Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Програмний засіб для опрацювання поведінкових даних обчислювальних елементів розподіленої системи
Назва теми

КвРКІ.200249.20.02.23 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія»*
Назва

Виконав: студент IV курсу, група КІ2–20–2 _____ Х. В. Шульга
Підпис Ініціали, прізвище

Керівник _____ П.Г. Регіда
Підпис, дата Ініціали, прізвище

Нормоконтролер _____ І.О. Засорнова
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної інженерії та інформаційних систем _____ Т.О. Говорущенко
Підпис Ініціали, прізвище

« 6 » червня 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень БАКАЛАВР

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма ОСВІТНЯ ПРОГРАМА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри _____ Г.О.Говорущенко

"10" 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Шульзі Харітону Вячеславовичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Програмний засіб для опрацювання повідомлень від інформаційно-обчислювальних елементів розподіленої системи

керівник проекту (роботи) Регеда П.Г., ст. викладач

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 15.02.2024 р. № 8

2. Строк подання студентом проекту (роботи) на кафедру 03.06.2024 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Паралельні обчислення та їх роль в розподілених системах

Огляд програмних засобів для реалізації в системі

Програмна реалізація системи розподілених обчислень

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

Блок схема архітектури програмного забезпечення

Мережева комунікація в розподіленій системі

Алгоритм роботи програмного додатку

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийня
Нормоконтроль	Засорнова І.О., доцент кафедри КІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІП		

7. Дата видачі завдання «__10__» _____01_____ 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітк
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2024	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2024	виконан
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2024	виконан
4	Робота над розділом 2 – обґрунтування вибору програмних засобів	01.04.2024	виконан
5	Робота над розділом 3 – програмна реалізація системи розподілених обчислень	30.04.2024	виконан
6	Оформлення пояснювальної записки згідно вимог	31.05.2024	виконан
7	Попередній захист ВКР	30.05.2024	виконан
8	Захист ВКР на засіданні ЕК	Червень 2024 року	

Студент _____ Х. В. Шульга
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ П. Г. Регіда
Підпис Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 200249.20.02.23 ПЗ	Пояснювальна записка	71		
			Графічні матеріали			
2		КвРКІ 200249.20.02.23 Е8	Блок схема архітектури	1		
			Програмного			
			забезпечення			
3		КвРКІ 200249.20.02.23 Е8	Мережева комунікація	1		
			в розподіленій системі			
4		КвРКІ 200249.20.02.23 Е8	Алгоритм роботи			
			програмного додатку			

Зм	Арк	№ докум	Підпис	Дата	КвРКІ.200249.20.02.23 ВП		
Розробив		Шульга Х.В.		5.06	Відомість проекту	Літера	Аркуш 1 / Аркушів 1
Перевір.		Регіда П.Г.		5.06		У	
Н. контр.		Засорнова І.О.		5.06		ХНУ, КІ2-20-2	
Затв.		Говорущенко Т.О		06.06			

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмний засіб для опрацювання поведінкових даних обчислювальних елементів розподіленої системи».

Автор роботи: Шульга Харітон Вячеславович.

Керівник роботи: Регіда Павло Геннадійович.

Пояснювальна записка: 71 с., 28 рис., 1 табл., 4 дод., 60 джерел.

Графічна частина: 3 презентаційних слайдів.

ОБЧИСЛЮВАЛЬНИЙ ДОДАТОК, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ПОТОКИ, МЕРЕЖЕВІ СОКЕТИ.

Метою роботи є розробка програмного засобу для опрацювання поведінкових даних обчислювальних елементів розподіленої системи.

В цій роботі було розроблено програмний засіб для опрацювання поведінкових даних, з присутністю деталей для проведення моніторингу одиниці системи, з виводами результаті виконання завдання, та налаштуванням з'єднання з віддаленою системою.

Підпис студента

30.05.2024

Дата

ЗМІСТ

ВСТУП

У сучасному світі зростаюча потреба в обробці великих обсягів даних та виконанні складних обчислень в реальному часі породжує необхідність в розробці та впровадженні ефективних програмних засобів. Зокрема, розподілені системи, що базуються на паралельних обчисленнях, стають все більш популярними та важливими для вирішення складних завдань в різноманітних сферах, таких як наука, бізнес, медицина та інші.

Актуальність технологій, що лежать в основі розподілених систем та паралельних обчислень, неможливо переоцінити. Завдяки їм ми можемо швидше та ефективніше обробляти великі обсяги даних, розв'язувати складні задачі та реалізовувати новаторські проекти. Саме тому розробка програмних засобів, які дозволяють ефективно керувати та опрацьовувати поведінкові дані обчислювальних елементів розподіленої системи, є актуальним завданням сучасної інформаційної технології.

В рамках даної дипломної роботи пропонується розробка програмного засобу на платформі Node.js, спрямованого на опрацювання поведінкових даних обчислювальних елементів у розподіленій системі. Цей програмний засіб буде інтегровано з іншими компонентами системи, зокрема з балансувальником навантаження для паралельних обчислень.

Розподілені системи - це комплексне об'єднання апаратних та програмних засобів, яке включає в себе велику кількість вузлів, що працюють разом, забезпечуючи високу доступність, масштабованість та надійність системи в цілому. Паралельні обчислення, у свою чергу, є ключовим елементом в реалізації ефективної роботи розподілених систем, оскільки дозволяють розділити завдання на менші частини, які можуть виконуватися паралельно на різних вузлах системи, зменшуючи час виконання та підвищуючи продуктивність в цілому.

У цьому контексті розробка програмного забезпечення для опрацювання поведінкових даних обчислювальних елементів розподіленої системи стає необхідністю. Відбір та аналіз таких даних може допомогти в розумінні

працездатності та ефективності системи в цілому, а також виявленні можливих точок оптимізації та покращення.

Node.js є серверною технологією, побудованою на основі JavaScript V8 від Google. Він відомий своєю ефективністю та можливістю обробки великих обсягів запитів з високою швидкістю. Однією з ключових переваг Node.js є його асинхронний характер, що дозволяє ефективно керувати багатозадачністю та реагувати на події в реальному часі. Ця особливість надає не лише швидкість та ефективність, але і гнучкість та можливість реалізації складних функцій, необхідних для взаємодії з іншими компонентами системи та опрацювання великих обсягів даних.

Метою кваліфікаційної роботи є забезпечення функціонування програмного додатку для виконання обчислювальних задач, що надсилаються керуючим центральним сервером, та опрацювання поведінкових даних цих елементів.

Поставлена мета досягається вирішенням наданої задачі: проєктування та розроблення програмного додатку збору поведінкових даних та виконання обчислювальних задач для розподіленої системи.

Об’єктом дослідження є процес збору поведінкових даних та виконання обчислювальних задач.

Предметом дослідження є програмний додаток для збору поведінкових даних та виконання обчислювальних задач.

Для досягнення поставленої мети використовуються методи системного програмування, засоби для роботи із мережею та передачі інформації через неї, підходи збору та аналізу поведінкових характеристик обчислювального елементу.

Практичне значення має спроектований та реалізований програмний додаток для виконання обчислювальних задач, що надходять із центрального керуючого серверу, та збір та опрацювання поведінкових даних хосту на якому він виконується.

1 ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ ТА ЇХ РОЛЬ В РОЗПОДІЛЕНИХ СИСТЕМАХ

1.1.Принципи та класифікація розподілених обчислень

Визначення та основні принципи паралельних обчислень відіграють важливу роль у розподілених системах, де потрібно обробляти великі обсяги даних або виконувати складні обчислення. Паралельні обчислення означають виконання кількох завдань одночасно з метою збільшення швидкості та ефективності обробки. Основний принцип паралельних обчислень полягає в розподіленні обчислювальних завдань між різними процесорами або вузлами, які можуть виконувати їх одночасно.

Цей підхід дозволяє використовувати ресурси системи максимально ефективно, зменшуючи час виконання завдань і покращуючи загальну продуктивність.

В паралельних обчисленнях важливо враховувати принципи синхронізації, щоб уникнути конфліктів та непорозумінь між обчислювальними процесами [1-3].

Одним із ключових принципів є розділення завдань на незалежні частини, які можна виконувати паралельно. Це дозволяє кожному процесору або вузлу системи працювати над своєю частиною завдання, що покращує загальний час виконання [4-5]. Ще одним важливим принципом є збір результатів та їх об'єднання у фінальний вихідний результат. Це може включати синхронізацію даних між різними процесорами або вузлами, а також аналіз та обробку отриманих результатів [6-7].

Крім того, важливо розуміти та враховувати характеристики обчислювальних ресурсів, такі як кількість ядер, швидкість обробки даних, доступна пам'ять та інші фактори, при розподілі завдань між ними. Це допомагає забезпечити оптимальне використання ресурсів і покращити продуктивність системи в цілому.

Існує два типи паралельних обчислень:

– з поділом даних: дані задачі розбиваються на частини, які обробляються одночасно;

– з поділом функцій: задача розбивається на підзадачі, які виконуються одночасно.

Важлива роль в розподілених системах:

– підвищення продуктивності: значно скорочується час, необхідний для розв'язання задачі;

– масштабування: система легко масштабується за рахунок додавання нових комп'ютерів;

– підвищення надійності: задача може продовжувати оброблятися, навіть якщо один із комп'ютерів вийде з ладу.

Приклади використання:

– обробка веб-запитів: веб-сервери обробляють декілька запитів одночасно;

– аналіз даних: аналіз великих наборів даних;

– наукові обчислення: моделювання клімату, розв'язання складних задач.

Також важливо пам'ятати, що розробка та управління розподіленими системами складніші, ніж централізованими. Розподілені системи можуть бути більш уразливими до атак. Розробка та експлуатація розподілених систем може бути достатньо дорогою [8-10].

Типи паралельних обчислень включають різноманітні підходи та стратегії розподілу обчислювальних завдань між різними процесорами або вузлами системи [11].

Кожен з цих типів має свої особливості та застосування в різних областях. Деякі з найпоширеніших типів паралельних обчислень:

Розподілені обчислення: Цей тип обчислень передбачає розподіл завдань між різними комп'ютерами або вузлами в мережі. Кожен вузол виконує свою частину обчислень, і результати об'єднуються для отримання загального результату.

Розподілені обчислення широко використовуються у наукових дослідженнях, обробці великих обсягів даних та інших областях, де необхідно обробляти складні завдання.

Багатоядерні системи: цей тип обчислень передбачає використання комп'ютерів або серверів з багатьма процесорами або ядрами. Кожне ядро може виконувати свої обчислення одночасно з іншими ядрами, що дозволяє прискорити обробку даних. Багатоядерні системи широко використовуються в серверних додатках, великих базах даних та інших завданнях, де потрібна висока продуктивність.

Кластерні системи: цей тип обчислень передбачає об'єднання декількох незалежних комп'ютерів або серверів у кластер з метою виконання спільних завдань. Кластери зазвичай використовуються в великих обчислювальних центрах та наукових лабораторіях для виконання складних обчислювальних завдань.

Грід-системи: цей тип обчислень передбачає використання розподілених ресурсів з різних джерел для виконання обчислень. Грід-системи можуть включати в себе різні типи обчислювальних ресурсів, такі як комп'ютери, сервери, сенсори та інші, і використовуються для виконання великих обчислювальних завдань [12-15].

Також часто паралельні обчислення поділяють на два типи: обчислення з поділом даних та обчислення з поділом функцій. Обчислення з поділом даних використовують коли дані задачі можна розділити на декілька частин, які можна обробити одночасно. Наприклад, при обробці зображення його можна розбити на пікселі, які потім обробляються паралельно. В такому підході дані можна легко розбивати на частини та можна використовувати різні алгоритми для обробки різних частин даних. Також при обчисленні з поділом даних можна досить легко масштабувати систему, додаючи більше процесорів. Але також в такому підході є і недоліки, до яких можна віднести складність при процесі синхронізації процесів, а також, що не всі задачі можна легко розбити на частини. Також до всього, можлива втрата продуктивності через затримки при синхронізації.

Іншим типом є обчислення з поділом функцій, коли задачу можна розділити на декілька підзадач, які можна виконати одночасно. Наприклад, при розрахунку траєкторії польоту ракети можна розділити задачу на розрахунок траєкторії, розрахунок опору повітря та розрахунок впливу гравітації. Тут можна використовувати різні алгоритми для виконання різних підзадач, а також можна легко масштабувати систему, додаючи більше обчислювальних елементів, та додатково, можна використовувати засоби для виконання різних підзадач. До недоліків можна віднести складність розбиття задачі на підзадачі, складність синхронізації процесорів, та ситуація де можлива втрата продуктивності через затримки при синхронізації.

Кожен з цих типів паралельних обчислень має свої переваги та обмеження, і вибір конкретного типу залежить від потреб та характеристик конкретного проєкту.

1.2.Застосування паралельних обчислень в розподілених системах

Застосування паралельних обчислень в розподілених системах помітно розширює можливості при побудові такої системи та може бути використане у різних галузях, включаючи наукові дослідження, інженерію, фінанси, медицину та інші. Розглянемо деякі приклади застосування паралельних обчислень та пов'язані з ними ризики та витрати [16].

Один з найпоширеніших прикладів застосування паралельних обчислень в розподілених системах – це наукові дослідження, такі як моделювання клімату, аналіз геномних даних або обчислення складних фізичних моделей. Використання паралельних обчислень дозволяє дослідникам прискорити обчислення та обробку даних, що дозволяє їм швидше робити висновки та розробляти нові наукові теорії.

Ще одним прикладом є фінансовий аналіз та прогнозування, де паралельні обчислення використовуються для обробки великих обсягів фінансових даних та моделювання різних фінансових сценаріїв. Це дозволяє фінансовим аналітикам швидше приймати рішення та реагувати на зміни на фінансових ринках [17-20].

Однак застосування паралельних обчислень може також призвести до певних ризиків та витрат. Наприклад, розробка та підтримка паралельних обчислень може вимагати значних витрат на інфраструктуру, програмне забезпечення та кваліфікацію персоналу. Крім того, виникають ризики пов'язані зі синхронізацією даних та управлінням ресурсами в розподілених системах, які можуть призвести до непередбачуваних результатів або втрати ефективності.

Крім того, паралельні обчислення можуть бути супроводжені ризиками щодо безпеки даних, так як розподілені системи зазвичай використовуються для обробки великих обсягів конфіденційної інформації. Це може створювати потенційні точки атаки для зловмисників та вимагати додаткових заходів безпеки та захисту даних.

Також, важливо враховувати ризики, пов'язані зі складністю управління паралельними обчисленнями та їхньою масштабованістю. При розробці та впровадженні паралельних обчислень можуть виникати проблеми з управлінням ресурсами, розподілом завдань, а також виявлення та виправлення помилок у програмному забезпеченні

Крім того, існує ризик виникнення конфліктів між різними частинами системи при використанні паралельних обчислень. Це може включати в себе конфлікти при доступі до спільних ресурсів, несправності у взаємодії між різними частинами програмного забезпечення, а також проблеми з конкуренцією за ресурси системи.

У разі неправильної реалізації паралельних обчислень може виникнути істотний негативний вплив на продуктивність системи. Неефективне використання ресурсів, надмірне розподілення завдань або недостатня синхронізація можуть призвести до збільшення часу виконання завдань або навіть до збоїв у роботі системи [21-24].

Щоб зменшити ризики та витрати, пов'язані зі впровадженням паралельних обчислень, необхідно провести детальний аналіз потреб системи, розробити ефективну стратегію розподілу завдань, забезпечити ефективне управління

ресурсами та використовувати надійні методи синхронізації та управління конфліктами.

Додатковими витратами при впровадженні паралельних обчислень можуть бути витрати на обладнання, програмне забезпечення та навчання персоналу. Розгортання паралельних обчислень може вимагати придбання спеціалізованого обладнання, такого як високопродуктивні сервери або графічні прискорювачі, які забезпечать необхідну обчислювальну потужність для виконання завдань.

Крім того, можуть бути витрати на придбання та ліцензування програмного забезпечення, що підтримує паралельні обчислення, такого як бібліотеки для розподілених обчислень або інструменти для автоматизації процесу [25-27].

Ризики включають в себе можливість виникнення помилок та вразливостей у програмному забезпеченні, які можуть призвести до збоїв в роботі системи або порушень безпеки даних. Однією з таких загроз є конфлікти при розділенні та обробці даних, особливо при використанні спільних ресурсів. Без належної уваги до синхронізації та управління конфліктами може виникнути непередбачене поведінка системи або втрата цілісності даних.

Одним із способів зменшення ризиків є впровадження надійних механізмів контролю якості та тестування програмного забезпечення, що використовується в паралельних обчисленнях. Ретельне тестування може допомогти виявити та усунути помилки перед впровадженням системи в експлуатацію.

Також важливо мати ефективну стратегію реагування на можливі проблеми та збої, що можуть виникнути під час роботи паралельних обчислень. Це включає в себе розробку планів відновлення та резервного копіювання даних, а також тренування персоналу для швидкого виявлення та усунення проблем.

Враховуючи всі ці фактори, компанії можуть успішно впроваджувати паралельні обчислення в своїх розподілених системах, забезпечуючи високу продуктивність та ефективність обробки даних, при цьому мінімізуючи ризики та витрати [28-30].

Однак, забезпечення технічної ефективності - це лише частина успіху. В сучасному світі, де користувач стоїть у центрі будь-якого сервісу, критично

важливо розуміти його потреби та очікування. Саме тут на передній план виходить опрацювання поведінкових даних у розподіленій системі. Ця технологія дозволяє не тільки оптимізувати технічні аспекти системи, але й адаптувати її до реальних потреб користувачів, що є ключовим для довгострокового успіху.

Опрацювання поведінкових даних в розподіленій системі є важливою задачею з точки зору аналізу користувачів та виявлення їхньої поведінки для подальшого вдосконалення сервісів та продуктів. Поведінкові дані включають в себе інформацію про дії та взаємодію користувачів з системою, такі як відвідування сторінок, кліки на кнопки, час проведений на сайті, виконані дії та інше. Ці дані можуть бути використані для розуміння поведінки користувачів, ідентифікації популярних функцій та сервісів, а також для покращення взаємодії з користувачами.

Важливість опрацювання поведінкових даних полягає у можливості зробити висновки про ефективність та придатність сервісу для користувачів. Аналіз поведінкових даних дозволяє виявити патерни поведінки користувачів, встановити тенденції та проблемні моменти в інтерфейсі або функціоналі, що можуть бути вдосконалені. Також ці дані можуть бути використані для персоналізації взаємодії з користувачами, наприклад, шляхом рекомендацій контенту або продуктів, які відповідають їхнім інтересам та поведінці.

Однак опрацювання поведінкових даних в розподіленій системі може бути викликаним завданням через великий обсяг і різноманітність даних. Розподілені системи можуть збирати дані з різних джерел та додатків, що може ускладнити їхнє оброблення та аналіз. Крім того, важливо забезпечити безпеку та конфіденційність поведінкових даних, оскільки вони містять особисту інформацію про користувачів.

Для успішного опрацювання поведінкових даних в розподіленій системі необхідно використовувати високопродуктивні алгоритми оброблення та аналізу даних, а також забезпечити ефективну систему зберігання та управління даними. Також важливо розробити чіткі процедури для захисту даних та виконання вимог щодо конфіденційності та безпеки.

Додатково, опрацювання поведінкових даних в розподіленій системі може допомогти підвищити рівень задоволення користувачів. Аналізуючи їхню поведінку та взаємодію з системою, можна ідентифікувати ключові аспекти, які впливають на їхню задоволеність, та розробити стратегії для поліпшення цих аспектів. Наприклад, якщо аналіз показує, що користувачі часто зупиняються на певному етапі процесу або відхиляються від певної функції, це може свідчити про проблеми з інтерфейсом або дизайном, які потрібно виправити для поліпшення взаємодії.

Більше того, опрацювання поведінкових даних може бути корисним для прогнозування та адаптації до змін у користувальницькому поведінці. Аналізуючи попередні дії та патерни взаємодії, система може передбачити майбутні дії користувачів та реагувати на них заздалегідь, надаючи персоналізовані рекомендації або послуги [31-33].

Однак, важливо враховувати етичні аспекти опрацювання поведінкових даних. Збір та аналіз особистої інформації повинен відбуватися з дотриманням законодавства про захист персональних даних та з урахуванням прав та приватності користувачів. Запобігання зловживанням зі сторони третіх осіб або несанкціонованого доступу до даних також є важливим завданням при опрацюванні поведінкових даних.

