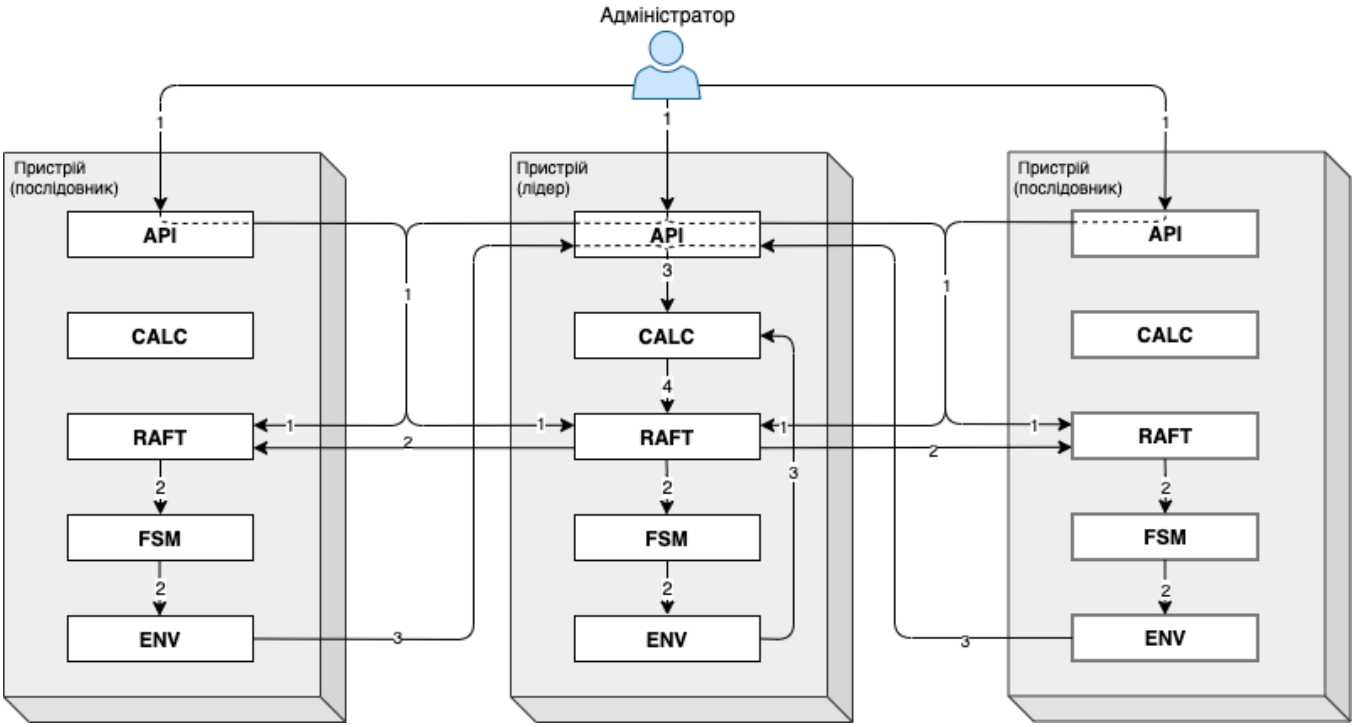
$$O_{next_i} = O_{opt_i} if\ O_{opt_i} \leq O_i\ else \frac{\left(R - \sum_{k=0}^{Q} MAX\left(O_i, O_{next_i}\right)\right)I_i}{\sum_{k=0}^{Q} I_k} \quad (3.2)$$

таким чином оновлюючи $SystemState$. Обрахунки формул 3.1 та 3.2 відбуваються в окремому потоці.

3. Періодично, модуль CALC лідера ініціює крок 2

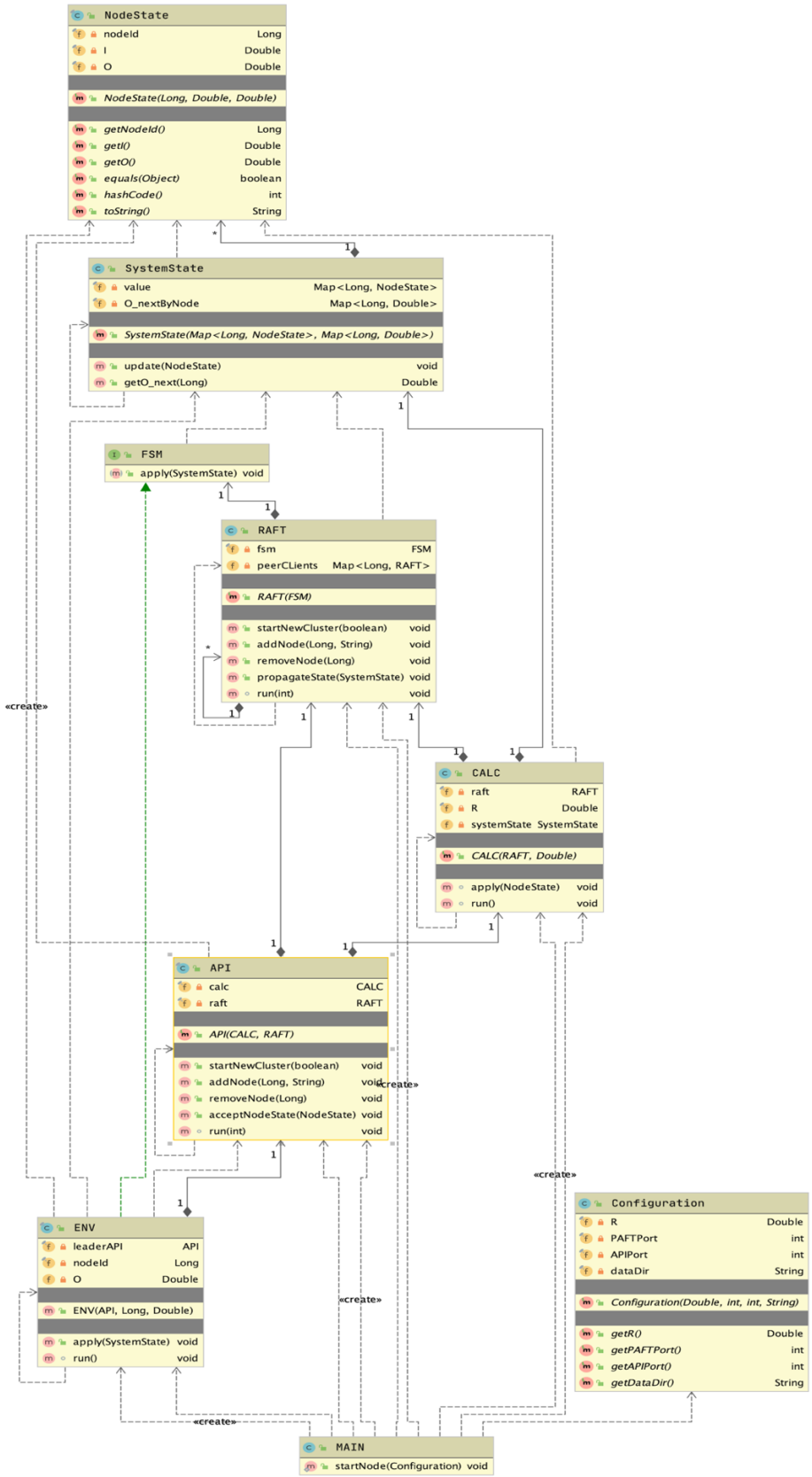
Відповідна діаграма класів та діаграми послідовностей зображені на рис. 3.3, 3.4, 3.5, 3.6, 3.7.