Таким чином, опрацювання поведінкових даних в розподіленій системі є ключовим елементом для вдосконалення продуктів та послуг, забезпечення високого рівня задоволення користувачів та прогнозування їхніх потреб. Однак важливо дотримуватися етичних стандартів та правил безпеки даних під час їхнього оброблення та аналізу [34-35].

У підсумку, опрацювання поведінкових даних в розподіленій системі є важливим завданням для аналізу та вдосконалення взаємодії з користувачами та покращення продуктів і сервісів. Врахування особливостей роботи з розподіленими даними та забезпечення безпеки та конфіденційності є ключовими аспектами при реалізації цього завдання.

1.3.Аналіз та збір поведінкових даних

Визначення поведінкових даних обчислювальних елементів в розподілених системах передбачає аналіз різних аспектів їх функціонування. Для цього можуть використовуватися різні джерела інформації, включаючи вбудовані датчики, системи моніторингу та інші інструменти аналізу системи. Наприклад, моніторинг температури та вологості може допомогти у виявленні перегріву та забезпечити оптимальні умови роботи обладнання.

Дані про використання ресурсів, такі як центральний процесор, пам'ять та диск, є ключовими для оцінки ефективності роботи обчислювальних елементів. Моніторинг стану компонентів, таких як вентилятори та жорсткі диски, допомагає вчасно виявляти проблеми та забезпечувати надійну роботу системи.

Значення поведінкових даних обчислювальних елементів полягає також у виявленні та вирішенні можливих проблем, таких як перевантаження мережі чи процесора. Це дозволяє уникнути витрат на ремонт та забезпечує безперебійну роботу системи.

Для отримання поведінкових даних обчислювальних елементів можуть використовуватися різні інструменти та технології, включаючи вбудовані модулі операційних систем, програми моніторингу та спеціалізовані платформи для аналізу даних. Такий аналіз може бути важливим для планування ресурсів та оптимізації роботи розподіленої системи.

Важливою складовою визначення поведінкових даних обчислювальних елементів є їх контекст. Наприклад, поведінкові дані серверів можуть включати в себе інформацію про трафік мережі, навантаження на процесор, використання пам'яті та інші характеристики. Визначення контекстуальної інформації допомагає зрозуміти, як обчислювальні елементи взаємодіють один з одним та з системою в цілому.

Поведінкові дані користувачів, зібрані на клієнтських пристроях, самі по собі не мають великої цінності без передачі на серверну частину для подальшої обробки. Дані можуть надходити з декількох клієнтів одночасно, на одну

серверну одиницю, рисунок 1.1, а може і одночасно на декілька серверних одиниць, залежить від потреб розробника програмного забезпечення.

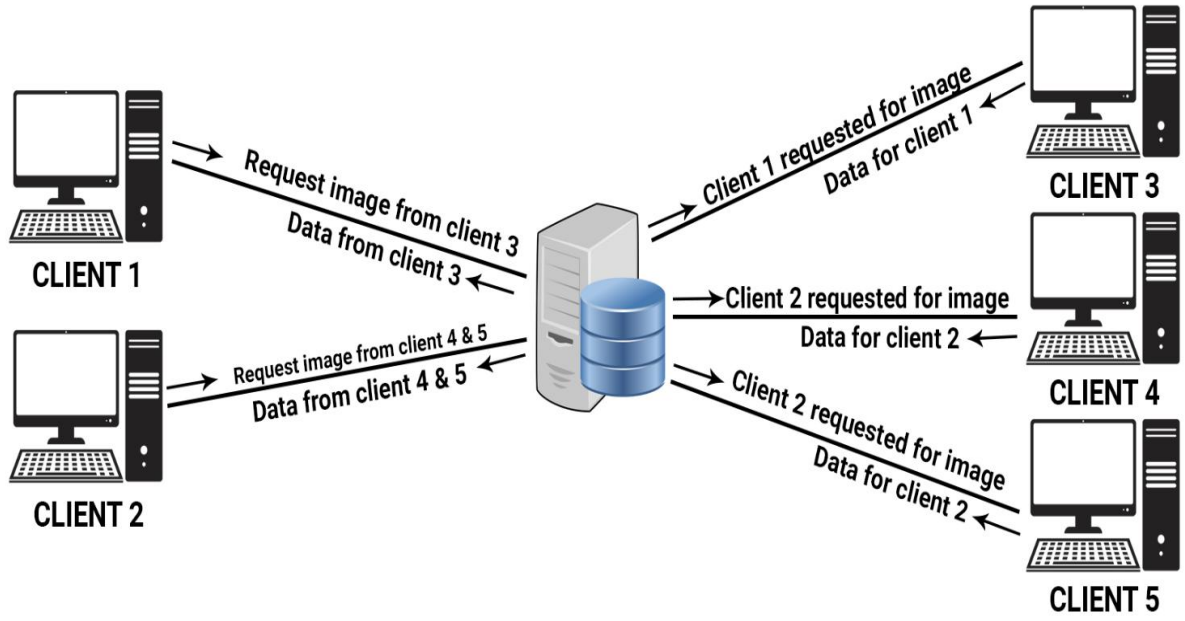


Рисунок 1.1 – Мережева передача даних

Дані про поведінку обчислювальних елементів можуть бути використані для розробки стратегій управління ресурсами. Наприклад, на основі аналізу використання ресурсів можна розробити алгоритми автоматичного масштабування системи для забезпечення оптимального рівня продуктивності та ефективності.

Застосування поведінкових даних обчислювальних елементів також дозволяє виявляти аномальну поведінку та вчасно реагувати на можливі проблеми. Наприклад, аналіз несподіваної зміни навантаження на сервер може вказувати на атаку або несправність обладнання, що дозволяє оперативно вжити заходів для запобігання проблемам.

Важливо також враховувати етичні аспекти збору та використання поведінкових даних обчислювальних елементів. Необхідно дотримуватися принципів конфіденційності та захисту приватності користувачів, а також уникати зловживань або недопустимого використання отриманих даних.

У контексті розподілених систем важливо визначити, які саме дані є критичними для опрацювання та моніторингу. Наприклад, для системи електронної комерції можуть бути важливими дані про транзакції та відвідування сайту, тоді як для медичних систем це можуть бути дані про стан пацієнтів та роботу медичного обладнання. Вибір та пріоритетність даних для опрацювання повинні відповідати конкретним вимогам та завданням системи.

Застосування поведінкових даних обчислювальних елементів у розподілених системах дозволяє вдосконалити їх функціональність та ефективність. Надійний моніторинг та аналіз цих даних може бути ключовим для успішного впровадження та експлуатації розподіленої системи.

В додачу, для отримання поведінкових даних можна розглянути використання інших інструментів, таких як аналітика веб-сайту, системи трасування запитів або моніторингу використання ресурсів. Ці інструменти допоможуть отримати детальну інформацію про взаємодію користувачів з системою та процеси, що відбуваються в розподіленій системі.

Крім того, для забезпечення ефективного опрацювання поведінкових даних важливо враховувати принципи та методики аналізу даних, такі як обробка даних в реальному часі, статистичний аналіз, машинне навчання та інші. Використання цих методів дозволяє отримати інсайди з поведінкових даних та вжити відповідних заходів для вдосконалення роботи розподіленої системи.

Використання поведінкових даних обчислювальних елементів у розподілених системах відкриває широкий спектр теоретичних переваг. Насамперед, це дозволяє оптимізувати продуктивність та надійність системи шляхом точного визначення вузьких місць, передбачення відмов обладнання та швидкого реагування на аномалії.

Крім того, такий підхід сприяє адаптивному масштабуванню ресурсів та створенню оптимальних умов експлуатації, що підвищує ефективність та довговічність системи**.**

Значення та використання поведінкових даних у розподілених системах є критичним для розуміння та оптимізації їх функціонування. Отримання і аналіз цих даних може мати ключове значення для різних аспектів розробки та управління розподіленими системами. Розглянемо деякі з них.

Вдосконалення продуктивності. Поведінкові дані можуть допомогти вдосконалити продуктивність розподіленої системи шляхом виявлення та виправлення проблемних моментів. Аналіз поведінкових патернів користувачів та реакції системи на їх дії дозволяє виявляти та усувати потенційні ситуації, що можуть призвести до зниження продуктивності або негативного впливу на користувачів.

Забезпечення безпеки. Поведінкові дані можуть використовуватися для виявлення підозрілих або небажаних дій у системі, таких як кібератаки або порушення безпеки. Аналізуючи поведінкові патерни та використання системи, можна вчасно виявляти та реагувати на можливі загрози безпеці.

Покращення масштабованості та надійності. Аналіз поведінкових даних дозволяє покращити масштабованість та надійність розподілених систем. На основі зібраних даних можна розробляти стратегії автоматичного масштабування ресурсів, щоб система могла ефективно пристосовуватися до змінних навантажень.

Управління ресурсами. Використання поведінкових даних дозволяє ефективно управляти ресурсами розподіленої системи, такими як обчислювальна потужність, мережевий трафік та використання пам'яті. Аналізуючи патерни використання ресурсів, можна виявити можливості для оптимізації та раціоналізації їх використання, що може призвести до зменшення витрат та підвищення ефективності.

Прогнозування та планування. На основі аналізу поведінкових даних можна розробляти прогнози щодо майбутньої активності та потреб користувачів, що дозволяє заздалегідь планувати та вирішувати потреби розподіленої системи. Це може включати планування ресурсів, масштабування системи, впровадження нових функцій або послуг та інше.

Оптимізація алгоритмів та архітектури. Аналіз поведінкових даних дозволяє розробникам систем вдосконалювати алгоритми та архітектуру програмного забезпечення. Наприклад, на основі виявлених патернів використання можна оптимізувати роботу алгоритмів обробки даних або вдосконалити розподіл завдань між обчислювальними вузлами.

1.4.Вибір та пріоритетність даних

Вибір та пріоритетність даних для опрацювання у розподілених системах є критично важливою задачею. Прийняття рішень щодо того, які дані обробляти та в якому порядку, може значно вплинути на ефективність та продуктивність системи.

У контексті паралельних обчислень в розподілених системах пріоритетність даних може бути визначена кількома факторами. Один з них - важливість даних для поточної операції або завдання. Деякі дані можуть бути критичними для успішного виконання завдання, тоді як інші можуть бути менш важливими і можуть оброблятися з меншим пріоритетом.

Крім того, пріоритетність даних може бути визначена також швидкодією та доступністю даних. Наприклад, дані, які легко доступні та можуть бути швидко оброблені, можуть мати вищий пріоритет порівняно з тими, що вимагають більших обчислювальних ресурсів або часу для обробки.

У розподілених системах, де тисячі або навіть мільйони клієнтів одночасно надсилають дані про свою поведінку, здатність Node.js обробляти велику кількість конкурентних з'єднань стає ключовою перевагою. Завдяки своєму однопоточному, керованому подіями дизайну, Node.js може легко

масштабуватися, дозволяючи одному серверу обробляти десятки тисяч одночасних з'єднань без значного зниження продуктивності. Це означає, що навіть при раптовому сплеску активності користувачів, наприклад, під час великого розпродажу чи прем'єри потокового відео, система на базі Node.js може ефективно збирати та обробляти величезні обсяги поведінкових даних, не вимагаючи при цьому надмірних апаратних ресурсів.

Вибір модулів Node.js та пакету systeminformation для отримання поведінкових даних обчислювальних елементів розподіленої системи може бути кращим в деяких випадках порівняно з альтернативними рішеннями, такими як Prometheus, Grafana або ELK Stack.

Ось кілька причин, чому було обрано саме цю технологію:

– простота використання та інтеграції: Модулі Node.js та пакет systeminformation можуть бути легко інтегровані в існуючий код програми на JavaScript. Вони працюють в середовищі Node.js, що дає можливість зручно взаємодіяти з іншими компонентами програми та отримувати доступ до потужних інструментів JavaScript;

– нативна підтримка системних операцій: Модуль os у Node.js надає простий спосіб отримання базової інформації про операційну систему та обладнання. Це дозволяє ефективно взаємодіяти з різними платформами та отримувати достатньо інформації для аналізу поведінки системи;

– широкий функціонал пакету systeminformation: Пакет systeminformation надає розширений функціонал для отримання різноманітної інформації про систему, такої як інформація про процесор, пам'ять, диски, мережеві інтерфейси та багато іншого.

Це дозволяє отримувати більш детальну та комплексну інформацію, що може бути корисною для аналізу поведінкових даних.

Ключову роль у цьому виборі відіграє npm (Node Package Manager) - потужний інструмент, який є серцем екосистеми Node.js. npm - це не просто репозиторій пакетів, а найбільша у світі екосистема бібліотек з відкритим кодом, дивіться рисунок 1.2.

Ця величезна різноманітність означає, що для майже будь-якої задачі, включаючи збір та аналіз поведінкових даних, у npm вже є готове, протестоване і часто добре оптимізоване рішення.

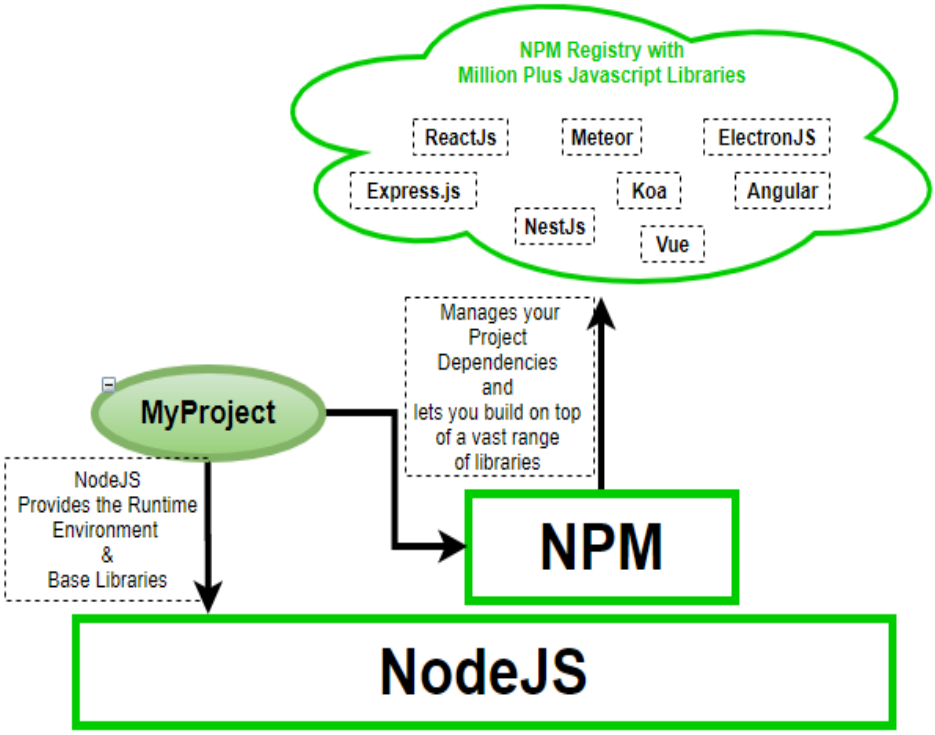


Рисунок 1.2 – роль npm у керуванні залежностями проєкту Node.js

Легкість встановлення та налаштування: використання модулів Node.js та пакету systeminformation може бути менш складним у порівнянні з іншими рішеннями, які можуть вимагати складних процесів встановлення та налаштування, таких як налаштування серверів або баз даних.

Таким чином, використання вбудованих модулів Node.js або пакету systeminformation дозволяє отримувати поведінкові дані про роботу розподіленої системи, що може бути корисним для подальшого аналізу та оптимізації її роботих [36-38].

Пакет systeminformation дозволяє отримувати детальну інформацію про пам'ять системи. Це включає не лише загальний обсяг та використання оперативної пам'яті, але й розподіл пам'яті за різними категоріями, такими як кеш та буфери. Завдяки цьому можна ефективно аналізувати, як саме

використовується пам'ять у системі, і виявляти можливі вузькі місця, що можуть впливати на продуктивність.

Також, збір даних про дискові системи, включаючи статистику вводу/виводу та швидкість операцій з дисками, дозволяє отримати повне уявлення про роботу та ефективність використання дискового простору.

Мережева інформація, зібрана за допомогою systeminformation, надає змогу аналізувати трафік та роботу мережевих інтерфейсів. Це включає дані про пакети, що передаються і приймаються, а також інформацію про IP-адреси та шлюзи. Така інформація є критично важливою для забезпечення безперебійної роботи мережі та своєчасного виявлення можливих проблем. Крім того, можливість отримувати дані про операційну систему, такі як версія ОС, час роботи системи та активні користувачі, забезпечує повний огляд на стан системи в цілому. Ці функції роблять systeminformation універсальним інструментом для всебічного моніторингу та діагностики комп'ютерних систем.

Окрім базових даних про процесор, пам'ять, диски та мережу, пакет systeminformation дозволяє збирати ще більше детальної інформації, яка може бути корисною для глибокого аналізу та моніторингу системи. Наприклад, можна отримати інформацію про батарею, що включає статус зарядки, рівень заряду та залишковий час роботи від батареї. Це особливо важливо для мобільних пристроїв та ноутбуків, де час автономної роботи має критичне значення. Дані про батарею дозволяють не лише відстежувати поточний стан, але й прогнозувати потребу в заміні батареї або оптимізації енергоспоживання.

Засіб, надає можливість отримувати дані про операційну систему, такі як її версія, час роботи системи та інформація про поточних користувачів. Це допомагає адміністраторам систем контролювати статус операційної системи, виявляти можливі проблеми з оновленнями та забезпечувати безпеку системи, відслідковуючи, хто і коли використовує систему. Така інформація може бути важливою для аудиту безпеки та управління доступом, дозволяючи швидко реагувати на потенційні загрози.

Додатково, systeminformation може збирати дані про периферійні пристрої, такі як монітори, принтери та інші підключені пристрої.

Це включає інформацію про роздільну здатність екрану, тип підключення та статус пристроїв. Такі дані корисні для управління апаратними ресурсами та діагностики проблем з периферійним обладнанням. Можливість моніторити стан цих пристроїв у реальному часі дозволяє забезпечувати безперебійну роботу всієї системи, вчасно виявляти та усувати несправності, а також оптимізувати використання апаратних ресурсів.

В контексті даної роботи, розподіл даних біль детально розписано в таблиці 1.1.

Таблиця 1.1 – Вибір пріоритетних даних

Ключові дані	Додаткові дані
Навантаженість процесора	Температура процесора
Використання оперативної пам'яті	Статистика сторінкових файлів
Навантаження мережі	Інформація про монітори
Версія операційної системи	Дані про периферійні пристрої
Виробник процесора	Інформація про батарею
Кількість ядер	Статистика використання віртуальної пам'яті
Частота процесора	Дані про підключення USB-пристроїв
Загальний обсяг оперативної пам'яті	

При виборі та пріоритизації даних також враховуються поточні вимоги та цілі системи. Наприклад, якщо головною метою є максимізація швидкодії виконання операцій, то дані, необхідні для цих операцій, матимуть вищий пріоритет порівняно з іншими типами даних.

Висновки

Було детально розглянуто концепцію паралельних обчислень та їх роль у розподілених системах. Було визначено основні принципи та типи паралельних обчислень, а також розглянуто їх застосування в різних галузях, таких як наука, фінанси, медицина та інші. Основна увага була приділена розподіленим обчисленням, багатоядерним системам, кластерним і грід-системам, що дозволяють значно підвищити продуктивність та ефективність обробки даних.

Паралельні обчислення відіграють ключову роль у сучасних розподілених системах, дозволяючи розподіляти завдання між різними процесорами або вузлами системи. Це забезпечує швидшу обробку даних, зменшує час виконання завдань та покращує загальну продуктивність системи. Важливо зазначити, що використання паралельних обчислень вимагає врахування принципів синхронізації та управління ресурсами для уникнення конфліктів та забезпечення стабільної роботи системи.

Розподілені обчислення, зокрема, забезпечують можливість виконання складних завдань, таких як моделювання клімату, аналіз геномних даних та обчислення складних фізичних моделей, шляхом розподілу завдань між різними вузлами мережі. Багатоядерні системи дозволяють ефективно використовувати ресурси сучасних процесорів, забезпечуючи високу продуктивність та швидкість обробки даних.

Крім того, було розглянуто важливість опрацювання поведінкових даних у розподілених системах. Поведінкові дані, що включають інформацію про дії та взаємодію користувачів з системою, дозволяють виявляти патерни поведінки, встановлювати тенденції та виявляти проблемні моменти, що потребують вдосконалення. Аналіз таких даних дозволяє підвищити рівень задоволення користувачів, персоналізувати взаємодію з ними та покращити загальну ефективність системи.

Важливо враховувати етичні аспекти збору та використання поведінкових даних, забезпечуючи захист приватності користувачів та конфіденційність даних. Використання високопродуктивних алгоритмів оброблення та аналізу даних, а

також ефективних систем зберігання та управління даними, є ключовими для успішного впровадження та експлуатації розподілених систем.

2 ОГЛЯД ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ДОДАТКУ

2.1.Вибір бібліотек для реалізації проєкту

Реалізація інтерактивного програмного інтерфейсу включає в себе використання таких технологій, як Electron.js, React та Zustand. Кожна з цих технологій має свої унікальні можливості та сприяє покращенню користувацького досвіду в програмному засобі.

React використовується для реалізації фронтенду програмного інтерфейсу. Він надає зручність у роботі зі станом додатка, реактивні компоненти та можливість побудови складних інтерфейсів [40]. React використовує віртуальний DOM для оптимізації оновлення інтерфейсу, що робить його ефективним для розробки масштабованих додатків.

Electron.js використовується для створення нативних десктопних додатків з використанням веб-технологій. Він дозволяє розробникам створювати додатки, які можуть працювати на різних операційних системах без значних змін у коді. Electron.js забезпечує можливість доступу до системних ресурсів, таких як файли, мережеві з'єднання та інші, що робить його відмінним вибором для створення десктопних додатків [39].

Крім того, використання Electron.js дозволяє зберегти однаковий вигляд та функціонал додатка на різних операційних системах, що спрощує розповсюдження та використання додатка користувачами з різних платформ.

Zustand є бібліотекою управління станом, яка дозволяє зручно та ефективно керувати станом додатка в React. Вона базується на контексті та хуках, що робить код більш зрозумілим та простим для розробників [41-43]. Zustand допомагає вирішувати проблеми зі збереженням та оновленням стану, що забезпечує плавну та ефективну роботу інтерфейсу.

Поєднання React, Electron.js та Zustand у створенні десктопного застосунку для відображення статистики комп'ютера та налаштувань відкриває безліч можливостей для розробки зручного та ефективного інтерфейсу.

Одним з ключових аспектів обрання цих технологій була їхня популярність та активна спільнота розробників. Кожна з цих технологій доповнює одна одну, створюючи збалансовану та функціональну систему, яка задовольняє вимоги користувачів та сприяє зручності розробки та підтримки програмного продукту. Це гарантує наявність документації, регулярні оновлення та підтримку, що є важливим для створення стабільного та сучасного програмного інтерфейсу. Обрані технології були обґрунтовані їхньою ефективністю та можливістю спільної інтеграції для досягнення поставленої мети - створення інтерактивного та зручного в управлінні програмного інтерфейсу.

Коли поєднуємо ці технології разом, отримуємо зручний механізм для розробки динамічного та функціонального десктопного застосунку. React забезпечує інтерактивний та зрозумілий інтерфейс, Electron.js дозволяє запускати додаток як нативний на різних платформах, а Zustand забезпечує простий та ефективний механізм управління станом.

Для кращого розуміння, нижче наведено схематичне представлення загальної роботи компонент програми, рисунок 2.1

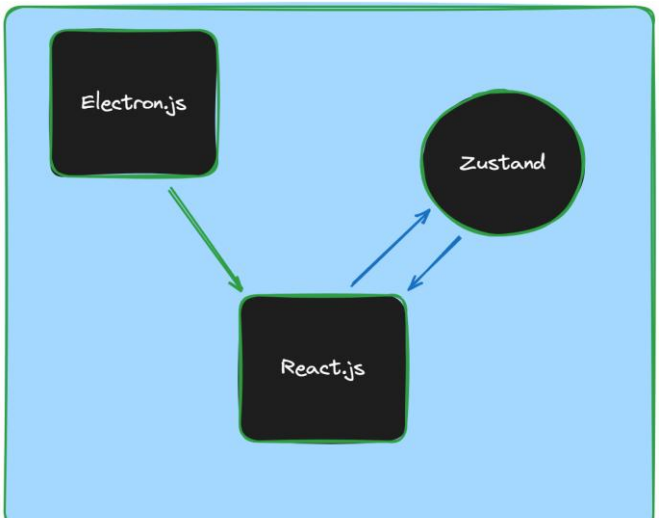


Рисунок 2.1 – Робота компонент клієнтської частини

Тут можна побачити як саме відбувається комунікація цих технічних компонент, можна побачити односторонній зв’язок між Electron, та React, це

зумовлено тим, що бібліотека React, використовується лиш для відображення користувацького інтерфейсу, а сам застосунок лежить в основі Electron [44].

Electron може відображати інтерфейс базуючись на одному лиш файлі index.html, тут вступає в силу наш React.js, який генерує весь інтерфейс в той самий index.html.

Така комбінація є дуже вдалою, адже будувати застосунок, традиційними методами, може забирати чимало часу при розробці застосунку, але з іншої сторони даний підхід вимагає певних знань та навичок для користування React, тому вибір технологій, в таких ситуаціях де потрібно легко та швидко побудувати застосунок, залежить в першу чергу від практичних та теоретичних навичок розробника застосунку.

Також на даному рисунку можна побачити двонаправлений зв’язок між React та Zustand, це обумовлене тим, що “спілкування” React та Zustand, дуже часто є постійним та дані які переходять між React та Zustand, надходять як від сторони User Interface до Business Logic Layer так і навпаки, тому що, між цими двома абстракціями архітектурно встановлений Flux зв’язок, це означає що, дані в інтерфейсі користувача напряму і тільки напряму, залежать від нашого стану застосунку.

2.2.Реалізація інтерактивного програмного інтерфейсу

Розгляд архітектури компонентів, слід почати з основної компоненти застосунку main.ts, яка, фактично, слугує основою для налаштування маршрутизації застосунку. Дана компонента є центральною точкою, звідки йде розгалуження шляхів для всього застосунку.

Нижче схематично показано, логіку компоненти main, рисунок 2.2.

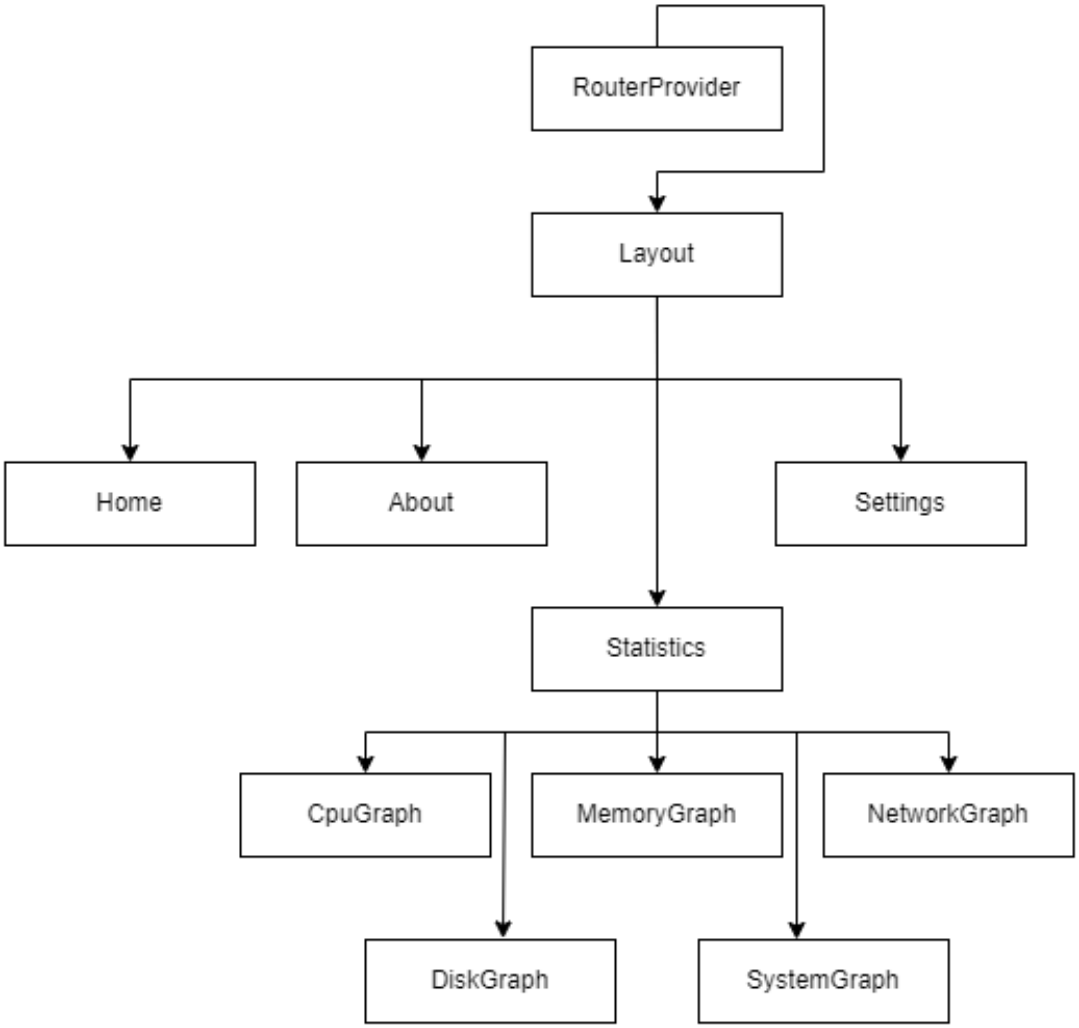


Рисунок 2.2 – Діаграма компоненти main.ts

Оформелення даної компоненти починається з підключення всіх сторонніх бібліотек, та інших засобів для побудування логіки роботи програми. Все починається з виклику коду import somePackage from “шлях до бажаного ресурсу” Дана компонента підключає всі компоненти, які наведено на діаграмі 3.1, тому, що дана компонента займається налаштуванням маршрутизації проекту, і їй необхідно мати доступ до всіх інших сегментів програми. Тут відбувається підключення модулів import Home from "./components/pages/Home/Home.tsx", та інших.

Також варто зазначити, що відбувається підключення головного інструменту для встановлення маршрутизації проєкту це два об’єкти import { RouterProvider, createHashRouter } from "react-router-dom".

Об'єкт RouterProvider є компонентом з бібліотеки react-router-dom, який використовується для інтеграції маршрутизатора в React-додаток. Він забезпечує контекст маршрутизації для всіх компонентів, що знаходяться в його дочірніх елементах, дозволяючи їм отримувати доступ до властивостей та методів маршрутизатора.

Функція createHashRouter створює маршрутизатор, який використовує хеши для навігації. Це означає, що маршрути будуть визначатися частиною URL після символу #. Використання хеш-маршрутизатора є корисним для додатків, які не мають серверної підтримки для обробки маршрутів. Це є особливо важливим використання саме хешованого маршрутизатору, для того щоб правильно зібрати проект, адже якщо використовувати звичайний CreateBrowserRouter, то під час користування застосунком, маршрутизація не буде працювати, клієнт просто не знайде шляхи до ресурсів, саме тому було використоно createHashRouter.

У моєму додатку ReactDOM імпортується з бібліотеки react-dom/client, що дозволяє рендерити React-компоненти в DOM. Основна функція ReactDOM.createRoot створює кореневий вузол для рендерингу React-компонент. Виклик методу render на цьому кореневому вузлі дозволяє React рендерити вказаний компонент всередині елемента DOM, який має ID root:

```
ReactDOM.createRoot(document.getElementById("root")!).render(
   <RouterProvider router={router} />
)
```

Цей код виконує кілька важливих функцій:

– створення Кореневого Вузла: ReactDOM.createRoot створює кореневий вузол для рендерингу React-компонент. Це новий API, представлений у React 18, який забезпечує більш ефективне рендеринг-дерево;

– рендеринг Компоненту RouterProvider: Метод render на кореневому вузлі викликається з компонентом RouterProvider, який отримує об'єкт router як пропс. Це означає, що всі дочірні компоненти RouterProvider матимуть доступ до контексту маршрутизації, що забезпечує навігацію в додатку;

– ініціалізація Маршрутизації: Компонент RouterProvider використовує переданий маршрутизатор (router), створений за допомогою функції createHashRouter. Це дозволяє додатку використовувати маршрутизацію на основі хешів у URL, що забезпечує підтримку навігації навіть без серверної підтримки для маршрутизації.

Використання createHashRouter для створення маршрутизатора, який дозволяє визначати маршрути та їхню ієрархію за допомогою хешів у URL, забезпечує ефективну навігацію в рамках односторінкового додатку (SPA). Основний маршрут визначає кореневий компонент Layout, який служить контейнером для інших сторінок. Всі дочірні маршрути, такі як головна сторінка (Home), сторінка налаштувань (Settings), сторінка інформації (About) та сторінка статистики (Statistics), включені як дочірні елементи компонента Layout.

Особливу увагу приділено створенню вкладених маршрутів (nested routes) у розділі статистики. Використовуючи children у визначенні маршруту для сторінки статистики, я додав кілька підмаршрутів, які відповідають за різні типи графіків: CpuGraph, MemoryGraph, NetworkGraph, DiskGraph, та SystemGraph. Кожен з цих підмаршрутів має свій власний шлях, що дозволяє користувачам переходити між різними графіками, зберігаючи контекст основної сторінки статистики.

Щоб краще зрозуміти архітектуру компонент, потрібно розглянути детальніше вкладені сторінки, та їх підсторінки, тому наступна компонента, яку варто розглянути стане Layout, яка несе в собі логіку, специфічного модуля, який не змінює в собі зовнішнього вигляду, допоки не буде визначено, допоміжна компонента Outlet, яка слугує місцем, куди динамічно вмонтовуються та вимонтовуються всі інші модулі.

Цілком це створює майже моментальний механізм переключення сторінок в застосунку, на відміну від звичайних інтернет сайтів. В тому і є величезна перевага SPA застосунків, яка покращує загальний досвід в користуванні застосунком.

Отже перейдемо до Layout, нижче наведено діаграму, яка показує вміст і логіку Layout модуля, рисунок 2.3.

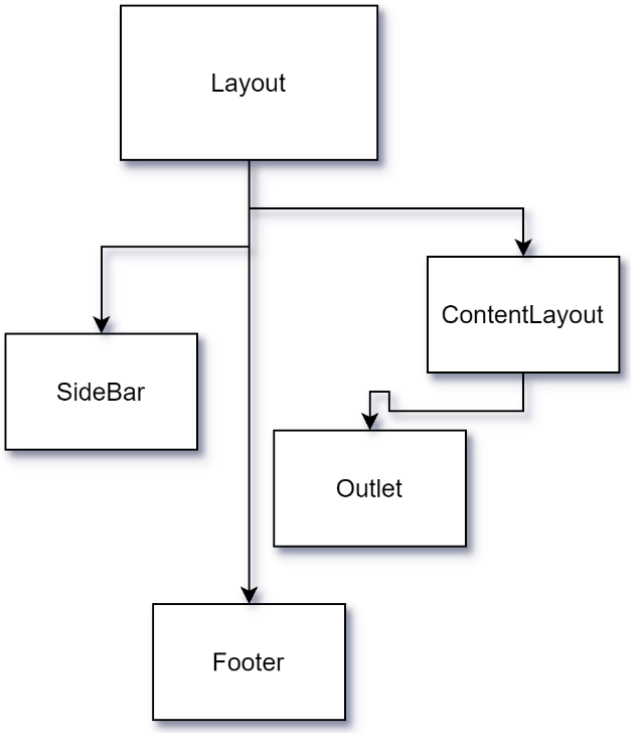


Рисунок 2.3 – Діаграма компоненти Layout із Outlet

Подивившись на компоненту Layout, спочатку може виникнути питання: яким чином такий маленький модуль містить в собі весь застосунок. Відповідь буде дуже проста: компонента Layout використовується та того, щоб під час розробки не задумуватись над деталями розташування, візуальних блоків, які не змінюються.

Після підключення модуля import { Outlet } from "react-router-dom", можна поставити дану компоненту в будь-яке комфортне місце, де розробник хотів би реалізувати перемикання сторінок SPA, дякуючи бібліотеці react-router-dom, реалізація даного підходу стає набагато простішою, аніж, якщо робити маршрутизцію та позиціонування блоків самостійно, і це не далеко всі можливості даної бібліотеки.

Після того як було налаштована маршрутизація, та всі інші роботи, потрібно, щоб були сторінки, які і будуть перемикатися, пропоную спочатку розглянути архітектуру розташування основних сторінок (pages), в директорії

проекту, відповідно і створено папку pages, в якій будуть розміщуватись всі необхідні компоненти.

Для кожної з сторінок, створюється окрема директорія, яка показує, що цей шлях пов’язаний з роботою конкретної компоненти, звідси назва директорії є ідентичною з самим файлом компоненти tsx, та разом з цим файлом, ще, зазвичай, розташований один модульний файл, який відповідає за стилізацію цих компонент.

Коли модуль має в собі вкладені компоненти, то, зазвичай, директорії цих вкладених компонент розташовуються, безпосередньо, в тій самій батьківській директорії компонента, в якій вона розташована, для кращого розуміння, пропоную переглянути діаграму архітектури програмного додатку, рисунок 2.4.

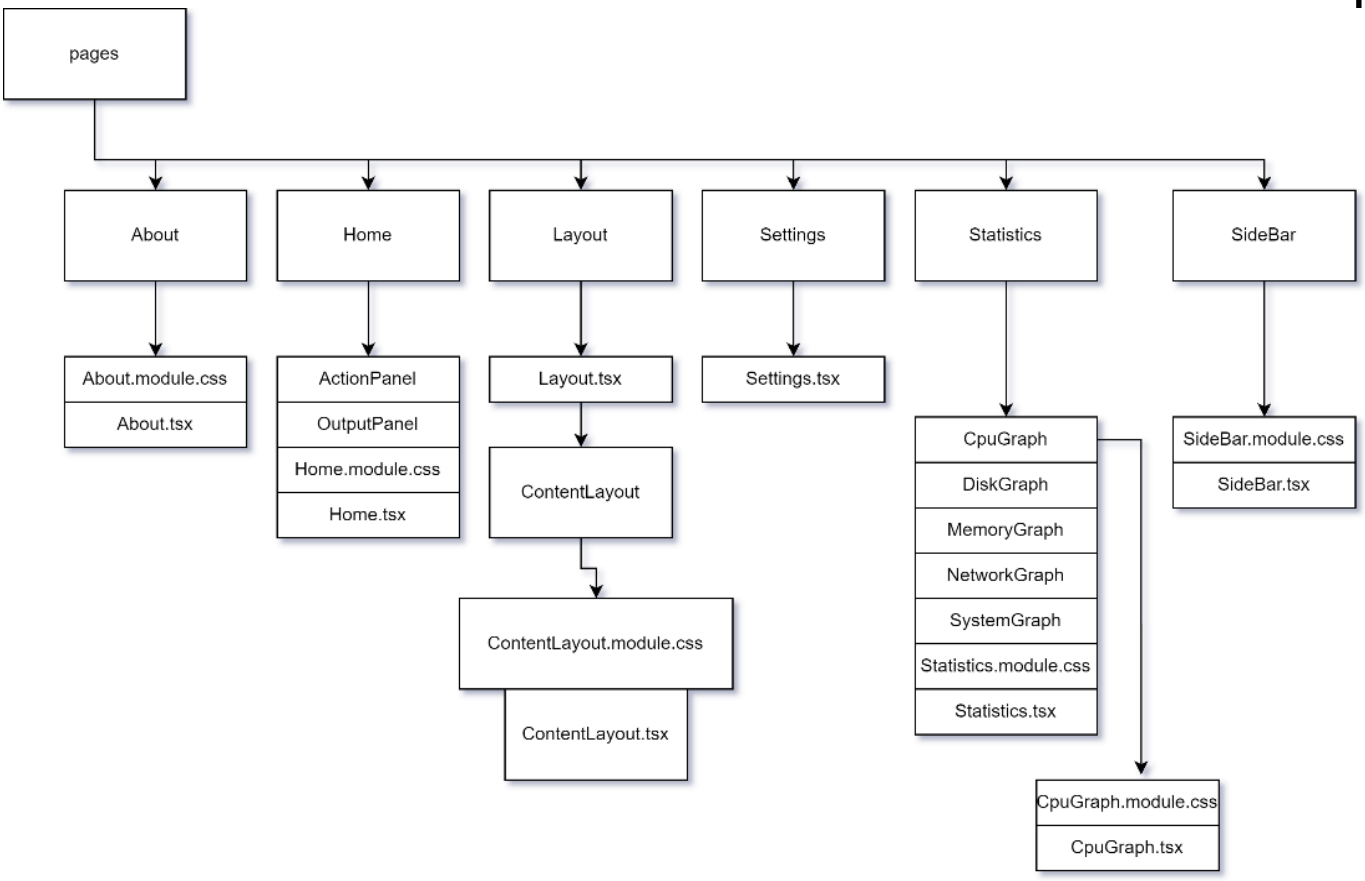


Рисунок 2.4 – Архітектура програмного додатку

Хочу зазначити точніше про використання модульних фалів стилів (module.css файлів). Використання модульних стилів, дає багато переваг над використання звичайних стилів.