Асинхронність обрахунків та періодичність пропагування їх результатів модулем CALC дозволяє ефективно обробляти високу частоту зміни I_i, пропускаючи через консенсус мінімум інформації (консенсус є відносно повільним).

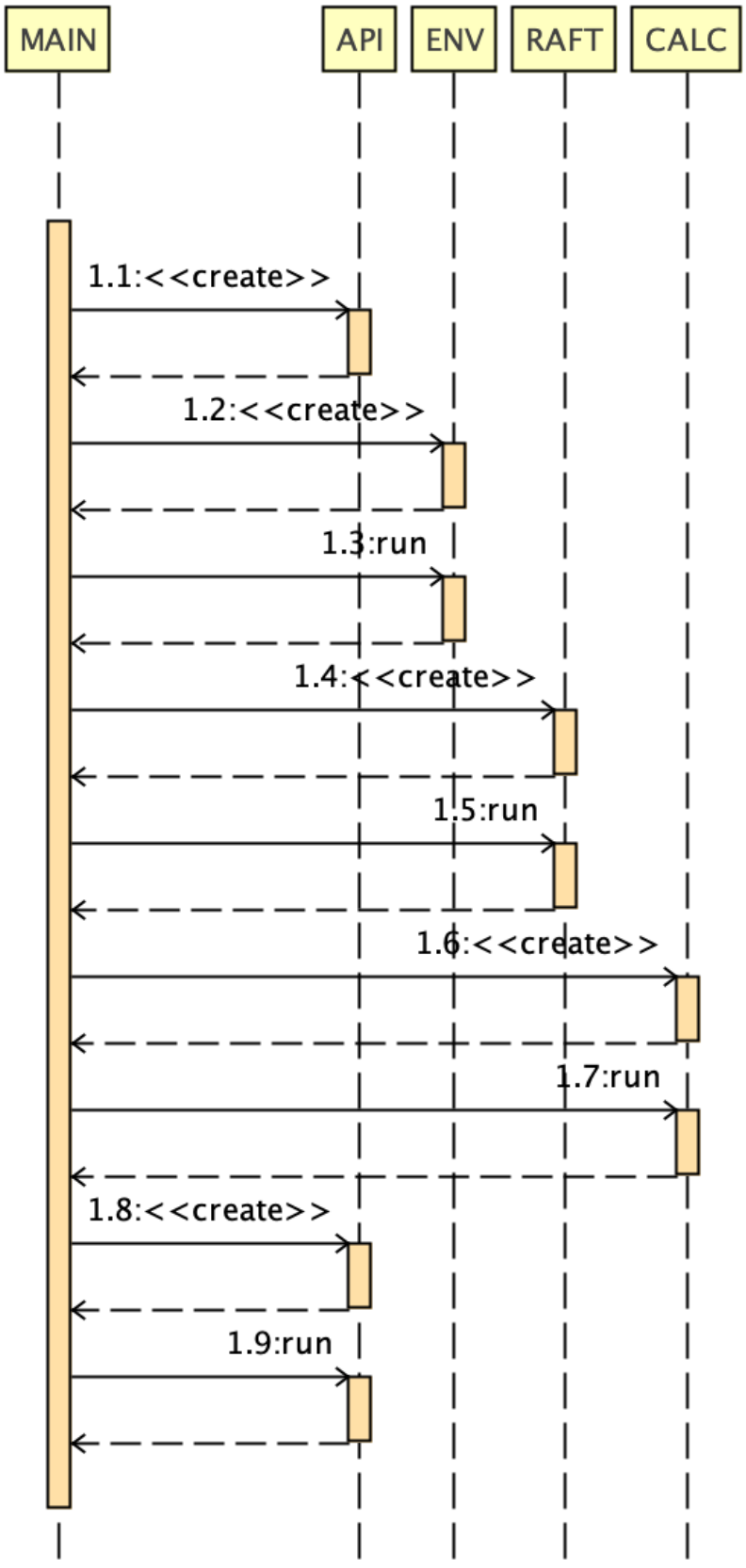


«Рисунок 3.2 – Загальна схема роботи системи»

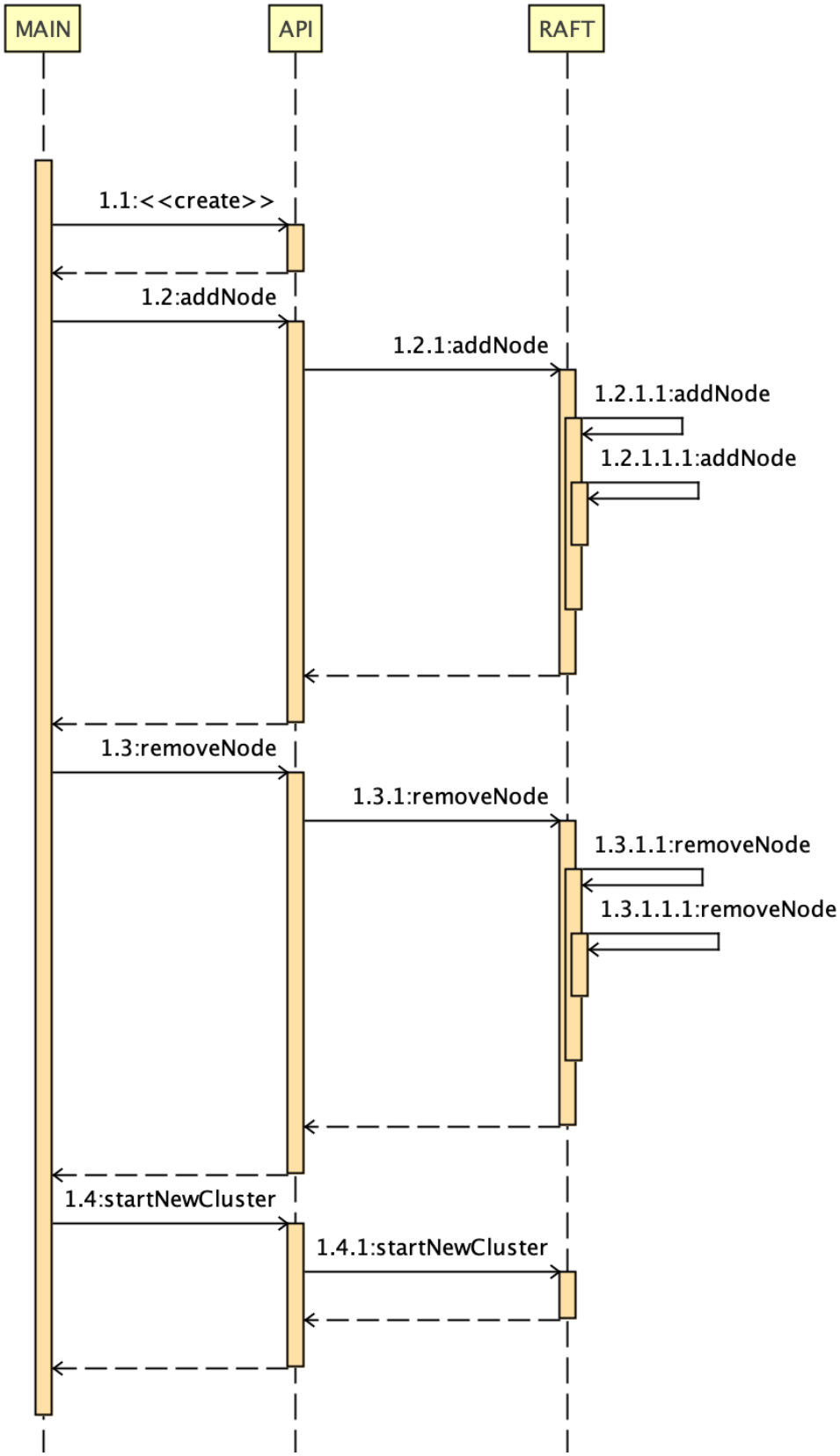
Також, В ході розробки використовувався Python 3 – для прототипування алгоритмів та автоматизації. Сімейство IDE від JetBrains використовувалося для задач, пов’язаних з написанням коду. В якості системи контролю версій використовувався Git/GitHub.



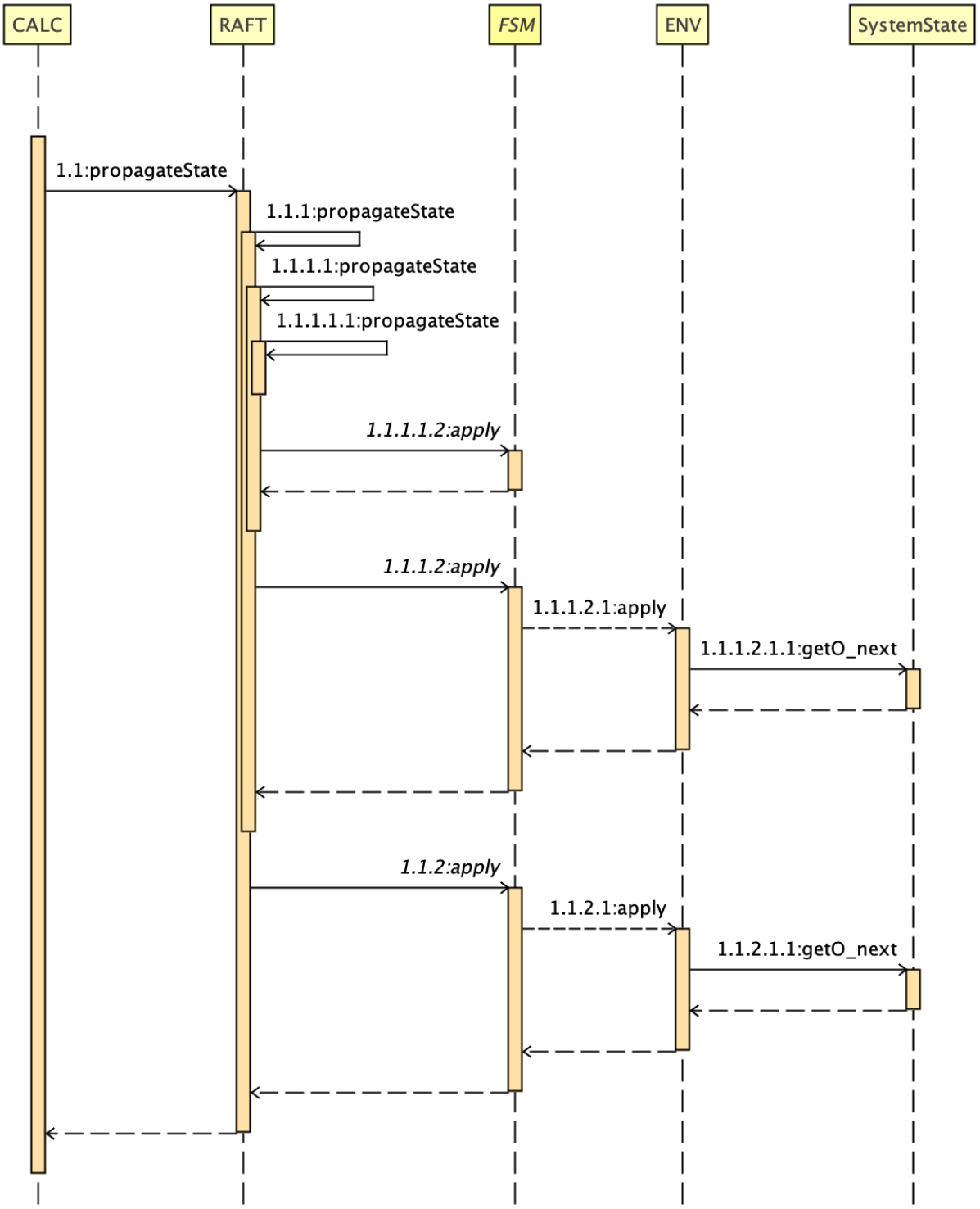
«Рисунок 3.3 – Діаграма класів»



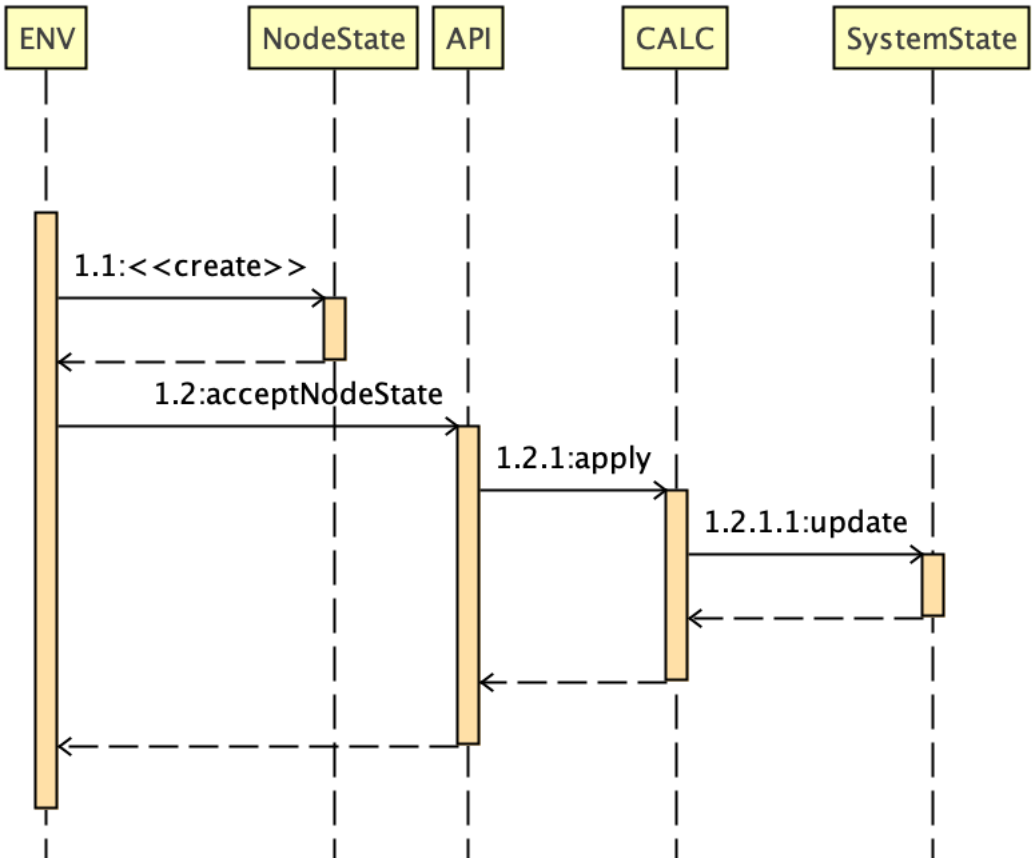
«Рисунок 3.4 – Sequence діаграма потоку запуску вузла»