Модульні CSS файли дозволяють уникнути глобального простору імен, оскільки кожен стиль обмежується конкретним компонентом, для якого він призначений. Це означає, що стилі, визначені в module.css файлах, не можуть випадково впливати на інші частини програми.

У звичайних CSS файлах всі стилі додаються до глобального простору, що часто призводить до конфліктів, коли різні розробники працюють над різними частинами проекту. Завдяки module.css, кожен компонент має свої власні ізольовані стилі, що значно спрощує підтримку та розширення проекту.

Однією з головних переваг модульних стилів є унікальність імен класів. Коли компонент імпортує модульний CSS файл, кожен клас автоматично отримує унікальне ім’я, що складається з оригінального імені класу та унікального хеша. Це гарантує, що стилі застосовуються тільки до відповідного компонента, забезпечуючи при цьому чистоту та структурованість коду. Я можу називати класи однаково в різних модульних CSS файлах, не турбуючись про конфлікти, що робить розробку більш інтуїтивною і безпечною.

Додатково, модульні CSS файли сприяють повторному використанню стилів. Компоненти, які я створюю, мають свої власні стилі, що не впливають на інші частини програми. Це дозволяє мені з легкістю підтримувати кодову базу, роблячи компоненти більш самодостатніми та незалежними. Завдяки цьому, управління стилями стає більш організованим і ефективним.

Далі розглянемо компоненту Home, яка є стартовою точкою в застосунку, це саме те що буде бачити користувач, вперше, коли запустить застосунок, отже сама компонента, є такою, обгорткою, в якій розташовані дві панелі, які призначені для взаємодії з користувачем, а саме: ActionPanel та OutputPanel. Перший модуль призначений для можливості змінювати кількість ядер, яка буде застосовуватись під час виконання завдань, які отримуються, з серверної частини,

та кнопки: для підключення до віддаленого хосту, відключення від нього, кнопка для збереження змін по ядрам процесора.

Також для комфортного відслідковування, на дану панель було винесено місце для логування роботи програми, тут буде відоражатися інформація по роботі кожного окремого потоку.

Компонента OutputPanel, призначення для відображення стану виконання програми, тобто чи виконується завдання в даний момент, та статус підключення клієнта.

Також на цій панелі, ще добавлено вивід інформаціє про навантаження процесора, це також, в своєму роді, інформує про виконання роботи клієнта, адже під час роботи потоків, навантаження процесору збільшується. Нижче наведено загальний вигляд сторінки Home, рисунок 2.5.

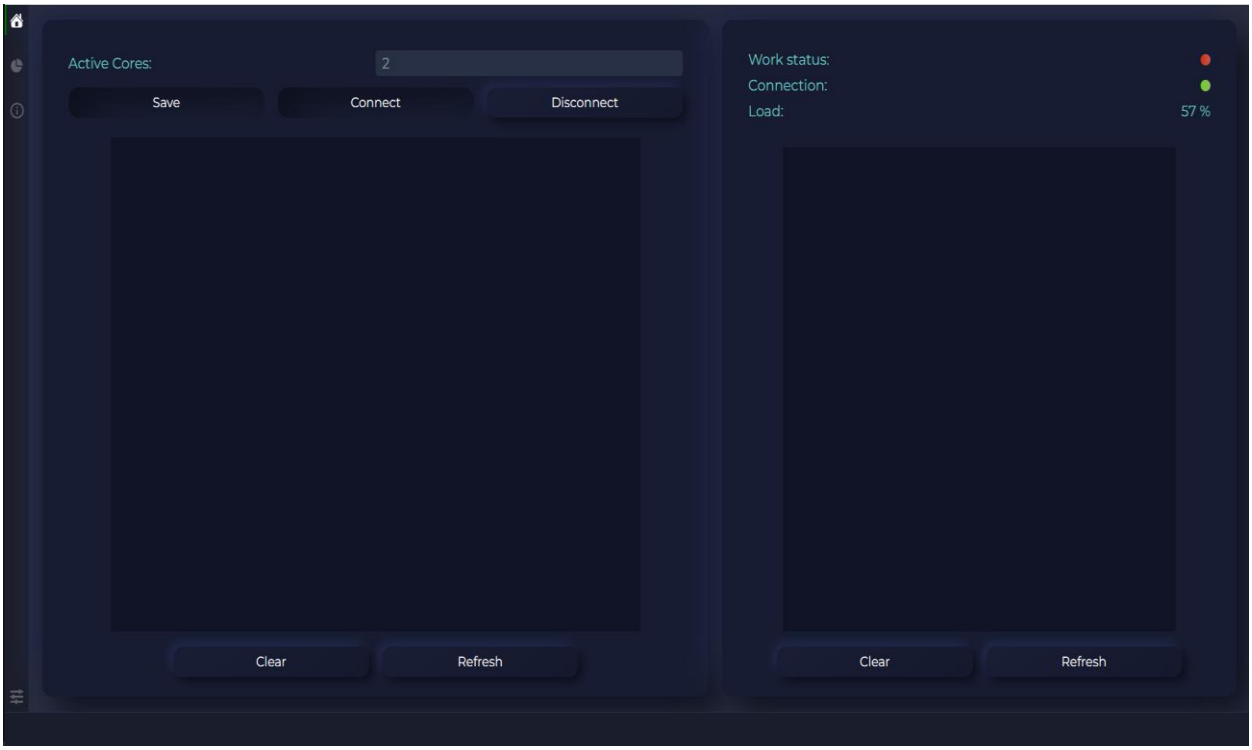


Рисунок 2.5 – Вигляд головної сторінки

Також важливо буде винести діаграми кожних з цих компонент, та розглянути особливості їх реалізації, адже тут виконуються найбільш головні дії по роботі застосунку: реалізація заборони невірного введення даних по потокам,

та натискання кнопкою користувачем, комунікація з основним процесом через ipc, та інше, тому розпочнемо з компоненти ActionPanel, діаграму наведено нижче, рисунок 2.6.

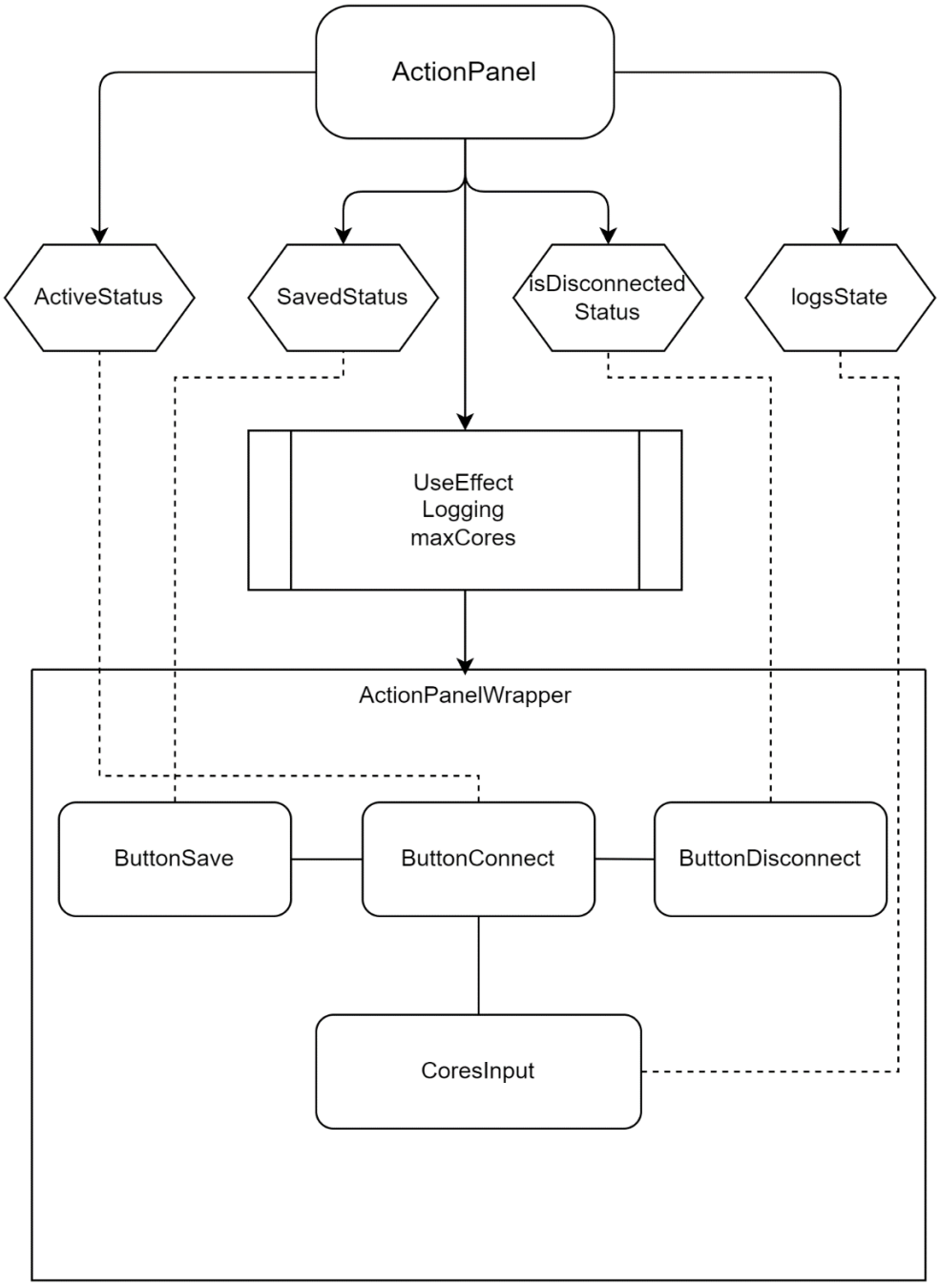


Рисунок 2.6 – Діаграма роботи компоненти ActionPanel

Отже можна побачити відповідно до діаграми, що тут доволі активно використовуються модулі бібліотеки React: UseEffect та UseState, тому зараз перейдемо до детального опису роботи даного модуля.

Користувач може приєднатися до віддаленого хоста, тільки тоді коли він визначив кількість ядер, і натиснув кнопку зберегти.

Приєднатися до серверу та від’єднатися користувач не може, не вказавши ці дані, це зроблено для того, щоб користувач не навантажував сервер, безглуздими з’єднаннями та від ’єднаннями. Також, весь цей функціонал, реалізований завдяки механізмам бібліотеки React: використання станів UseState, де зберігаються стани для контролю поведінки роботи програми, також завдяки цим технологіям, заздалегідь забезпечено випадок, коли користувач взагалі нічого не ввів в поле для визначення ядер, в такому випадку буде виділятися одне ядро, для вирішення задач.

Перейдемо до наступної компоненти OutputPanel. Даний модуль призначений для того, щоб взаємодіяти з користувачем, показуючи статуси роботи програми. Вигляд даної панелі був показаний вище на рисунку 2.5, але діаграма робота даної компоненти наведено нижче рисунок 2.7.

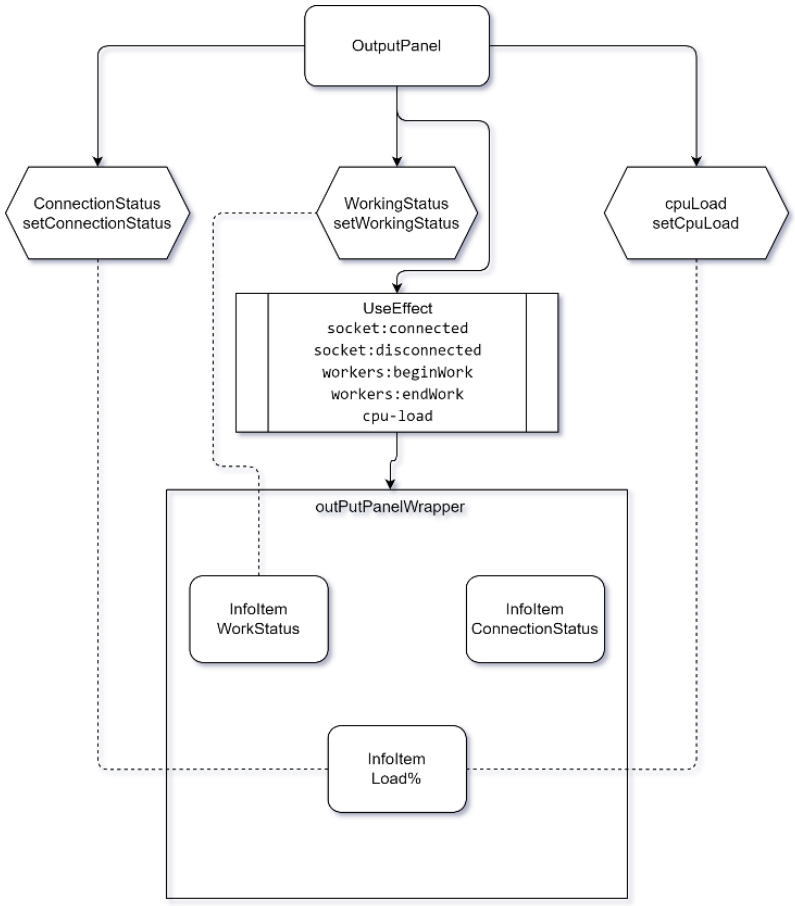


Рисунок 2.7 – Діаграма роботи компоненти OutputPanel

Можна побачити по діаграмі, що тут достатньо часто використовується UseEffect, і так, він в даному модулі – доволі великий. Взагалі UseEffect, це механізм, який дозволяє робити певні дії в залежності від життєвого циклу компоненти.

Наприклад, в даному прикладі, компонента під час свого монтування в віртуальне дерево DOM, підписується на події головного процесу IpcMain, який з своєї сторони буде постачати компоненту потрібною інформацією для відображення в компоненту.

Ще є інші життєві цикли компоненти, наприклад при відмотуванні компоненти, або просто компонента буде робити ререндеринг при зміні асоціативного масиву deps, в який можна помістити дані від яких буде залежати ререндеринг компоненти.

Перейдемо до наступної сторінки About.tsx. Даний модуль слугує для відображення для користувача даних про його комп'ютер, це виключно інформативний модуль, він не несе в собі основну логіку програми, але є приємним додатком для загального досвіду в користуванні програмою. Спочатку пропоную розглянути вигляд даної компоненти, рисунок 2.8.

Рисунок 2.8 – Вигляд компоненти About

Отже можна побачити, що тут немає якоїсь складної логіки, чи функціоналу, але реалізація даного модуля цікавіша, тому пропоную одразу перейти до діаграми роботи компоненти About, рисунок 2.9.

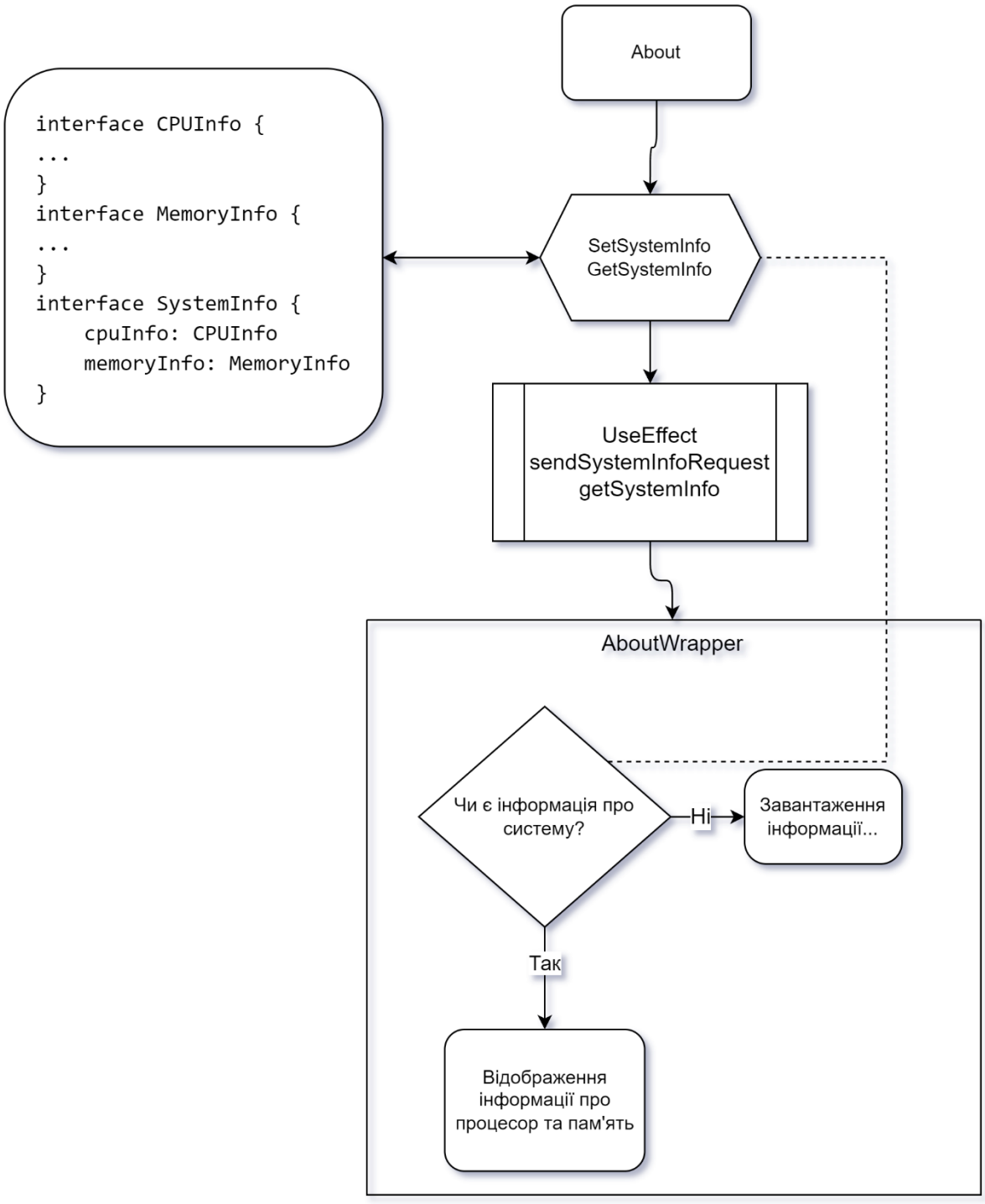


Рисунок 2.9 – Діаграма компоненти About

У About.tsx я використовую інтерфейси для визначення структури даних, які передаються між основним процесом Electron і рендерним процесом. Це забезпечує чітке визначення типів даних, що полегшує їх використання та обробку. Зокрема, я визначив три інтерфейси:

– CPUInfo: містить інформацію про процесор, включаючи виробника, бренд, швидкість і кількість ядер;

– MemoryInfo: містить інформацію про оперативну пам'ять, включаючи загальну кількість, вільну та використану пам'ять;

– SystemInfo: об'єднує в собі об'єкти CPUInfo та MemoryInfo, що представляє собою повну інформацію про систему.

Компонента About використовує стан systemInfo для зберігання системної інформації, яку вона отримує через ipcRenderer. Використання хука useState дозволяє зберігати та оновлювати цей стан. Спочатку systemInfo має значення null, що вказує на те, що дані ще не завантажені.

При завантаженні компоненти я використовую хук useEffect для налаштування обробки системної інформації. Я реєструю обробник подій для прослуховування повідомлень від основного процесу Electron, використовуючи ipcRenderer. Коли основний процес надсилає повідомлення з системною інформацією, цей обробник оновлює стан systemInfo, зберігаючи отримані дані.

Компонента також надсилає запит на отримання системної інформації, коли вона завантажується. Це робиться через виклик window.ipcRenderer.send("get-system-info"), що дозволяє основному процесу Electron надсилати актуальні дані про систему. При розмонтуванні компоненти обробник подій видаляється, щоб уникнути потенційних витоків пам'яті.

У рендер-методі компонента перевіряє, чи systemInfo не є null. Якщо дані вже завантажені, компонента відображає їх у вигляді розділів з інформацією про процесор та пам'ять. Якщо дані ще завантажуються, відображається повідомлення "Loading system information...".

Таким чином, компонента About забезпечує зручний і наочний спосіб перегляду ключових параметрів системи, інтегруючи різні технології для досягнення високої продуктивності та зручності використання. Вона демонструє, як ефективно використовувати React для управління станом і відображення даних, а також як налаштувати комунікацію між основним і рендерним процесами в Electron для отримання актуальної інформації про систему.

Далі розглянемо компоненту, яка також потрібно лише для відображення корисної інформації, але інформація тут подана в вигляді графіків, тобто це окрема сторінка для моніторингу стани системи у вигляді графіків. Нижче буде наведено вигляд даної сторінки, але самі графіки будуть детальніше розглядуватись в наступних пунктах, дивіться рисунок 2.10.

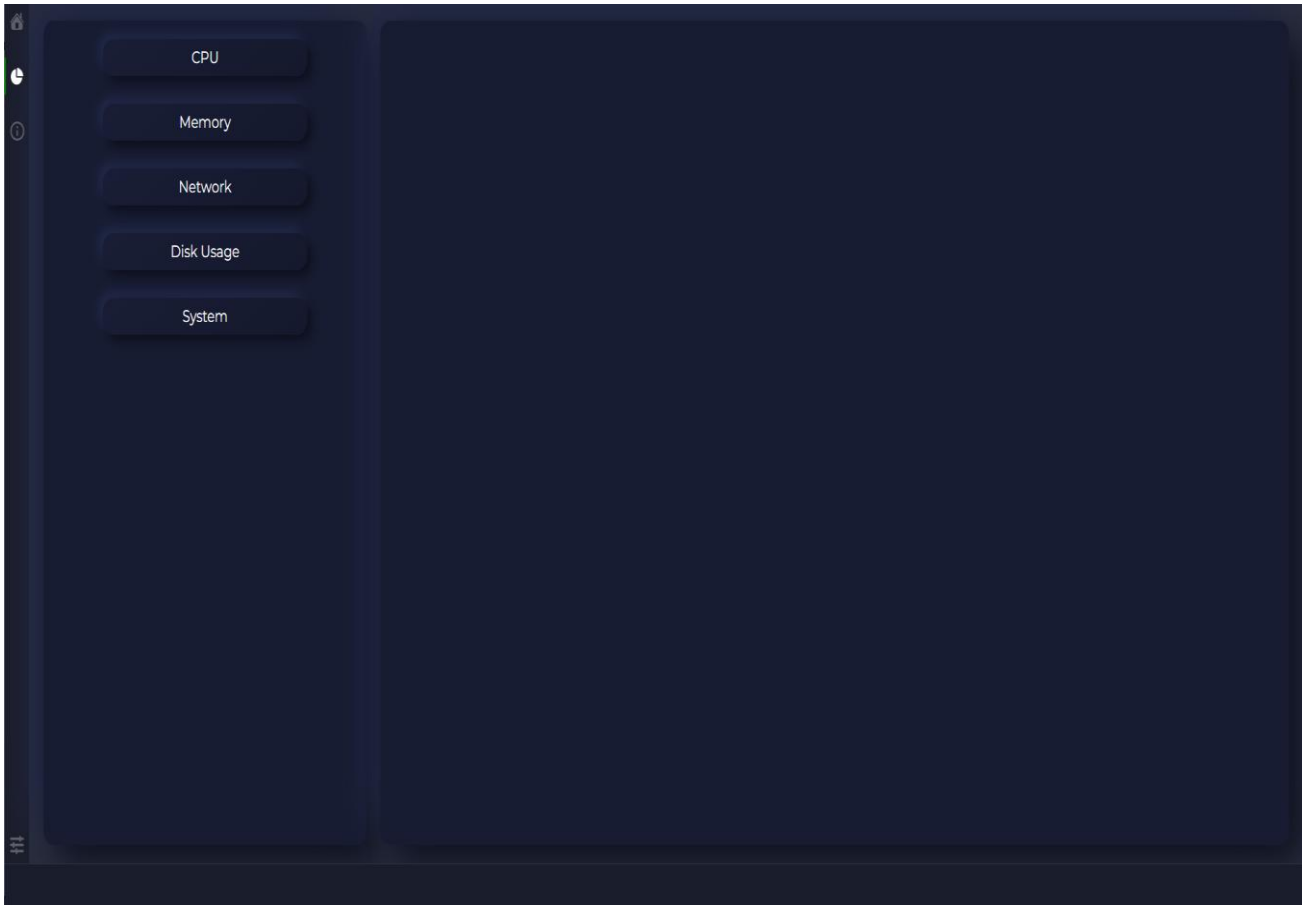


Рисунок 2.10 – Вигляд компоненти Statistics

Але, все ж таки, більш важливіше буде розглянути саму логіки побудови даної сторінки, адже тут буде пов’язано багато різних модулів, діаграма нижче буде показувати як відбувається процес вкладеної маршрутизації для цього модуля, сама комунікація та загальний алгоритм показано на діаграмі нижче, рисунок 2.11.

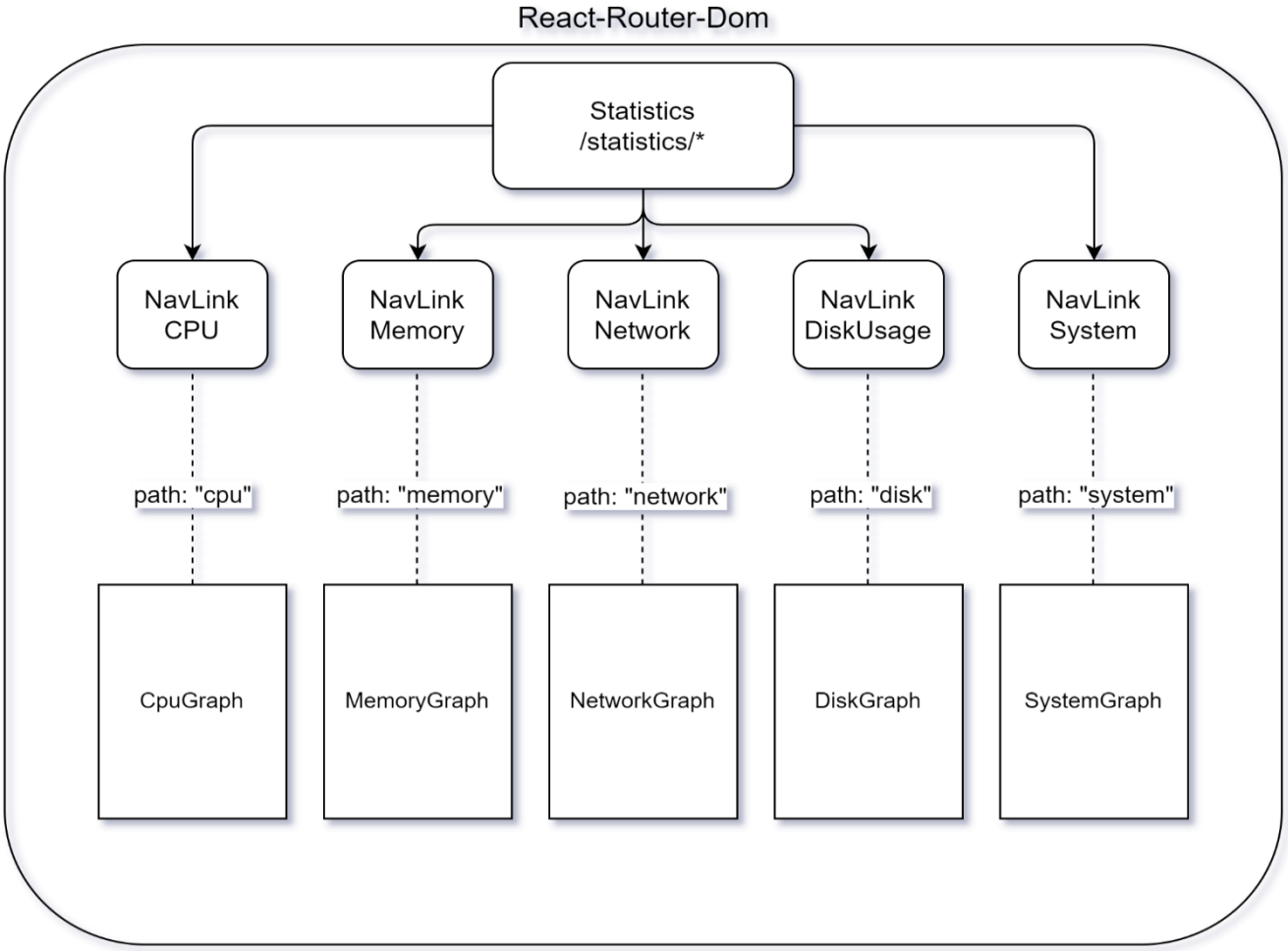


Рисунок 2.11 – Вкладена маршрутизація компоненти Statistics

Отже, діаграма описує як саме відбувається вкладена маршрутизація за шляхами “/statistics/*”, звідси знак “*”, вказує на те, що після шляху statistics, можуть бути і інші маршрути, відповідно до діаграми.

2.3.Огранізація обміну інформації по мережі

Socket.io є надзвичайно потужним інструментом для організації зв'язку та обміну даними між клієнтом і сервером у розподілених системах. Його основна особливість полягає у використанні WebSockets, що дозволяє створювати надійне та ефективне двонаправлене з'єднання. У контексті нашого додатку, де важлива миттєва передача статусів та інших даних між клієнтом і сервером, використання Socket.io стає незамінним [45].

При обранні WebSockets через Socket.io, ми отримуємо значні переваги порівняно зі звичайними Sockets [46]. Одна з головних переваг - це здатність працювати через проксі-сервери та мережеві екрани, що дозволяють уникнути багатьох обмежень, що звичайно постають перед додатками, які використовують звичайні Sockets. Це робить наш додаток більш гнучким і доступним для користувачів.

Окрім цього, Socket.io надає можливість обробляти різні події та повідомлення, що дозволяє створювати складні механізми взаємодії між клієнтом і сервером. Наприклад, ми можемо використовувати події для оновлення даних, підтвердження отримання повідомлень чи реагування на зміни в стані системи.

Також варто зазначити, що Socket.io забезпечує безпеку передачі даних шляхом використання протоколу HTTPS та можливості шифрування [47]. Це дозволяє нам передавати конфіденційну інформацію, таку як дані користувача, з високим рівнем захисту.

Більше того, використання Socket.io гарантує надійність та безпеку передачі даних. Вбудована підтримка шифрування забезпечує захист конфіденційної інформації від несанкціонованого доступу. Це особливо важливо для додатків, де дуже важливо зберігати безпеку даних клієнтів.

Також варто відзначити, що Socket.io забезпечує постійне з'єднання між клієнтом і сервером.

Це означає, що ми можемо миттєво реагувати на події та зміни стану системи без затримок, що робить наш додаток більш керованим та ефективним для користувачів [48].

Узагальнюючи, Socket.io є необхідним інструментом для нашого додатку, який забезпечує нам швидку, надійну та безпечну передачу даних між клієнтом та сервером. Його функціональність і можливості допомагають нам створювати інтерактивний інтерфейс, який задовольняє потреби наших користувачів та забезпечує їм зручну та ефективну роботу з додатком.

Для більш вірного розуміння, процесу предачі даних, розглянемо рисунок 2.12, наведений нижче.

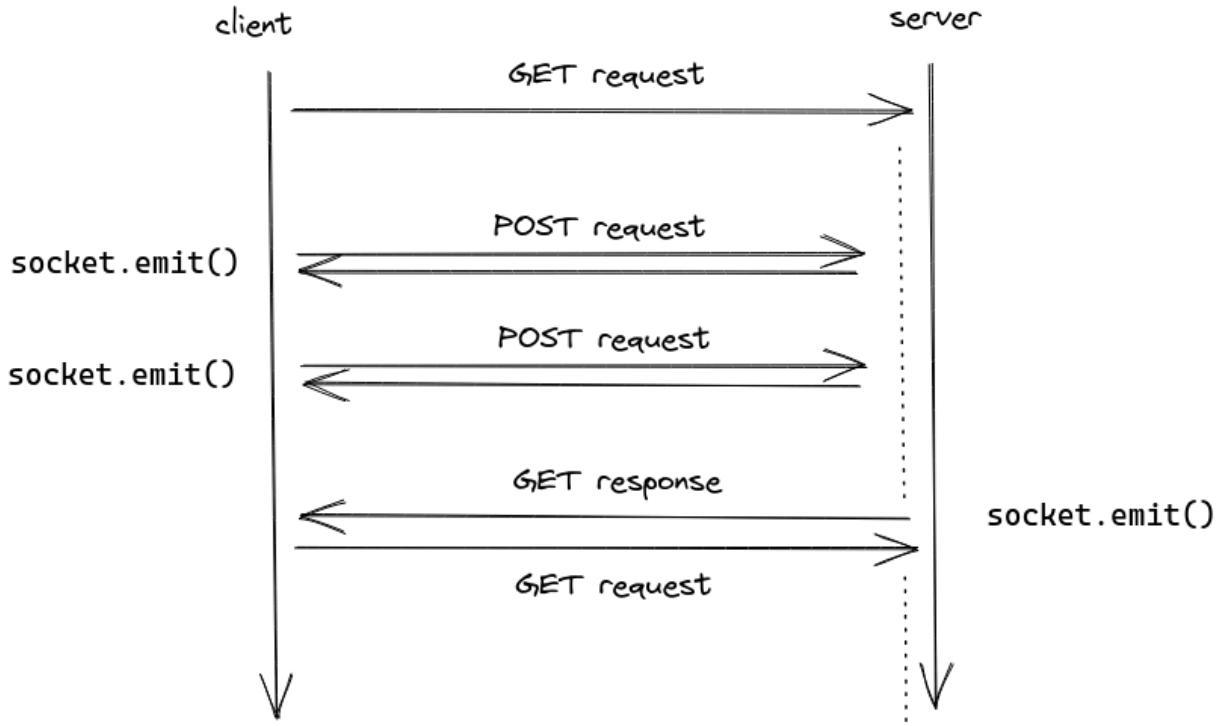


Рисунок 2.12– Комунікація між клієнтом та сервером

Socket.io може використовувати TCP та UDP як основні транспортні протоколи, але в основному використовує TCP.

Протокол TCP (Transmission Control Protocol) забезпечує надійне з'єднання між двома кінцевими точками в мережі. Це досягається шляхом встановлення з'єднання через процес "триетапного рукостискання", що включає обмін спеціальними повідомленнями для встановлення зв'язку перед передачею даних. TCP гарантує, що всі пакети даних будуть доставлені в правильному порядку та без втрат, завдяки використанню механізмів контролю помилок і повторної передачі втрачених пакетів. Розглянемо схему, роботи протоколу TPC, рисунок 2.13.

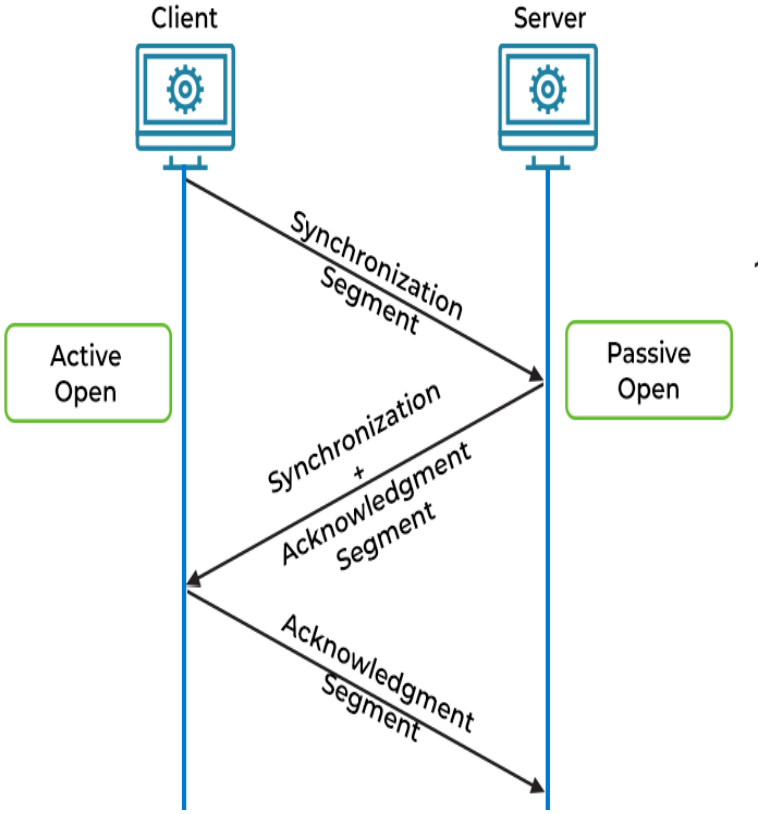


Рисунок 2.13 – Робота протоколу TCP

Процес починається з того, що клієнт ініціює з'єднання, відправляючи серверу сегмент синхронізації (SYN). Цей сегмент сигналізує про намір клієнта встановити з'єднання та включає початковий номер послідовності, який буде використовуватись для синхронізації пакунків даних.

Після отримання SYN-сегмента сервер відповідає клієнту сегментом синхронізації та підтвердження (SYN + ACK).

Цей сегмент виконує подвійну функцію: сервер відправляє свій власний SYN-сегмент, який містить початковий номер послідовності сервера, і одночасно підтверджує отримання SYN-сегмента від клієнта за допомогою ACK-сегмента. Це дозволяє обом сторонам синхронізувати свої початкові номери послідовності та підтвердити готовність до встановлення з'єднання.

Клієнт, отримавши SYN + ACK від сервера, відповідає сегментом підтвердження (ACK). Цей заключний сегмент підтверджує отримання SYN-сегмента від сервера та завершує процес триетапного рукостискання. Після цього з'єднання вважається встановленим, і обидві сторони можуть починати передачу даних. Цей процес забезпечує надійне встановлення з'єднання, підтверджуючи, що обидві сторони готові до обміну даними, і синхронізуючи номери послідовностей для коректної передачі та отримання інформації.

У моєму проекті мережевий обмін даними між клієнтом та сервером здійснюється через кілька ключових точок комунікації, що забезпечують стабільне та ефективне з'єднання. Використовуючи технологію Socket.io, ми встановлюємо надійне двонаправлене з'єднання між клієнтом і сервером, що дозволяє нам обмінюватися різними видами даних у реальному часі.

Пропоную розпочати перегляд, з точки підключення:

let socket: any = undefined.

На початку програми створюється змінна сокету, це потрібно для того щоб потім, під час нового з’єднання не створювати новий сокет, тому, що це є достатньо витратна операція. Під час нового з’єднання, це коли клієнт наприклад вже виконує якесь завдання, і коли воно виконалось, він має змогу змінити кількість ядер для роботи, тому він натискає кнопку Disconnect, тим самим він робить відключення наявного сокету, а не видалення, і тепер, коли вже користувач хоче підключитися, він натискає Connect, потім через Ipc, йде відправка події:

window.ipcRenderer.send("requestConnection"),

яка потрапляє на обробник цієї події:

ipcMain.on("requestConnection", () => { if (socket && socket.connected) { console.log("Already connected"); return; }}).

Тут ми ще додатково перевіряємо, чи існує об’єкт сокету, і чи якщо існує, то чи він є під’єднаним, якщо так, то виводиться невелике повідомлення, про те, що з’єдання вже встановлено, та відбувається вихід з функції, і це є логічним, адже без цього, нам довелось би створювати кожного разу новий сокет. Коли підключення встановлено, клієнт відправляє повідомлення до вікна рендеру, інформуючи про успішне з'єднання.