«Рисунок 3.5 – Sequence діаграма методів API для адміністратора»



«Рисунок 3.6 – Sequence діаграма потоку модуля CALC»



«Рисунок 3.7 – Sequence діаграма потоку модуля ENV»

В якості апаратної платформи обрано Raspberry Pi Zero W. Даний пристрій має WI-FI модуль, невеликі розміри, та вдосталь обчислювальних ресурсів. Короткі характеристики, від виробника:

- 1GHz single-core ARMv6 CPU (BCM2835)
- VideoCore IV GPU, 512MB RAM
- Mini HDMI and USB on-the-go ports
- Micro USB power
- HAT-compatible 40-pin header
- Composite video and reset headers
- CSI camera connector
- 802.11n wireless LAN
- Bluetooth 4.0

На пристрій встановлюється варіація Debian Linux, спеціалізована для сімейства пристроїв даного виробника – Raspbian – з попередньо встановленим та ПЗ даного рішення: воно є простим консольним додатком, процес якого запускається автоматично разом з ОС пристрою за допомогою supervisord – популярного інструменту для управління процесами на UNIX-подібних системах.

Унікальний ідентифікатор вузла генерується випадково (UUID) за першого запуску. Параметри конфігурації ПЗ вузла включають (клас Configuration на рис. 3.1):

- Налаштування TCP для Raft та API
- Налаштування протоколу Raft (тайм-аути виборів, тощо)
- Шлях до директорії файлової системи, що міститиме стан Raft (журнал, тощо), діагностичний журнал, та згенерований UUID
- Параметр R

Для простоти та зручності, усі ці параметри мають значення, встановлені за замовчуванням (порт API – 8888).

Для початку експлуатації, адміністратор має зробити вузли видимими одне одному в мережі: для цього він має приєднатися до локальної WI-FI точки доступу пристрою, почати з ним сесію терміналу (SSH), та підімкнути його до бажаної мережі, користуючись стандартними утилітами Rapsbian. Після цього, за допомогою будь-якого HTTP клієнта використати відповідні методи API для об’єднання їх у групу Raft. Далі система працює автономно. Такий користувацький інтерфейс є достатньо сирим, хоча й достатнім. До того ж, API надає достатньо можливостей для створення зовнішніх автоматизованих сценаріїв управління кластером (скриптів) або ж зручного графічного інтерфейсу.

ПЗ вузла є незалежним від розміщення, і тому його зручно тестувати. Тестування проводилося у автоматичному режимі. Відповідно до вимог, кожен вузол пише діагностичну інформацію у локальний файл. Тести (див. додаток А, файл «root_test.go»), написані на мові Go, запускають вузли в окремих потоках, формують їх у групу, як це робив би адміністратор, дають системі попрацювати, симулючи

неполадки, та паралельно перевіряючи їхні діагностичні журнали згідно до умов постановки задачі (прим 1.1). Також, ця діагностична інформація має призначення в рамках production-використання: інженер підтримки може користуватися нею в рамках усунення неполадок (відповідає прецеденту 3 на рис. 3.1).

Висновки до розділу 3

Було обрано архітектуру, оптимальну для проблеми, та розроблено рішення за нею. Організовано фінальний пакет продукту – пристрій з інстальованим та сконфігурованим ПЗ. Передбачено засоби ініціалізації та адміністрації системи на боці користувача. Передбачено метод тестування, та розроблено за ним відповідні автоматичні тести. Описано схему роботи та ключові алгоритми.

ВИСНОВКИ

В бакалаврській роботі було розглянуто розробку апаратно-програмного комплексу управління розподіленою мережею сенсорів. В якості відправної точки, створено модель проблеми, що імітує реальні задачі та середовище розподіленої системи, що використовує сенсори для прийняття локальних рішень, та закладено умову, виконання якої вимагає кооперацію між її вузлами. Було проаналізовано недоліки типового підходу до рішення подібної проблеми, варіанти його покращення, і продемонстровано еволюцію ідей, що приводить до моделі однорангової системи на базі розподіленого кінцевого автомату, та з особливим акцентом досліджено її центральний елемент – алгоритми консенсусу. Теоретичні відомості та практичні міркування було синтезовано в оптимальному рішенні з огляду на умови постановки задачі, та як наслідок отримано робочий прототип та архітектурний шаблон, що може використовуватися для побудови прикладних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. M. Bertocco, F. Ferraris, C. Offelli and M. Parvis, "A client-server architecture for distributed measurement systems," in IEEE Transactions on Instrumentation and Measurement, vol. 47, no. 5, pp. 1143-1148, Oct. 1998, doi: 10.1109/19.746572.

2. Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (Dec. 1990), 299–319.

3. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374–382.

4. Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558–565.

5. M. Pease, R. Shostak, and L. Lamport. 1980. Reaching Agreement in the Presence of Faults. J. ACM 27, 2 (April 1980), 228–234.

6. Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. Concurrency: the Works of Leslie Lamport. Association for Computing Machinery, New York, NY, USA, 203–226

7. Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. J. ACM 32, 4 (Oct. 1985), 824–840.

8. Lechtenbörger J. (2009) Two-Phase Commit Protocol. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA

9. Al-Houmaily Y.J., Samaras G. (2009) Three-Phase Commit. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA

10. Leslie Lamport. 1998. The part-time parliament. ACM Trans. Comput. Syst. 16, 2 (May 1998), 133–169.

11. Leslie Lamport. 2001. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)

12. Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. ACM Comput. Surv. 47, 3, Article 42 (April 2015), 36 pages.

13. Lamport L. (2011) Byzantizing Paxos by Refinement. In: Peleg D. (eds) Distributed Computing. DISC 2011. Lecture Notes in Computer Science, vol 6950. Springer, Berlin, Heidelberg

14. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014.

15. Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery." ACM Transactions on Computer Systems (TOCS) 20.4 (2002): 398-461.

16. Nakamoto, Satoshi. Bitcoin: A peer-to-peer electronic cash system. Manubot, 2019.

17. C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen and E. Dutkiewicz, "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities," in IEEE Access, vol. 7, pp. 85727-85745, 2019

18. M. S. Ali, M. Vecchio, M. Pincheira, K. Dolui, F. Antonelli and M. H. Rehmani, "Applications of Blockchains in the Internet of Things: A Comprehensive Survey," in IEEE Communications Surveys & Tutorials, vol. 21, no. 2, pp. 1676-1717, Secondquarter 2019, doi: 10.1109/COMST.2018.2886932.