Процес отримання завдань можна описати таким чином: Після встановлення з'єднання клієнт отримує завдання від сервера через подію request:task:

```
socket.on("request:task", (encPayload: any) => {
    console.log(encPayload);
    let payload: any = JSON.parse(
      decrypt(encPayload, process.env.SECRET_KEY)
    );
    let A = payload.A;
    let b = payload.b;
    win?.webContents.send("workers:beginWork");
    let dataForWorkers = [];
    for (let i = 0; i < A.length; i++) {
      dataForWorkers.push({ A: A[i], B: b[i] });
    }
```

```
    workerPool?.addTask(dataForWorkers, payload.maxIterations);
  });
```

Дані завдання шифруються, і клієнт їх розшифровує, після чого починає виконання завдання, передаючи дані до пулу робітників. Що таке пул робітників буде наведено в наступних пунктах. Далі перейдемо до наступної точки з’єднання: відправка даних про систему.

Для моніторингу та відправки даних про стан системи, клієнт використовує наступний код:

```
  socket.on("system:cpu-usage", (payload: any, callback: any) => {
    si.get(payload.usageConfig).then((data) => {
      callback(data.currentLoad);
    });
  });
  socket.on("system:mem-usage", (payload: any, callback: any) => {
    si.get(payload.usageConfig).then((data) => {
      callback(data.currentLoad);
    });
  });
```

Тут одразу наведе дві точки для обробки подій , і клієнтська частина віддає дані про процесор та про використання оперативної пам’яті відповідно, використовуючи модуль “systeminformation”, очевидно, цей модуль імпортується вище викликом import si from "systeminformation". Далі розглянемо процес відключення від серверної частини:

```
ipcMain.on("requestDisconnect", () => {
  if (socket) {
    socket.disconnect();
  }
```

});

Ця функція забезпечує можливість розірвати з'єднання за запитом користувача, завершивши всі активні процеси та звільнивши ресурси.

Таким чином, мережевий обмін даними між клієнтом і сервером у моєму проекті реалізовано через низку точок підключення, обробки завдань та моніторингу системи, що забезпечує стабільну і ефективну роботу програми. Для простоти розуміння, також рекомендую переглянути діаграму комунікації між клієнтом та сервером, рисунок 2.14.

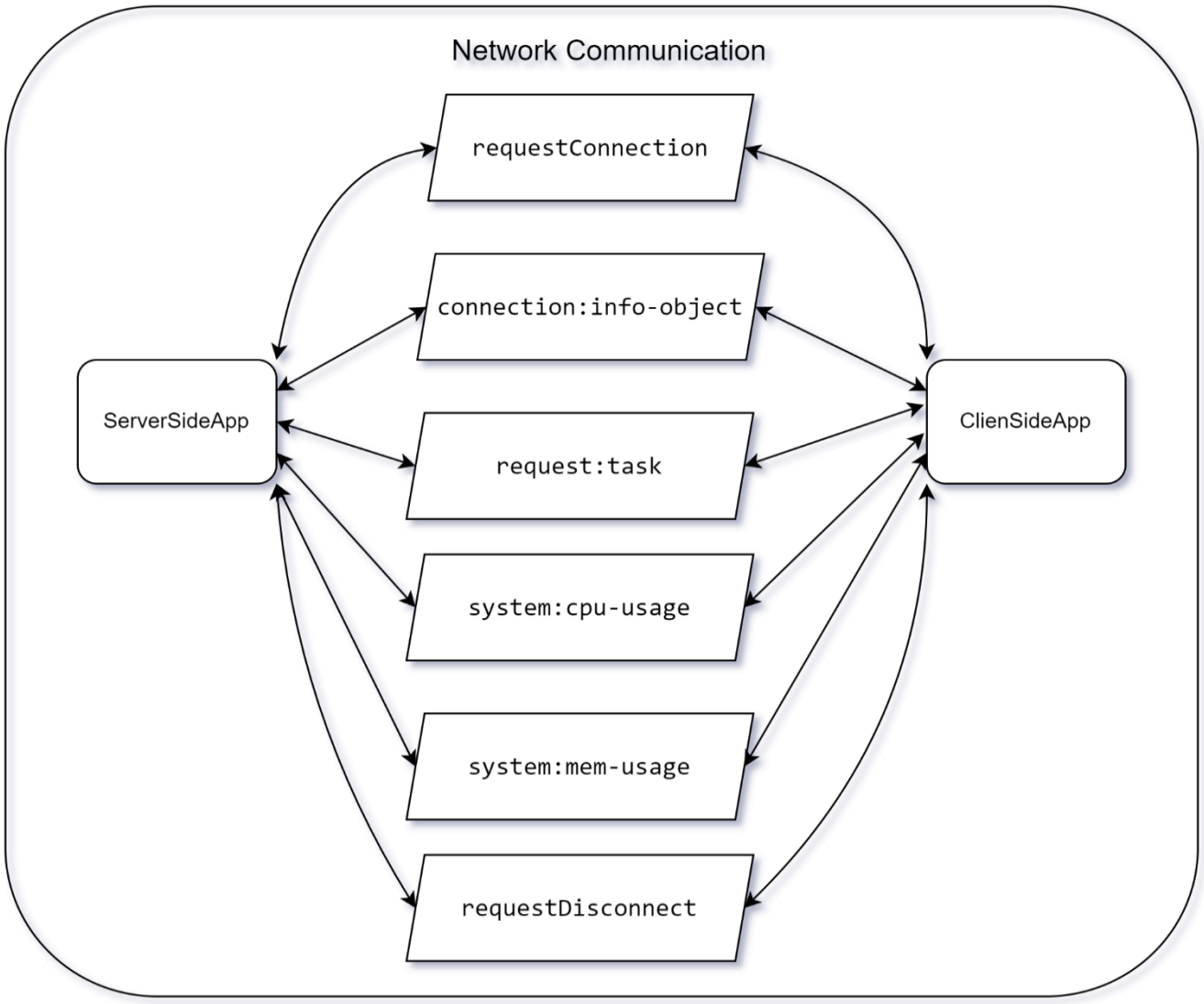


Рисунок 2.14 – Діаграма комунікації між клієнтом та сервером

Кінцеві точки на цій діаграмі позначені паралелограмами, з шляхами в середині них.

В цілому, такого налаштування достатньо для нормального функціонування системи, але це не означає що клієнтська та серверна частина обмежена цими кінцевими точками, при бажанні сюди можна просто додати новий функціонал.

Висновки

У другому розділі роботи було розглянуто та проаналізовано різні програмні засоби для реалізації системи, включаючи технології, бібліотеки та методи, необхідні для створення ефективного та надійного програмного забезпечення. Основна увага була зосереджена на інтерактивному програмному інтерфейсі, забезпеченні зв'язку та організації обміну даними, а також на забезпеченні захисту з'єднання розподіленої системи.

Перш за все, було детально описано реалізацію інтерактивного програмного інтерфейсу за допомогою таких технологій, як Electron.js, React та Zustand. Ці технології забезпечують створення нативних десктопних додатків з використанням веб-технологій, надаючи можливість розробляти додатки, які можуть працювати на різних операційних системах без значних змін у коді. Використання React для фронтенд-розробки забезпечує зручність у роботі зі станом додатка та ефективність оновлення інтерфейсу завдяки використанню віртуального DOM. Zustand, у свою чергу, дозволяє ефективно керувати станом додатка в React, забезпечуючи простий та зрозумілий механізм управління даними.

Також було розглянуто питання забезпечення зв'язку та організації обміну даними між клієнтом і сервером за допомогою технології Socket.io. Використання Socket.io також гарантує надійність та безпеку передачі даних, забезпечуючи шифрування та підтримку протоколу HTTPS.

Таким чином, у другому розділі було обґрунтовано вибір та використання сучасних технологій і методів для реалізації інтерактивного програмного інтерфейсу, забезпечення зв'язку та захисту даних у розподіленій системі.

3 РОЗРОБКА ОБЧИСЛЮВАЛЬНОГО ПРОГРАМНОГО ЗАСОБУ

3.1.Організація розподілу завдань між потоками

У моєму проекті важливим аспектом є ефективне розподілення завдань між потоками для досягнення максимальної продуктивності. Для цього я використовую worker_threads, що дозволяє створювати та управляти потоками у Node.js. Ця технологія дозволяє запускати паралельні обчислення, що є критичним для роботи з великими обсягами даних і складними обчислювальними завданнями.

В основі розподілу завдань між потоками лежить поняття пулу робітників (WorkerPool). Пул робітників дозволяє створювати та управляти кількома потоками, які можуть виконувати завдання незалежно один від одного. Це забезпечує ефективне використання ресурсів системи, зокрема процесорних ядер, і зменшує час виконання завдань.

Однією з ключових функцій мого додатку є можливість користувача встановлювати кількість ядер процесора, які будуть використовуватися для виконання завдань. Це робиться через графічний інтерфейс користувача, що дозволяє налаштувати параметри продуктивності відповідно до потреб і можливостей системи. Наприклад, користувач може зменшити або збільшити кількість використовуваних ядер, залежно від інших завдань, що виконуються на комп'ютері.

При створенні пулу робітників я ініціалізую необхідну кількість потоків, виходячи з встановленого значення. Кожен потік отримує свою частину завдання і починає його виконувати. Це дозволяє обробляти великі масиви даних паралельно, що значно підвищує ефективність роботи додатку.

Коли завдання надходить до системи, воно розподіляється між доступними потоками в пулі. Кожен потік отримує окрему частину завдання, обробляє її і повертає результат. Після завершення роботи всіх потоків результати

об'єднуються і передаються назад до основного потоку для подальшої обробки або відображення.

Використання worker_threads дозволяє моєму додатку працювати більш ефективно, оскільки завдання виконуються паралельно, а не послідовно. Детальний алгоритм реалізації WorkerPool, розглянемо нижче, рисунок 3.12.

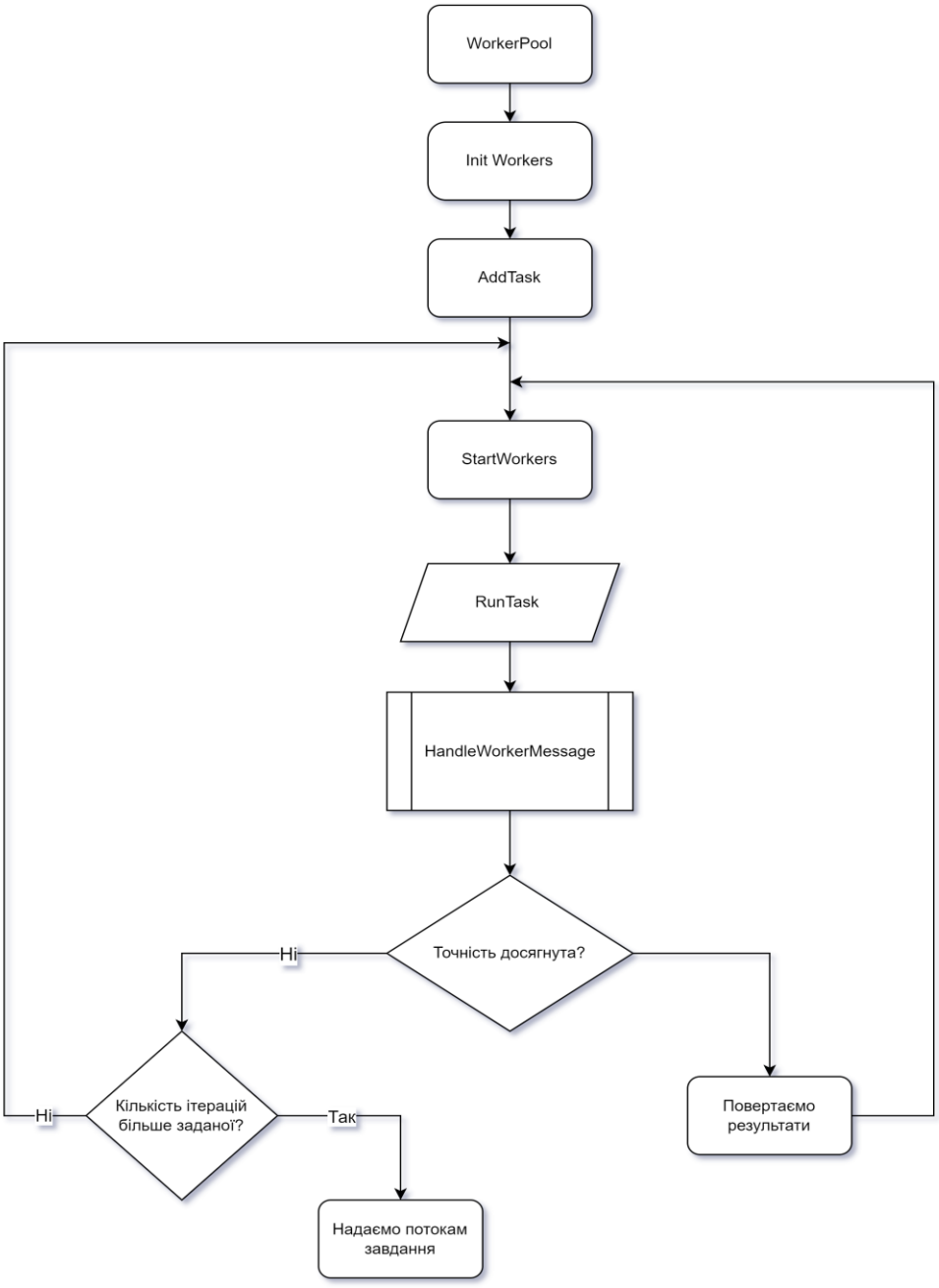


Рисунок 3.12 – Діаграма роботи класу WorkerPool

У контексті даної роботи "воркер" означає worker-thread або робочий потік у Node.js, який виконує паралельні обчислення. Воркер є окремим потоком виконання, створеним за допомогою модуля worker_threads, що дозволяє виконувати складні обчислювальні завдання незалежно від головного потоку. Це забезпечує ефективне використання багатоядерних процесорів, дозволяючи розподіляти завдання між кількома потоками і тим самим значно підвищуючи продуктивність додатку.

Цей клас забезпечує управління пулом воркерів, які паралельно виконують завдання. Діаграма на рисунку ілюструє основні етапи роботи WorkerPool.

Клас WorkerPool спочатку ініціалізується з використанням необхідних параметрів, таких як шлях до скрипта воркера, максимальна кількість воркерів, а також колбеки для логування та завершення роботи. Після ініціалізації, метод initWorkers створює та запускає воркерів.

При створенні воркера викликається метод createWorker, який налаштовує обробку повідомлень від воркера. Основний метод обробки повідомлень - це handleWorkerMessage. Цей метод отримує результати від воркера, оновлює стан завдання та вирішує, чи потрібно продовжувати роботу, або завдання вже виконано з необхідною точністю.

Коли нове завдання додається за допомогою методу addTask, воно розбивається на підзадачі та додається до черги завдань. Потім метод startWorkers розподіляє ці підзадачі між доступними воркерами. Якщо воркер вільний і в черзі є завдання, метод runTask запускає це завдання на воркері.

В процесі виконання завдання воркер надсилає результати назад до WorkerPool. Метод handleWorkerMessage обробляє ці результати та перевіряє, чи досягнута необхідна точність. Якщо точність досягнута, завдання вважається завершеним, і результати повертаються. Якщо точність не досягнута, воркери отримують нові підзадачі для подальшої обробки.

У разі завершення роботи, всі воркери зупиняються за допомогою методу terminateAll, який завершує всі активні воркери та очищує чергу завдань.

Наступним до перегляду, стає алгоритм виконання, безпосередньо, самої системи лінійних алгебраційних рівнянь, методом якобі.

Це є та частина роботи, яка розпаралелюється між потоками, кожен потік отримує свою частину роботи. Якщо находить рівняння, з трьома невідомими, то система розбиває це рівняння на кожний рядок для окремого потоку та додатково відповідне значення матриці вільних членів.

Розглянемо, поетапно роботу алгоритму для одного воркера. Перший етап ініціалізцації, воркери використовують модуль worker_threads для паралельного виконання завдань. Основний файл воркера починається з імпорту необхідного модуля:

```
const { parentPort } = require("node:worker_threads");
```

Цей модуль дозволяє воркеру отримувати та надсилати повідомлення головному потоку, забезпечуючи двосторонню комунікацію.

Воркери використовують метод Якобі для розв'язання систем лінійних рівнянь. Функція jacobiMethod реалізує цей метод:

```
function jacobiMethod(A, B, equationIndex, equationCount, newResults, taskId)
{
    if (newResults.length == 0) {
        newResults = new Array(equationCount).fill(0);
    }
    let sum = 0;
    for (let j = 0; j < equationCount; j++) {
        if (equationIndex !== j) {
            sum += A[j] * newResults[j];
        }
    }
    const x_new = (B - sum) / A[equationIndex];
```

```
    const toleranceDif = Math.abs(x_new - newResults[equationIndex]);
    return {
      index: equationIndex,
      x_new,
      toleranceDif,
      taskId
    };
  }
```

Ця функція приймає матрицю коефіцієнтів A, вектор вільних членів B, індекс рівняння equationIndex, кількість рівнянь equationCount, поточні результати newResults та ідентифікатор завдання taskId. Вона обчислює нове значення x_new для поточного рівняння, порівнює його з попереднім значенням для визначення різниці toleranceDif і повертає ці результати.

Воркери отримують завдання від головного потоку через повідомлення. Для цього використовується обробник подій:

```
  parentPort.on("message", (message) => {
    if (message === "terminate") {
      process.exit(0);
    } else {
      parentPort.postMessage(
        jacobiMethod(
          message.A,
          message.B,
          message.equationIndex,
          message.equationCount,
          message.newResults,
          message.id
```

```
            )
        );
    }
});
```

Для більш наявного представлення, потрібно ознайомитись з діаграмою роботи даного алгоритму, рисунок 3.13.

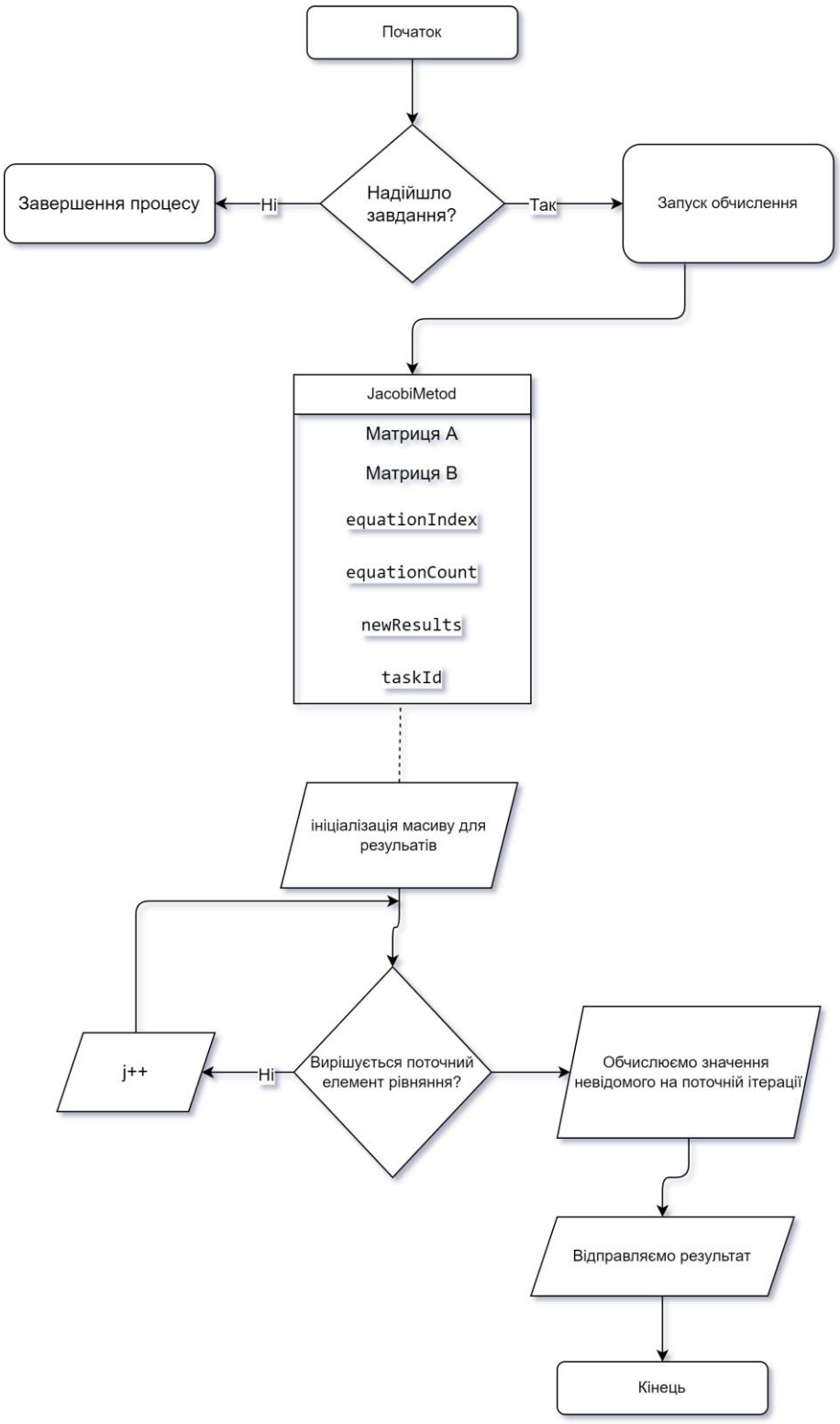


Рисунок 3.13 – Діаграма роботи алгоритму для вокрера

Цей обробник перевіряє тип повідомлення. Якщо це команда на завершення (terminate), воркер завершує свою роботу. Якщо це завдання на обчислення,

воркер викликає метод Якобі з параметрами, отриманими в повідомленні, і надсилає результати назад головному потоку.