19. The Go Programming Language Documentation [Електронний ресурс] – Режим доступу: https://golang.org/doc/

ДОДАТОК А

go.mod

```
module diplom_go

go 1.13

require (
   github.com/hashicorp/raft v1.1.2
   github.com/hashicorp/raft-boltdb v0.0.0-20191021154308-4207f1bf0617
)
```

api.go

```
package diplom_go

import (
   "diplom_go/util"
   "diplom_go/util/jsonrpc_http"
   "errors"
   "net"
   "time"

   "github.com/hashicorp/raft"
)

type APIs struct {
   API
}

func (self *APIs) FrontendInit(tcp_addr jsonrpc_http.TCPAddrFn) *APIs {
   jsonrpc_http.PopulateClients(self, tcp_addr)
   return self
}

func (self *APIs) BackendInit(raft *RAFT, calc *CALC) *APIs {
   self.raft = raft
   self.calc = calc
   self.leader_apis = new(APIs).FrontendInit(func() *net.TCPAddr {
      leader_addr := self.raft.Leader()
      if len(leader_addr) == 0 {
         return nil
      }
      addr, err := net.ResolveTCPAddr("tcp", string(self.raft.Leader()))
      util.Check(err)
      addr.Port = RPCPort(addr.Port)
      return addr
   })
   return self
}

type API struct {
   client      jsonrpc_http.Client
   raft        *RAFT
   calc        *CALC
   leader_apis *APIs
}
```

```
func (self *API) AcceptNodeState(in *NodeState, out RPCVoid) error {
   if self.client != nil {
      return self.client(in, out)
   }
   return call_leader(
      func() error {
         if self.raft.State() != raft.Leader {
            return raft.ErrNotLeader
         }
         self.calc.AcceptNodeState(in)
         return nil
      },
      func() error {
         return self.leader_apis.AcceptNodeState(in, out)
      },
   )
}

type AddNodeReq = struct {
   NodeID   NodeID
   NodeAddr raft.ServerAddress
   Timeout  time.Duration
}

func (self *API) AddNode(in *AddNodeReq, out RPCVoid) error {
   if self.client != nil {
      return self.client(in, out)
   }
   return call_leader(
      func() error {
         return self.raft.AddVoter(in.NodeID, in.NodeAddr, 0, in.Timeout).Error()
      },
      func() error {
         return self.leader_apis.AddNode(in, out)
      },
   )
}

func (self *API) StartNewCluster(in, out RPCVoid) error {
   if self.client != nil {
      return self.client(in, out)
   }
   return self.raft.BootstrapCluster(raft.Configuration{[]raft.Server{{
      ID:      self.raft.cfg.NodeID,
      Address: self.raft.transport.LocalAddr(),
   }}}).Error()
}

var ErrNoLeader = errors.New("No known leader")

func call_leader(apply_local, apply_remote func() error) error {
   if err := apply_local(); err != nil {
      if err != raft.ErrNotLeader {
         return err
      }
      if err := apply_remote(); err != nil {
         if err == jsonrpc_http.ErrNoEndpointAddress {
            return ErrNoLeader
         }
         return err
      }
   }
   return nil
}
```

```
type RPCVoid = *struct{}
```

calc.go

```
package diplom_go

import (
   "diplom_go/util"
   "fmt"
   "math/big"
   "runtime"
   "sync"
   "sync/atomic"
   "time"
)

type CALC struct {
   fsm_state      *SystemState
   fsm_state_mu   sync.Mutex
   node_states    sync.Map
   node_states_mu sync.RWMutex
   subprocess_cnt int32
   status         uint32
}
type CALCConfig = struct {
   CyclePeriod time.Duration
}

func (self *CALC) Init(cfg CALCConfig, state *SystemState, apply func([]byte)) *CALC {
   var apply_val []byte
   var apply_mu sync.Mutex
   apply_signal := sync.NewCond(&apply_mu)
   atomic.AddInt32(&self.subprocess_cnt, 1)
   go func() {
      defer atomic.AddInt32(&self.subprocess_cnt, -1)
      for {
         apply_mu.Lock()
         if apply_val == nil {
            apply_signal.Wait()
            if apply_val == nil {
               return
            }
         }
         apply_val_tmp := apply_val
         apply_val = nil
         apply_mu.Unlock()
         apply(apply_val_tmp)
      }
   }()
   state = state.DeepCopy()
   atomic.AddInt32(&self.subprocess_cnt, 1)
   go func() {
      defer atomic.AddInt32(&self.subprocess_cnt, -1)
      defer func() {
         for ; atomic.LoadInt32(&self.subprocess_cnt) != 1; runtime.Gosched() {
            apply_signal.Signal()
         }
      }()
      for atomic.LoadUint32(&self.status) != 1 {
         time.Sleep(cfg.CyclePeriod)
```

```
            self.fsm_state_mu.Lock()
            if self.fsm_state != nil && state.Version <= self.fsm_state.Version {
               state = self.fsm_state
            }
            self.fsm_state_mu.Unlock()
            have_updates := false
            self.node_states_mu.Lock()
            self.node_states.Range(func(key, value interface{}) bool {
               state.NodeStates[key.(NodeID)] = value.(IO)
               self.node_states.Delete(key)
               have_updates = true
               return true
            })
            self.node_states_mu.Unlock()
            if !have_updates {
               continue
            }
            R_left := new(big.Int).Set(R)
            I_sum := new(big.Int)
            for node_id, io := range state.NodeStates {
               I_sum = new(big.Int).Add(I_sum, new(big.Int).SetUint64(io.I))
               R_left = new(big.Int).Sub(R_left, new(big.Int).SetUint64(util.MaxU64(io.O,
state.O_next[node_id])))
            }
            ok := true
            for node_id, io := range state.NodeStates {
               if io.O != state.O_next[node_id] {
                  continue
               }
               I := new(big.Int).SetUint64(io.I)
               O_next := new(big.Int).Div(
                  new(big.Int).Mul(I, R),
                  I_sum,
               )
               if O := new(big.Int).SetUint64(io.O); O_next.Cmp(O) > 0 {
                  O_next = new(big.Int).Add(O, new(big.Int).Div(
                     new(big.Int).Mul(I, R_left),
                     I_sum,
                  ))
               }
               if !O_next.IsUint64() {
                  ok = false
                  break
               }
               state.O_next[node_id] = O_next.Uint64()
            }
            if !ok {
               fmt.Println("error in calculation, current state must be behind fsm state,
retrying...")
               continue
            }
            apply_mu.Lock()
            if apply_val == nil {
               state.Version++
            }
            apply_val = util.MustEncodeToJson(state)
            apply_mu.Unlock()
            apply_signal.Signal()
         }
      }()
      return self
   }

   func (self *CALC) AcceptFSMState(state *SystemState) {
      state = state.DeepCopy()
      defer util.LockUnlock(&self.fsm_state_mu)()
```

```
    self.fsm_state = state
}

func (self *CALC) AcceptNodeState(node_state *NodeState) {
    defer util.LockUnlock(self.node_states_mu.RLocker())()
    self.node_states.Store(node_state.NodeID, node_state.IO)
}

func (self *CALC) Stop() bool {
    if !atomic.CompareAndSwapUint32(&self.status, 0, 1) {
        return false
    }
    for atomic.LoadInt32(&self.subprocess_cnt) != 0 {
        runtime.Gosched()
    }
    return true
}
```

data.go

```
package diplom_go

import (
    "math"
    "math/big"

    "github.com/hashicorp/raft"
)

type NodeID = raft.ServerID

type IO = struct{ I, O uint64 }

type NodeState = struct {
    NodeID NodeID
    IO
}

type SystemState struct {
    Version    uint64
    NodeStates map[NodeID]IO
    O_next     map[NodeID]uint64
}

func (self *SystemState) Init() *SystemState {
    self.NodeStates = make(map[NodeID]IO)
    self.O_next = make(map[NodeID]uint64)
    return self
}

func (self *SystemState) DeepCopy() *SystemState {
    ret := &SystemState{
        Version:    self.Version,
        NodeStates: make(map[NodeID]IO, len(self.NodeStates)),
        O_next:     make(map[NodeID]uint64, len(self.O_next)),
    }
    for k, v := range self.NodeStates {
        ret.NodeStates[k] = v
    }
    for k, v := range self.O_next {
        ret.O_next[k] = v
```

```
    }
    return ret
}

var R = new(big.Int).SetUint64(math.MaxUint64)
```

env.go

```
package diplom_go

import (
    "diplom_go/util"
    "encoding/json"
    "math/rand"
    "os"
    "runtime"
    "sync"
    "sync/atomic"
    "time"
)

type ENV struct {
    cfg ENVConfig
    IO
    io_mu          sync.Mutex
    O_next         uint64
    version        uint64
    update_mu      sync.Mutex
    status         uint32
    subprocess_cnt int32
}
type ENVConfig = struct {
    JournalPath        string
    IChangeProbability float64
    UpdateApplyPeriod  time.Duration
}

type ENVJournalRecord = struct {
    Time    time.Time
    O       uint64
    Version uint64
}

func (self *ENV) Init(cfg ENVConfig) *ENV {
    util.Assert(0 <= cfg.IChangeProbability && cfg.IChangeProbability < 1)
    self.cfg = cfg
    self.I = rand.Uint64()
    journal, err := os.OpenFile(self.cfg.JournalPath, os.O_APPEND|os.O_CREATE|os.O_WRONLY,
0777)
    util.Check(err)
    var last_record []byte
    util.ReadLines(self.cfg.JournalPath, func(bytes []byte) bool {
        last_record = bytes
        return true
    })
    if last_record != nil {
        var rec ENVJournalRecord
        util.Check(json.Unmarshal(last_record, &rec))
        self.O_next, self.O, self.version = rec.O, rec.O, rec.Version
    }
    atomic.AddInt32(&self.subprocess_cnt, 1)
```

```
   go func() {
      defer atomic.AddInt32(&self.subprocess_cnt, -1)
      defer journal.Close()
      for atomic.LoadUint32(&self.status) != 1 {
         time.Sleep(self.cfg.UpdateApplyPeriod)
         self.update_mu.Lock()
         O_next, version := self.O_next, self.version
         self.update_mu.Unlock()
         self.io_mu.Lock()
         O := self.O
         self.io_mu.Unlock()
         if O == O_next {
            continue
         }
         _, err := journal.Write(util.MustEncodeToJson(ENVJournalRecord{time.Now(), O_next,
version}))
         util.Check(err)
         _, err = journal.WriteString("\n")
         util.Check(err)
         util.Check(journal.Sync())
         self.io_mu.Lock()
         self.O = O_next
         self.io_mu.Unlock()
      }
   }()
   return self
}

func (self *ENV) Update(O_next, version uint64) {
   defer util.LockUnlock(&self.update_mu)()
   if self.version < version {
      self.O_next, self.version = O_next, version
   }
}

func (self *ENV) GetIO() IO {
   defer util.LockUnlock(&self.io_mu)()
   if rand.Float64() < self.cfg.IChangeProbability {
      self.I = rand.Uint64()
   }
   return self.IO
}

func (self *ENV) Stop() bool {
   if !atomic.CompareAndSwapUint32(&self.status, 0, 1) {
      return false
   }
   for atomic.LoadInt32(&self.subprocess_cnt) != 0 {
      runtime.Gosched()
   }
   return true
}
```

fsm.go

```
package diplom_go

import (
   "diplom_go/util"
   "encoding/json"
   "io"
   "sync"
```

```
    "github.com/hashicorp/raft"
)

type FSM struct {
    state    *SystemState
    init_mu  sync.Mutex
    listener FSMListener
}
type FSMListener = struct {
    OnInitial, OnUpdated, OnDynamicallyRestored func(*SystemState)
}

func (self *FSM) Init() *FSM {
    self.init_mu.Lock()
    self.state = new(SystemState).Init()
    return self
}

func (self *FSM) ApplyBatch(logs []*raft.Log) []interface{} {
    defer util.LockUnlock(&self.init_mu)()
    updated := false
    for _, log := range logs {
        if log.Type == raft.LogCommand {
            val := new(SystemState).Init()
            util.Check(json.Unmarshal(log.Data, val))
            if val.Version == self.state.Version+1 {
                self.state = val
                updated = true
            }
        }
    }
    if updated && self.listener.OnUpdated != nil {
        self.listener.OnUpdated(self.state)
    }
    return make([]interface{}, len(logs))
}

func (self *FSM) Apply(*raft.Log) interface{} { panic("N/A") }

func (self *FSM) Snapshot() (raft.FSMSnapshot, error) {
    defer util.LockUnlock(&self.init_mu)()
    return snapshot(util.MustEncodeToJson(&self.state)), nil
}

func (self *FSM) Restore(src io.ReadCloser) error {
    self.state = new(SystemState).Init()
    if err := json.NewDecoder(src).Decode(self.state); err != nil {
        return err
    }
    if self.listener.OnDynamicallyRestored != nil {
        self.listener.OnDynamicallyRestored(self.state)
    }
    return src.Close()
}

func (self *FSM) Unlock(listener FSMListener) {
    self.listener = listener
    if self.listener.OnInitial != nil {
        self.listener.OnInitial(self.state)
    }
    self.init_mu.Unlock()
}

type snapshot []byte
```

```
func (self snapshot) Persist(sink raft.SnapshotSink) error {
   if n, err := sink.Write(self); err != nil {
      return err
   } else {
      util.Assert(n == len(self))
   }
   return sink.Close()
}

func (self snapshot) Release() {}
```