Таким чином, алгоритм роботи воркера забезпечує ефективне паралельне обчислення завдань, використовуючи метод Якобі для розв'язання систем лінійних рівнянь. Це дозволяє досягти високої продуктивності та швидкості виконання завдань у розподіленій системі.

3.2. Аналіз продуктивності програмного засобу та результати його функціонування у системі

Результати виконання систем лінійних алгебраїчних рівнянь (СЛАР) були відображені на головній сторінці проекту. На цій сторінці знаходилася інформація про кількість витраченого часу на один запит з багатьма системами СЛАР. Кожен запит включав у себе обчислення декількох систем рівнянь одночасно, що дозволило оцінити продуктивність системи в умовах багатозадачності.

Було також імплементовано паралельне обчислення кількості виконаних завдань за 0.2 секунди. Це дало змогу в режимі реального часу оцінювати продуктивність обчислювальної системи та її здатність до обробки великих обсягів даних. Дані про кількість виконаних завдань зберігалися та відображалися на головній сторінці для наочності.

Варто зазначити, що виміри можуть бути неточними на різних системах. Це обумовлено багатьма факторами, які впливають на точність вимірювань. Наприклад, швидкість запису або читання даних для різних носіїв пам'яті може значно відрізнятися, що впливає на загальний час виконання завдань. Крім того, швидкість інтернет-з'єднання також може впливати на продуктивність, особливо в умовах інтенсивного обміну даними між клієнтом і сервером.

Також слід враховувати, що продуктивність процесора, оперативної пам'яті та інші апаратні характеристики можуть змінюватися в залежності від системи, що використовувалася для тестування. Таким чином, результати можуть

варіюватися, і це потрібно враховувати при оцінці загальної ефективності системи.

Нехай, серверна частина відправляє на клієнтську частину 50000 систем СЛАР, тоді результати вимірювання такої кількості систем при 2-х активних ядер буде такий, рисунок 3.14.

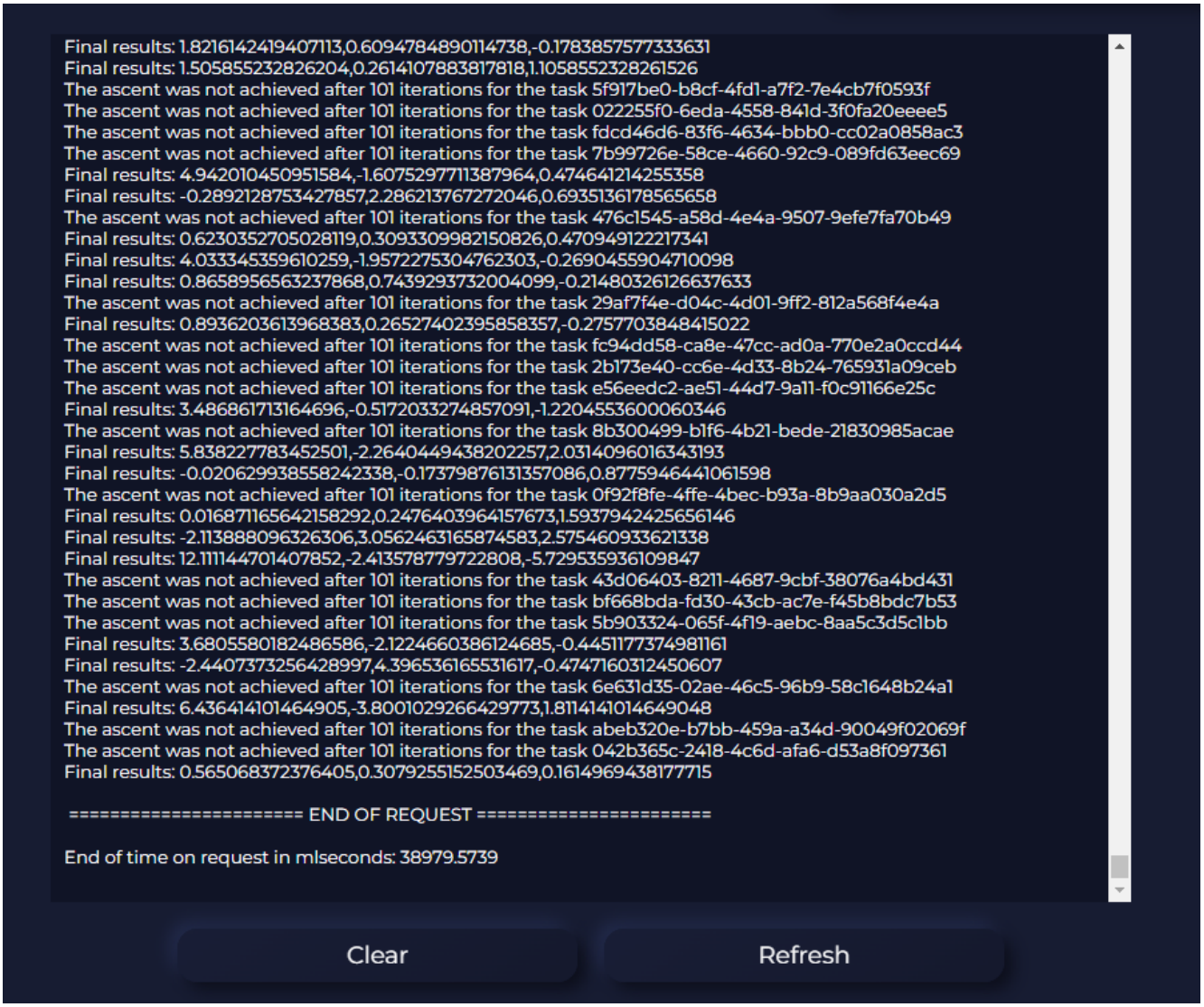


Рисунок 3.14 – Результат роботи програми з використанням 2 ядер

Отже, отримуємо результат близько 39 секунд, якщо взяти до уваги, що тут ще також присутня похибка, то можна сказати що результат достатньо непоганий, враховуючи, що працюють лише 2 ядра, нижче наведено результати паралельних розрахунків кількості виконаних завдань для кожних 0.2 секунди, рисунок 3.15.

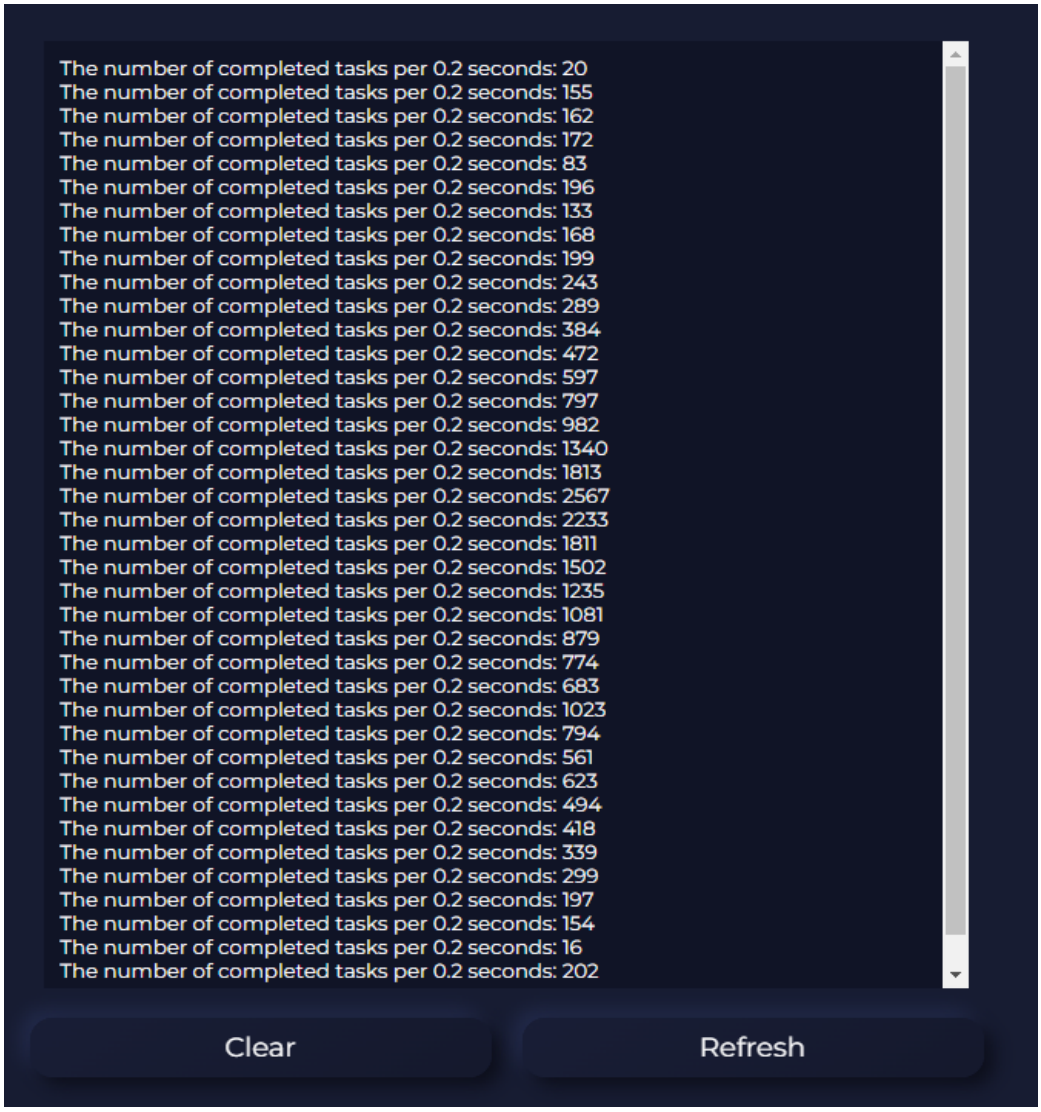


Рисунок 3.15 – Результат паралельних обчислень для 2-х ядер

Результати паралельного обчислення для 2 ядер показали кількість виконаних задач за 0.2 секунди для вирішення 50000 систем лінійних алгебраїчних рівнянь (СЛАР). Кількість виконаних задач варіювалася від 16 до 2567 за 0.2 секунди, а середні значення коливалися між 100 і 2000 виконаних задач за цей час.

Пікове значення виконаних задач досягло 2567, а найнижча кількість виконаних задач становила 16. Незважаючи на наявність пікових значень,

більшість вимірювань знаходилася в діапазоні від 150 до 2000 задач за 0.2 секунди. Такі значні коливання могли бути пов'язані з різними факторами, такими як складність задач, обчислювальні витрати або можливі накладки у використанні ресурсів.

Розглянемо навантаженість процесора під час виконання цих задач, рисунок 3.16.

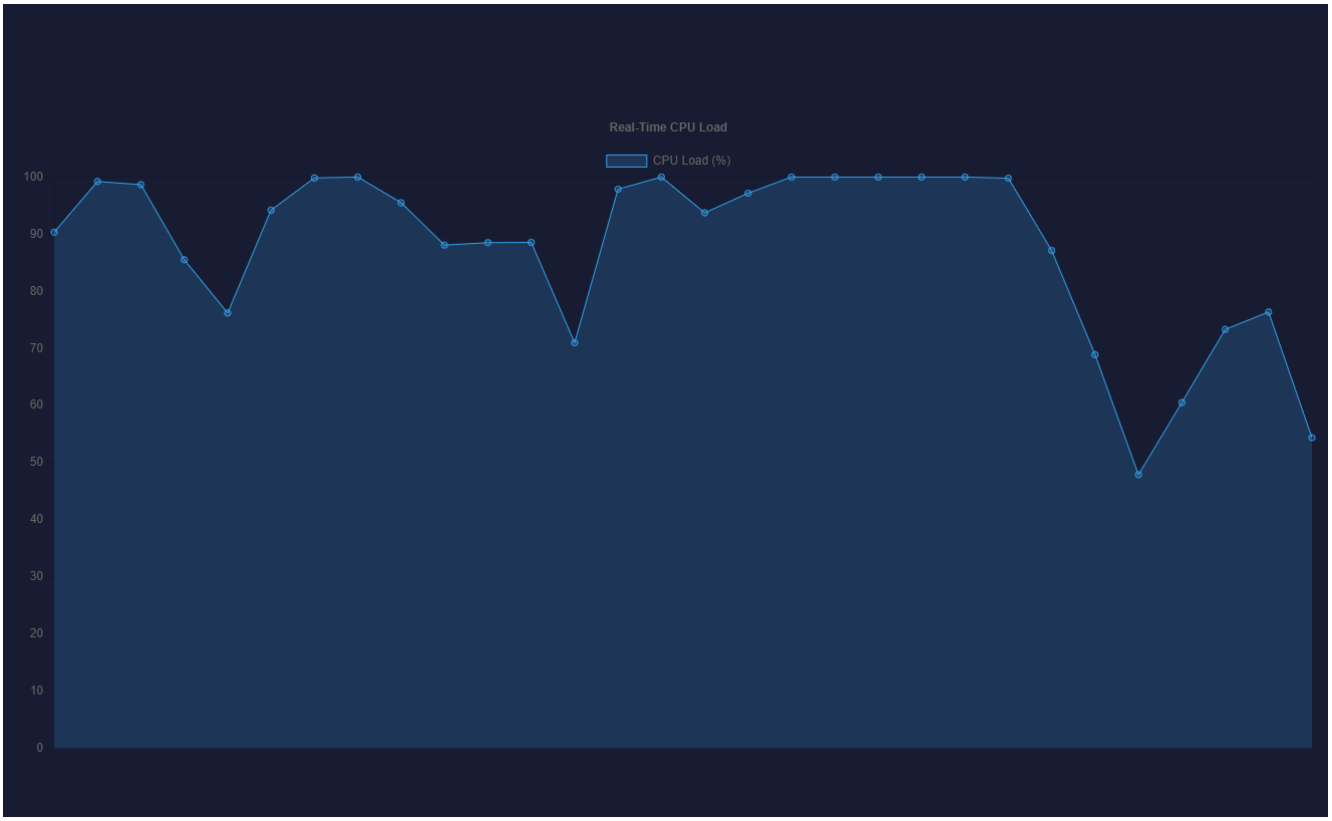


Рисунок 3.16 – Навантаженість процесора під час виконання завдань

Графік навантаження процесора показав коливання в реальному часі під час виконання паралельних обчислень. Протягом тестування спостерігалося значне навантаження на процесор, що досягало майже 100%, із періодичними спадками до 60%.

Високе навантаження процесора могло бути обумовлене запуском окремих процесів, які паралельно навантажували процесор. Це включало не тільки саму задачу обчислення систем лінійних алгебраїчних рівнянь, але й інші процеси, які могли виконуватися одночасно в системі. Така поведінка процесора відображала

характерні піки і спади, що могли виникати через різноманітні фактори, такі як обробка вхідних і вихідних даних, управління пам'яттю або інші операційні завдання, які вимагали ресурсів процесора.

Протягом більшої частини тестування навантаження на процесор залишалося стабільно високим, що свідчило про інтенсивну обчислювальну активність. Це могло вказувати на ефективне використання доступних ядер процесора, однак періодичні спади могли вказувати на необхідність додаткового аналізу для оптимізації розподілу ресурсів та зменшення втрат продуктивності.

Аналізуємо наступний графік використання оперативної пам’яті, рисунок 3.17.

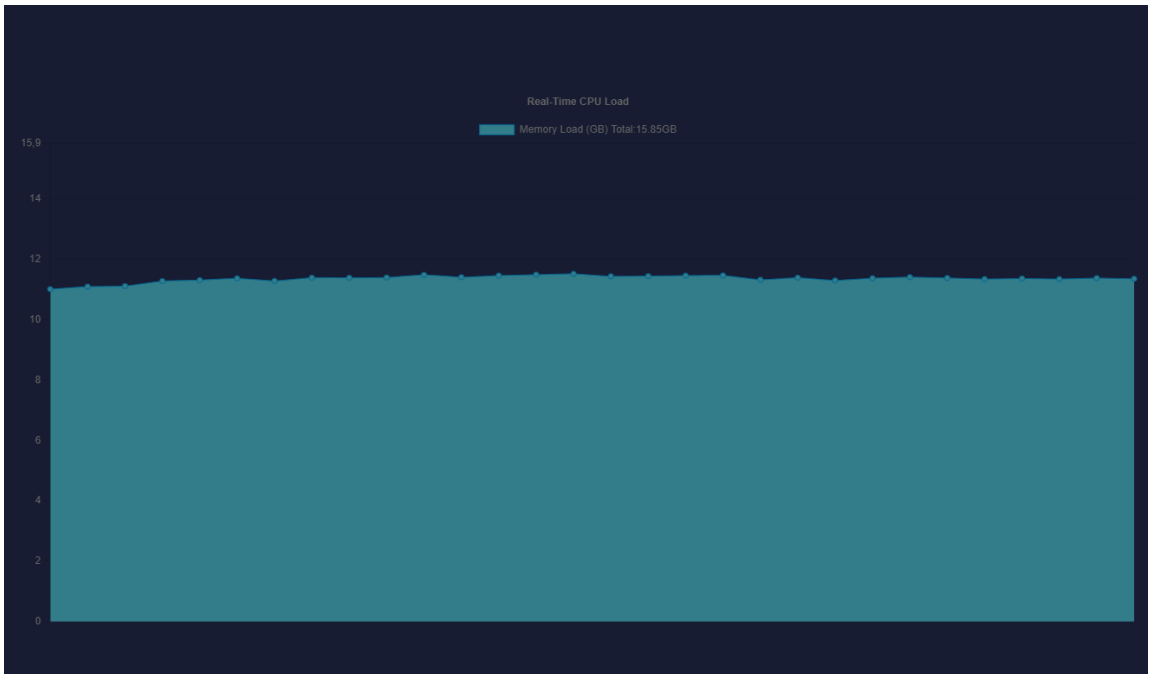


Рисунок 3.17 – Навантаженість оперативної пам’яті для 2 ядер

Графік навантаження пам'яті показав стабільність використання оперативної пам'яті протягом виконання паралельних обчислень. Протягом всього часу спостереження обсяг використаної пам'яті коливався близько 11-12 ГБ з доступних 15.85 ГБ, що вказує на ефективне використання ресурсів системи.

Стабільне навантаження пам'яті свідчило про те, що обчислювальні процеси, які виконувалися паралельно, мали стабільне використання пам'яті. Це

могло бути обумовлено тим, що задачі, які вирішувалися, вимагали постійного обсягу пам'яті, і не було значних коливань у використанні пам'яті через завантаження або вивантаження даних.

Навантаження на пам'ять також могло бути пов'язане з іншими паралельно запущеними процесами, які вимагали ресурси пам'яті. Однак відсутність різких піків або спадів у графіку вказує на те, що такі процеси не мали значного впливу на загальну стабільність використання пам'яті під час обчислень. Перейдемо до останнього графіку навантаженості мережі, рисунок 3.18.