node.go

```
package diplom_go

import (
   "diplom_go/util"
   "diplom_go/util/jsonrpc_http"
   "os"
   "path"
   "runtime"
   "sync"
   "sync/atomic"
   "time"
)

type Node struct {
   cfg            NodeConfig
   raft           *RAFT
   calc           *CALC
   env            *ENV
   rpc_server     *jsonrpc_http.Server
   apis           *APIs
   status         uint32
   subprocess_cnt int32
   start_stop_mu  sync.Mutex
}
type NodeConfig = struct {
   RAFT                   RAFTConfig
   CALC                   CALCConfig
   ENV                    ENVConfig
   ENVReadBroadcastPeriod time.Duration
}

func (self *Node) Init(cfg NodeConfig) *Node {
   self.cfg = cfg
   if len(self.cfg.ENV.JournalPath) == 0 {
      self.cfg.ENV.JournalPath = ENVJournalPath(self.cfg.RAFT.DataDir)
   }
   return self
}

func (self *Node) Start() (started bool) {
   if started = atomic.CompareAndSwapUint32(&self.status, 0, 1); !started {
      return
   }
   defer util.LockUnlock(&self.start_stop_mu)()
   util.Check(os.MkdirAll(self.cfg.RAFT.DataDir, 0777))
   fsm := new(FSM).Init()
   self.raft = new(RAFT).Init(self.cfg.RAFT, fsm)
   self.env = new(ENV).Init(self.cfg.ENV)
```

```
    fsm.Unlock(FSMListener{
        OnInitial: func(state *SystemState) {
            self.calc = new(CALC).Init(self.cfg.CALC, state, func(val []byte) {
                self.raft.Apply(val, 0)
            })
        },
        OnUpdated: func(state *SystemState) {
            self.env.Update(state.O_next[self.cfg.RAFT.NodeID], state.Version)
            self.calc.AcceptFSMState(state)
        },
        OnDynamicallyRestored: func(*SystemState) {
            panic("not expected")
        },
    })
    self.apis = new(APIs).BackendInit(self.raft, self.calc)
    self.rpc_server = new(jsonrpc_http.Server).Init(RPCPort(self.cfg.RAFT.Port), self.apis)
    defer atomic.AddInt32(&self.subprocess_cnt, 1)
    go func() {
        defer atomic.AddInt32(&self.subprocess_cnt, -1)
        for atomic.LoadUint32(&self.status) != 2 {
            time.Sleep(self.cfg.ENVReadBroadcastPeriod)
            self.apis.AcceptNodeState(&NodeState{self.cfg.RAFT.NodeID, self.env.GetIO()}, nil)
        }
    }()
    return
}

func (self *Node) Stop() (stopped bool, errs []error) {
    if stopped = atomic.CompareAndSwapUint32(&self.status, 1, 2); !stopped {
        return
    }
    defer atomic.StoreUint32(&self.status, 0)
    defer util.LockUnlock(&self.start_stop_mu)()
    for atomic.LoadInt32(&self.subprocess_cnt) != 0 {
        runtime.Gosched()
    }
    util.Assert(self.env.Stop())
    util.Assert(self.calc.Stop())
    errs = append(self.rpc_server.Stop())
    errs = append(errs, self.raft.Stop()...)
    return
}

func ENVJournalPath(data_dir string) string {
    return path.Join(data_dir, "env_journal.json")
}

func RPCPort(raft_port int) int {
    return raft_port + 1
}
```

raft.go

```
package diplom_go

import (
    "diplom_go/util"
    "os"
    "path/filepath"
    "time"
```

```
    "github.com/hashicorp/raft"
    raftboltdb "github.com/hashicorp/raft-boltdb"
)

type RAFT struct {
    *raft.Raft
    cfg            RAFTConfig
    transport      *raft.NetworkTransport
    log_store      *raftboltdb.BoltStore
    stable_store   *raftboltdb.BoltStore
    snapshot_store *raft.FileSnapshotStore
}
type RAFTConfig = struct {
    Port            int
    DataDir         string
    NodeID          NodeID
    TCPConnPoolSize int
    TCPConnTimeout  time.Duration
    *raft.Config
}

func (self *RAFT) Init(cfg RAFTConfig, fsm raft.FSM) *RAFT {
    self.cfg = cfg
    if self.cfg.Config == nil {
        self.cfg.Config = raft.DefaultConfig()
    } else {
        util.Assert(len(self.cfg.LocalID) == 0)
    }
    self.cfg.LocalID = self.cfg.NodeID
    self.cfg.ShutdownOnRemove = false
    raft_addr := util.ResolveLocalTCPAddress(cfg.Port)
    var err error
    self.transport, err = raft.NewTCPTransport(
        raft_addr.String(), raft_addr, cfg.TCPConnPoolSize, cfg.TCPConnTimeout, os.Stdout)
    util.Check(err)
    self.log_store, err = raftboltdb.NewBoltStore(filepath.Join(cfg.DataDir, "log_store"))
    util.Check(err)
    self.stable_store, err = raftboltdb.NewBoltStore(filepath.Join(cfg.DataDir,
"stable_store"))
    util.Check(err)
    self.snapshot_store, err = raft.NewFileSnapshotStore(filepath.Join(cfg.DataDir,
"snapshot_store"), 1, os.Stdout)
    util.Check(err)
    self.Raft, err = raft.NewRaft(
        self.cfg.Config,
        fsm,
        self.log_store,
        self.stable_store,
        self.snapshot_store,
        self.transport,
    )
    util.Check(err)
    return self
}

func (self *RAFT) Stop() (errs []error) {
    errs = append(errs, self.Shutdown().Error())
    self.transport.CloseStreams()
    errs = append(errs, self.transport.Close())
    errs = append(errs, self.log_store.Close())
    errs = append(errs, self.stable_store.Close())
    return
}
```

util/util.go

```
package util

import (
   "bufio"
   "encoding/json"
   "fmt"
   "io"
   "net"
   "os"
   "sync"
)

func Check(errors ...error) {
   for _, err := range errors {
      if err != nil {
         panic(err)
      }
   }
}

func ConstFNStr(val string) func() string {
   return func() string {
      return val
   }
}

func ResolveLocalTCPAddress(port int) *net.TCPAddr {
   addr, err_0 := net.ResolveTCPAddr("tcp", "localhost:"+fmt.Sprint(port))
   Check(err_0)
   l, err_1 := net.ListenTCP("tcp", addr)
   Check(err_1)
   addr = l.Addr().(*net.TCPAddr)
   Check(l.Close())
   return addr
}

func Assert(cond bool) {
   if !cond {
      panic("assertion failed")
   }
}

func MustEncodeToJson(val interface{}) []byte {
   ret, err := json.Marshal(val)
   Check(err)
   return ret
}

func ReadLines(path string, cb func([]byte) bool) error {
   f, err := os.Open(path)
   if err != nil {
      return err
   }
   r := bufio.NewReader(f)
   for {
      line, err := r.ReadBytes('\n')
      if err != nil {
         if err == io.EOF {
            break
         }
         return err
```

```
      }
      if !cb(line) {
         break
      }
   }
   return nil
}

func LockUnlock(l sync.Locker) func() {
   l.Lock()
   return l.Unlock
}

func MaxU64(x, y uint64) uint64 {
   if x > y {
      return x
   }
   return y
}
```

util/jsonrpc_http/jsonrpc.go

```
package jsonrpc_http

import (
   "bytes"
   "diplom_go/util"
   "encoding/json"
   "errors"
   "fmt"
   "io"
   "net"
   "net/http"
   "net/rpc"
   "net/rpc/jsonrpc"
   "reflect"
   "runtime"
   "strings"
   "unsafe"
)

var ErrNoEndpointAddress = errors.New("endpoint address empty")

type Client func(in, out interface{}) error
type TCPAddrFn func() *net.TCPAddr

func PopulateClients(services interface{}, tcp_addr TCPAddrFn) {
   http_client := new(http.Client)
   services_v := reflect.ValueOf(services).Elem()
   services_t := services_v.Type()
   for i, n := 0, services_t.NumField(); i < n; i++ {
      service_name := services_t.Field(i).Name
      service_ref := reflect.Indirect(services_v.Field(i))
      client_ref := service_ref.FieldByName("client").Addr()
      *(*Client)(unsafe.Pointer(client_ref.Pointer())) = func(in, out interface{}) error {
         addr := tcp_addr()
         if addr == nil {
            return ErrNoEndpointAddress
         }
         pc, _, _, ok := runtime.Caller(1)
         util.Assert(ok)
```

```
            method_name_parts := strings.Split(runtime.FuncForPC(pc).Name(), ".")
            method_name := method_name_parts[len(method_name_parts)-1]
            req_body := bytes.NewBuffer(nil)
            util.Check(json.NewEncoder(req_body).Encode(&struct {
               Method string         `json:"method"`
               Params [1]interface{} `json:"params"`
            }{
               service_name + "." + method_name,
               [...]interface{}{in},
            }))
            res, err := http_client.Post("http://"+addr.String(), "application/json", req_body)
            if err != nil {
               return err
            }
            var rpc_err_str string
            util.Check(json.NewDecoder(res.Body).Decode(&struct {
               Result interface{} `json:"result"`
               Error  *string     `json:"error"`
            }{
               out,
               &rpc_err_str,
            }))
            if len(rpc_err_str) != 0 {
               err = errors.New(rpc_err_str)
            }
            return err
         }
      }
   }

   type Server struct {
      rpc_server  *rpc.Server
      http_router *http.ServeMux
      listener    net.Listener
      stop_chan   chan error
   }

   func (self *Server) ServeHTTP(writer http.ResponseWriter, request *http.Request) {
      self.rpc_server.ServeCodec(jsonrpc.NewServerCodec(struct {
         io.ReadCloser
         io.Writer
      }{
         request.Body,
         writer,
      }))
   }

   func (self *Server) Init(port int, services interface{}) *Server {
      self.rpc_server = rpc.NewServer()
      services_v := reflect.ValueOf(services).Elem()
      services_t := services_v.Type()
      for i, n := 0, services_t.NumField(); i < n; i++ {
         service_ref := services_v.Field(i).Addr()
         self.rpc_server.RegisterName(services_t.Field(i).Name, service_ref.Interface())
      }
      self.http_router = http.NewServeMux()
      self.http_router.Handle("/", self)
      var err error
      self.listener, err = net.Listen("tcp", ":"+fmt.Sprint(port))
      util.Check(err)
      self.stop_chan = make(chan error, 1)
      go func() {
         self.stop_chan <- http.Serve(self.listener, self.http_router)
      }()
      return self
   }
```

```
func (self *Server) Addr() *net.TCPAddr {
   return self.listener.Addr().(*net.TCPAddr)
}

func (self *Server) Stop() []error {
   return []error{self.listener.Close(), <-self.stop_chan}
}
```

root_test.go

```
package diplom_go

import (
   "diplom_go/util"
   "encoding/json"
   "fmt"
   "math/big"
   "math/rand"
   "net"
   "os"
   "path"
   "sort"
   "strconv"
   "sync/atomic"
   "testing"
   "time"

   "github.com/hashicorp/raft"
)

func TestRoot(t *testing.T) {
   base_data_dir := path.Join("/tmp", "test_nodes")
   var configs [20]NodeConfig
   var api_clients [len(configs)]*APIs
   for i := range configs {
      id := strconv.Itoa(i)
      configs[i] = NodeConfig{
         RAFT: RAFTConfig{
            Port:            8100 + i*2,
            DataDir:         path.Join(base_data_dir, id),
            NodeID:          NodeID(id),
            TCPConnPoolSize: 5,
            TCPConnTimeout:  30 * time.Second,
         },
         CALC: CALCConfig{
            CyclePeriod: 100 * time.Millisecond,
         },
         ENV: ENVConfig{
            UpdateApplyPeriod:  100 * time.Millisecond,
            IChangeProbability: 0.10,
         },
         ENVReadBroadcastPeriod: 100 * time.Millisecond,
      }
      i := i
      api_clients[i] = new(APIs).FrontendInit(func() *net.TCPAddr {
         return &net.TCPAddr{
            IP:   []byte{127, 0, 0, 1},
            Port: RPCPort(configs[i].RAFT.Port),
         }
      })
```

```
    }
    util.Check(os.RemoveAll(base_data_dir))
    var nodes [len(configs)]*Node
    var subprocess_cnt int32
    for i := range nodes {
        fmt.Println("Initializing node", i)
        nodes[i] = new(Node).Init(configs[i])
        util.Assert(nodes[i].Start())
        if i == 0 {
            util.Check(api_clients[i].StartNewCluster(nil, nil))
            continue
        }
        i := i
        atomic.AddInt32(&subprocess_cnt, 1)
        go func() {
            defer atomic.AddInt32(&subprocess_cnt, -1)
            for attempt := 0; true; attempt++ {
                time.Sleep(1 * time.Second)
                if attempt == 300 {
                    panic("timeout")
                }
                err := api_clients[0].AddNode(&AddNodeReq{
                    NodeID:   configs[i].RAFT.NodeID,
                    NodeAddr: raft.ServerAddress("127.0.0.1:" +
strconv.Itoa(configs[i].RAFT.Port)),
                }, nil)
                if err == nil {
                    break
                }
                switch err.Error() {
                case
                    raft.ErrLeadershipLost.Error(),
                    raft.ErrNotLeader.Error(),
                    ErrNoLeader.Error(),
                    raft.ErrLeadershipTransferInProgress.Error():
                    continue
                }
                panic(err)
            }
        }()
    }
    for atomic.LoadInt32(&subprocess_cnt) != 0 {
        time.Sleep(1 * time.Second)
    }
    var stop_nodes uint32
    for _, node := range nodes {
        atomic.AddInt32(&subprocess_cnt, 1)
        node := node
        go func() {
            defer atomic.AddInt32(&subprocess_cnt, -1)
            for atomic.LoadUint32(&stop_nodes) != 1 {
                time.Sleep(time.Duration(3+rand.Intn(5)) * time.Second)
                stopped, _ := node.Stop()
                util.Assert(stopped)
                time.Sleep(time.Duration(1+rand.Intn(3)) * time.Second)
                util.Assert(node.Start())
                time.Sleep(time.Duration(3+rand.Intn(5)) * time.Second)
            }
            node.Stop()
        }()
    }
    time.Sleep(60 * time.Second)
    atomic.StoreUint32(&stop_nodes, 1)
    for atomic.LoadInt32(&subprocess_cnt) != 0 {
        time.Sleep(1 * time.Second)
    }
```

```
    type Record struct {
        NodeID NodeID
        ENVJournalRecord
    }
    var records []Record
    for _, cfg := range configs {
        util.Check(util.ReadLines(ENVJournalPath(cfg.RAFT.DataDir), func(bytes []byte) bool {
            var rec ENVJournalRecord
            util.Check(json.Unmarshal(bytes, &rec))
            records = append(records, Record{cfg.RAFT.NodeID, rec})
            return true
        }))
    }
    sort.Slice(records, func(i, j int) bool {
        return records[i].Time.Before(records[j].Time)
    })
    R_curr := new(big.Int).Set(R)
    O_by_node := make(map[NodeID]*big.Int)
    efficiency_sum := new(big.Float)
    for _, rec := range records {
        O := new(big.Int).SetUint64(rec.O)
        O_curr := O_by_node[rec.NodeID]
        if O_curr == nil {
            O_curr = new(big.Int)
        }
        R_curr = new(big.Int).Sub(new(big.Int).Add(R_curr, O_curr), O)
        util.Assert(R_curr.Sign() >= 0)
        O_by_node[rec.NodeID] = O
        efficiency_sum = new(big.Float).Add(
            efficiency_sum, new(big.Float).Sub(big.NewFloat(1.0), new(big.Float).Quo(
                new(big.Float).SetInt(R_curr),
                new(big.Float).SetInt(R))))
    }
    fmt.Println("efficiency % avg.:", new(big.Float).Quo(
        new(big.Float).Mul(efficiency_sum, big.NewFloat(100)),
        big.NewFloat(float64(len(records)))))
}
```