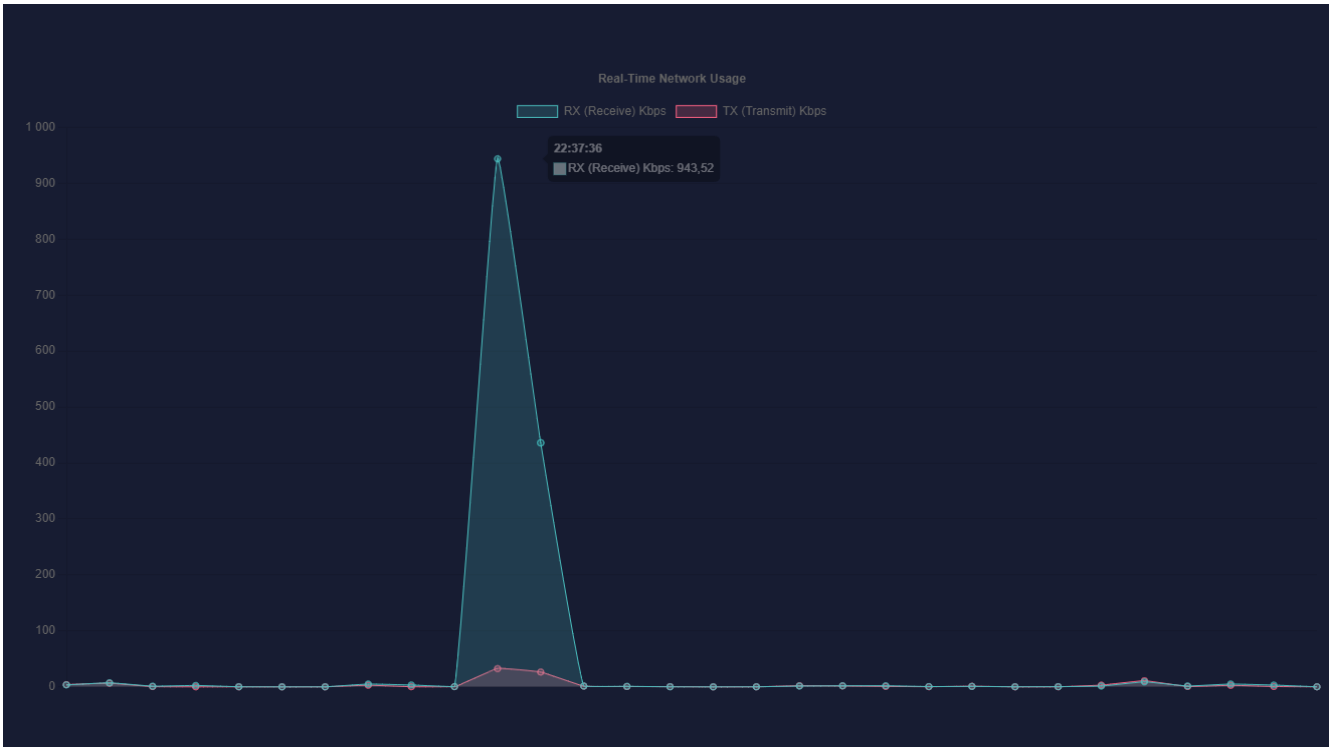


Рисунок 3.17 – Навантаженість мережі для 2 ядер

Графік навантаження мережі відобразив різкі коливання в передачі та прийомі даних під час виконання паралельних обчислень. Зокрема, спостерігався один значний пік у прийомі даних (RX), який досягав майже 943.52 Кбіт/с. Після цього піку рівень передачі та прийому даних стабілізувався на значно нижчих значеннях.

Високе навантаження мережі в певний момент могло бути обумовлене запуском окремих процесів, які паралельно передавали або приймали великі

обсяги даних. Це включало не тільки самі обчислювальні задачі, але й можливі додаткові операції, такі як завантаження даних, синхронізація або інші мережеві активності, що вимагали значних ресурсів мережі.

Після пікового навантаження спостерігалося повернення до стабільного рівня, що свідчило про тимчасовий характер високої мережевої активності. Це могло вказувати на те, що система успішно обробляла основну частину даних під час піку, а подальша робота вимагала меншого обсягу мережевих ресурсів

Тепер перейдемо до тестування роботи програми в режимі 8 ядер, на моїй системі це є максимально допустиме значення ядер. Отже розглянемо спочатку час виконання 50000 задач для 8 ядерного режиму, рисунок 3.18.

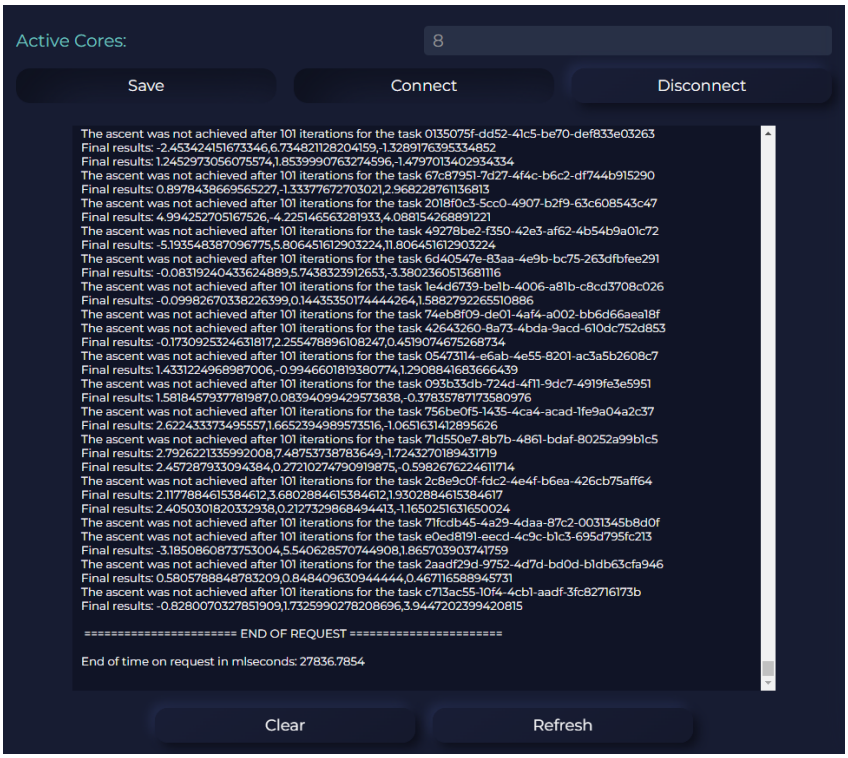


Рисунок 3.18 – Результат роботи програми з використанням 8 ядер

За підсумками проведених тестів, загальний час виконання запиту склав 27836.7854 мілісекунд, що дорівнює приблизно 27.8 секунди. Як і очікувалося, цей результат виявився більш продуктивним порівняно з режимом, де використовувалося лише два ядра процесора. У попередніх тестах час виконання

запиту складав приблизно 36 секунд. Покращення продуктивності становило близько 22.8%.

Таким чином, збільшення кількості використовуваних ядер з двох до восьми дозволило суттєво скоротити час виконання обчислень. Це підтверджує гіпотезу про те, що використання більшої кількості ядер підвищує загальну продуктивність системи при обчисленні СЛАР.

Варто зазначити, що виміри можуть бути не точними на різних системах. Існує багато факторів, які впливають на точність вимірювань, такі як швидкість запису або читання даних для різних носіїв пам'яті, швидкість інтернет-з'єднання, загальна завантаженість системи та інші. Ці фактори можуть призводити до певних відхилень у результатах.

Переглянемо кількість виконаних завдань за 0.2 секунди рисунок 3.19.

```
The number of completed tasks per 0.2 seconds: 95
The number of completed tasks per 0.2 seconds: 690
The number of completed tasks per 0.2 seconds: 215
The number of completed tasks per 0.2 seconds: 82
The number of completed tasks per 0.2 seconds: 39
The number of completed tasks per 0.2 seconds: 54
The number of completed tasks per 0.2 seconds: 94
The number of completed tasks per 0.2 seconds: 83
The number of completed tasks per 0.2 seconds: 106
The number of completed tasks per 0.2 seconds: 180
The number of completed tasks per 0.2 seconds: 267
The number of completed tasks per 0.2 seconds: 313
The number of completed tasks per 0.2 seconds: 398
The number of completed tasks per 0.2 seconds: 487
The number of completed tasks per 0.2 seconds: 616
The number of completed tasks per 0.2 seconds: 683
The number of completed tasks per 0.2 seconds: 887
The number of completed tasks per 0.2 seconds: 1191
The number of completed tasks per 0.2 seconds: 1690
The number of completed tasks per 0.2 seconds: 2318
The number of completed tasks per 0.2 seconds: 3037
The number of completed tasks per 0.2 seconds: 3689
The number of completed tasks per 0.2 seconds: 3201
The number of completed tasks per 0.2 seconds: 2745
The number of completed tasks per 0.2 seconds: 2341
The number of completed tasks per 0.2 seconds: 2011
The number of completed tasks per 0.2 seconds: 1603
The number of completed tasks per 0.2 seconds: 1353
The number of completed tasks per 0.2 seconds: 1159
The number of completed tasks per 0.2 seconds: 908
The number of completed tasks per 0.2 seconds: 847
The number of completed tasks per 0.2 seconds: 735
The number of completed tasks per 0.2 seconds: 609
The number of completed tasks per 0.2 seconds: 932
The number of completed tasks per 0.2 seconds: 380
The number of completed tasks per 0.2 seconds: 581
The number of completed tasks per 0.2 seconds: 437
The number of completed tasks per 0.2 seconds: 504
The number of completed tasks per 0.2 seconds: 486
The number of completed tasks per 0.2 seconds: 243
The number of completed tasks per 0.2 seconds: 66
The number of completed tasks per 0.2 seconds: 302
The number of completed tasks per 0.2 seconds: 202
The number of completed tasks per 0.2 seconds: 170
The number of completed tasks per 0.2 seconds: 154
The number of completed tasks per 0.2 seconds: 263
The number of completed tasks per 0.2 seconds: 33
```

Рисунок 3.19 – Результат паралельних обчислень для 8 ядер

З наведених даних видно, що кількість виконаних завдань за 0.2 секунди варіюється від 33 до 3689 завдань. Це показує, що продуктивність системи може значно змінюватися залежно від поточного навантаження, конфігурації системи та інших факторів.

Максимальна кількість виконаних завдань за 0.2 секунди становила 3689, тоді як мінімальна – лише 33 завдання. Така варіація в результатах може бути пов'язана з різними факторами, які впливають на продуктивність системи, включаючи:

– швидкість запису та читання даних: Різні носії пам'яті мають різну швидкість запису та читання даних, що може впливати на час виконання завдань;

– швидкість інтернет-з'єднання: Умови мережі можуть впливати на швидкість обміну даними між клієнтом та сервером;

– завантаженість системи: Інші процеси, що виконуються на системі, можуть впливати на продуктивність воркерів.

Важливо також враховувати, що ці виміри можуть бути неточними через згадані фактори. Наприклад, під час пікових навантажень система може працювати повільніше, а в умовах меншої завантаженості – швидше.

У середньому, система виконувала приблизно від 100 до 900 завдань за 0.2 секунди, що є хорошим показником для паралельного обчислення. Проте, варіація в результатах свідчить про необхідність подальшої оптимізації та врахування різних умов роботи системи для досягнення стабільних результатів.

Таким чином, результати виконання завдань за 0.2 секунди демонструють значні коливання продуктивності, що залежить від різних зовнішніх та внутрішніх факторів. Це підкреслює важливість комплексного підходу до оцінки продуктивності системи та врахування всіх можливих впливів на результати обчислень. Перейдемо, до наступних результатів обрахунків: навантаженість процесора в режимі роботи для 8 ядер, рисунок 3.20.

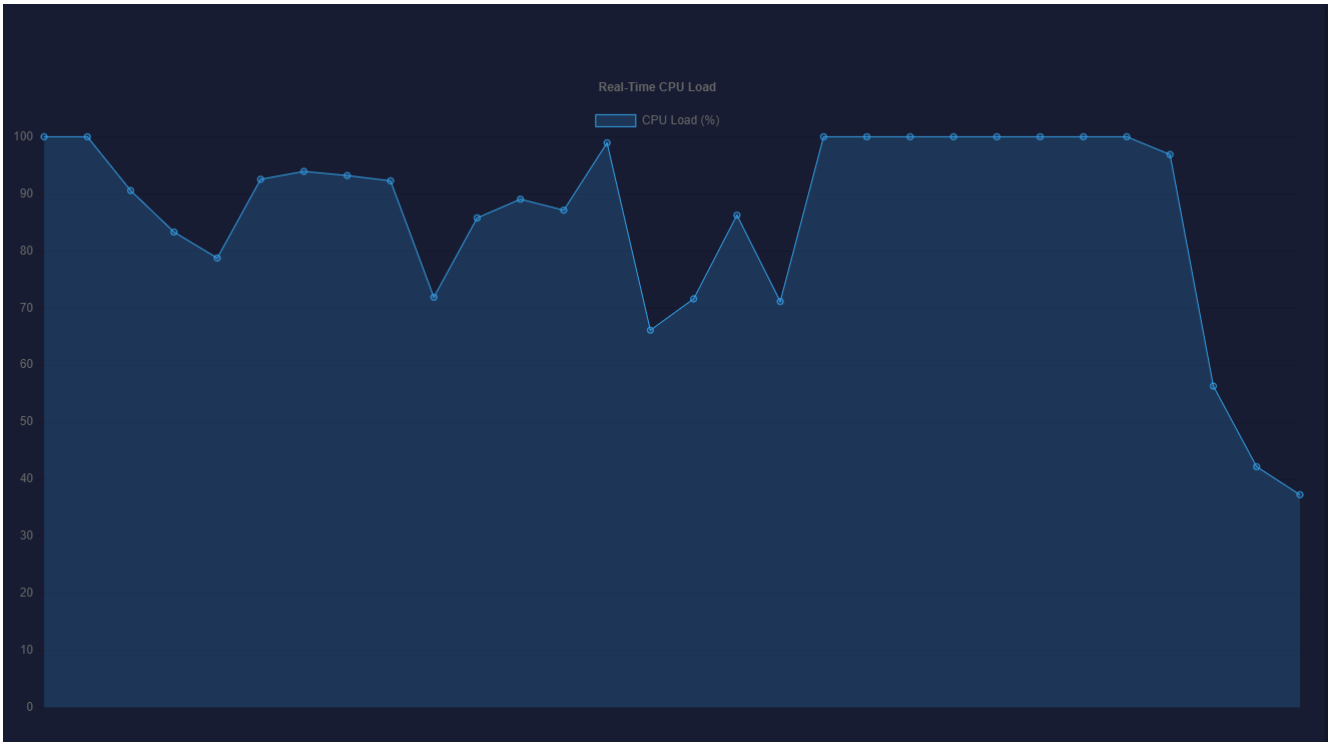


Рисунок 3.20 – Навантаженість процесора під час виконання завдань

На графіку представлено навантаження процесора в реальному часі при використанні 8 ядер. Графік демонструє зміну навантаженості процесора протягом обчислювального процесу, що дозволяє оцінити ефективність використання ресурсів системи.

На початку роботи процесор був завантажений майже на 100%, що свідчило про інтенсивне використання всіх ядер для обчислення завдань. Протягом деякого часу навантаження зберігалося на високому рівні, з невеликими коливаннями в межах 90-100%. Це показало, що система активно використовувала всі доступні ядра для паралельного обчислення.

Далі спостерігалися декілька періодів, коли навантаження процесора значно знижувалося до рівня 60-70%. Це могло бути викликано закінченням деяких завдань і необхідністю переназначення ресурсів для нових завдань, або іншими внутрішніми факторами, які впливали на розподіл роботи між ядрами.

Після цих періодів навантаження процесора знову поверталося до рівня близько 90%, що свідчило про продовження інтенсивної обчислювальної діяльності. Згодом процесор знову працював на максимальному завантаженні

протягом тривалого періоду, що підтвердило стабільну роботу системи при високому навантаженні.

Наприкінці роботи спостерігалося значне зниження навантаження процесора до рівня близько 30-40%. Це свідчило про завершення основних обчислювальних завдань і переходу системи в менш інтенсивний режим роботи. Таке зниження навантаження було очікуваним, оскільки після завершення основних завдань потреба в інтенсивному використанні всіх ядер зникла. Аналізуємо наступний графік використання оперативної пам’яті, рисунок 3.21.

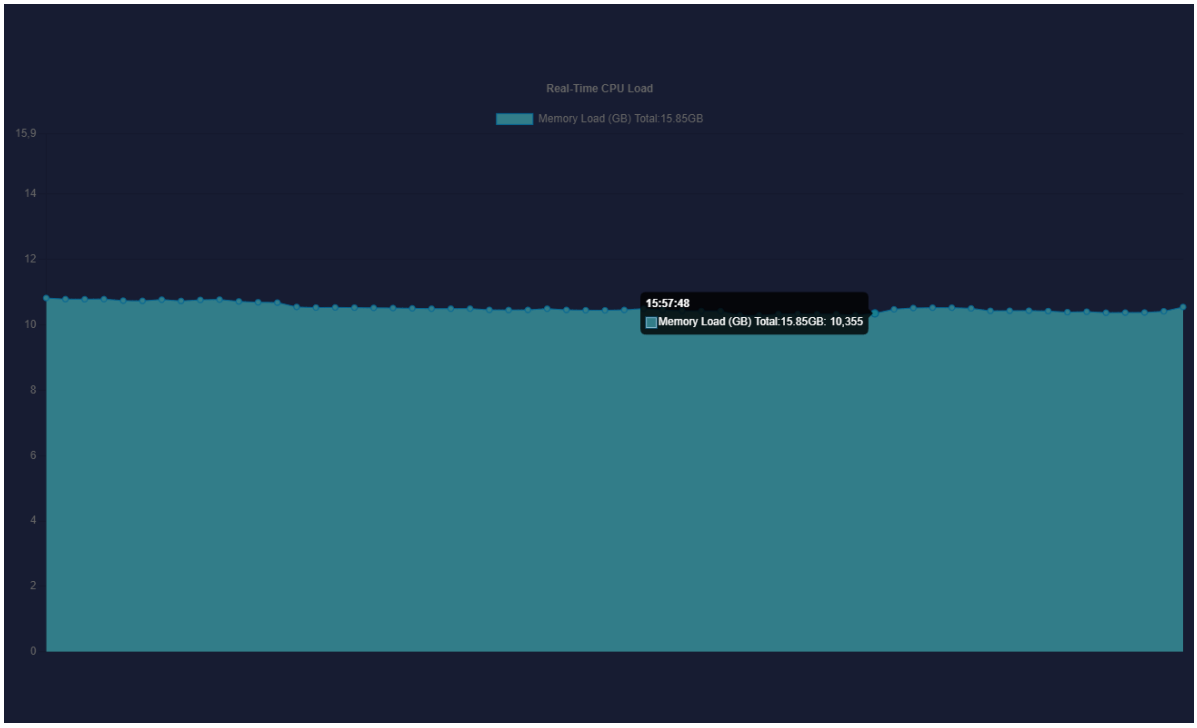


Рисунок 3.21 – Навантаженість оперативної пам’яті для 2 ядер

На початку роботи обсяг використаної оперативної пам'яті становив близько 10.4 ГБ з доступних 15.85 ГБ. Протягом всього періоду виконання завдань використання пам'яті залишалося стабільним, з незначними коливаннями в межах декількох сотень мегабайтів. Це свідчило про ефективне управління ресурсами пам'яті і відсутність великих витоків або надмірних коливань у споживанні пам'яті.

Навантаження пам'яті підтримувалося на рівні близько 10.3-10.4 ГБ, що дозволяло системі виконувати паралельні обчислення без значного збільшення використання оперативної пам'яті. Це також підтверджує, що програма ефективно використовувала доступну пам'ять, розподіляючи завдання між потоками та уникаючи перевантаження системи.

Наприкінці періоду обчислень обсяг використаної пам'яті трохи зменшився, що могло бути пов'язано із завершенням виконання деяких завдань та звільненням зайнятих ресурсів. Загалом, використання пам'яті залишалося стабільним протягом всього часу виконання завдань, що свідчить про стабільну роботу системи і її здатність ефективно управляти пам'яттю.

Як і в інших вимірюваннях, слід враховувати можливі похибки, викликані різними факторами. Наприклад, різні апаратні конфігурації та фонові процеси можуть впливати на загальне використання пам'яті. Проте, на основі цього графіку можна зробити висновок, що система здатна ефективно використовувати доступну оперативну пам'ять для виконання паралельних обчислювальних завдань. Перейдемо до останнього графіку навантаженості мережі, рисунок 3.22.

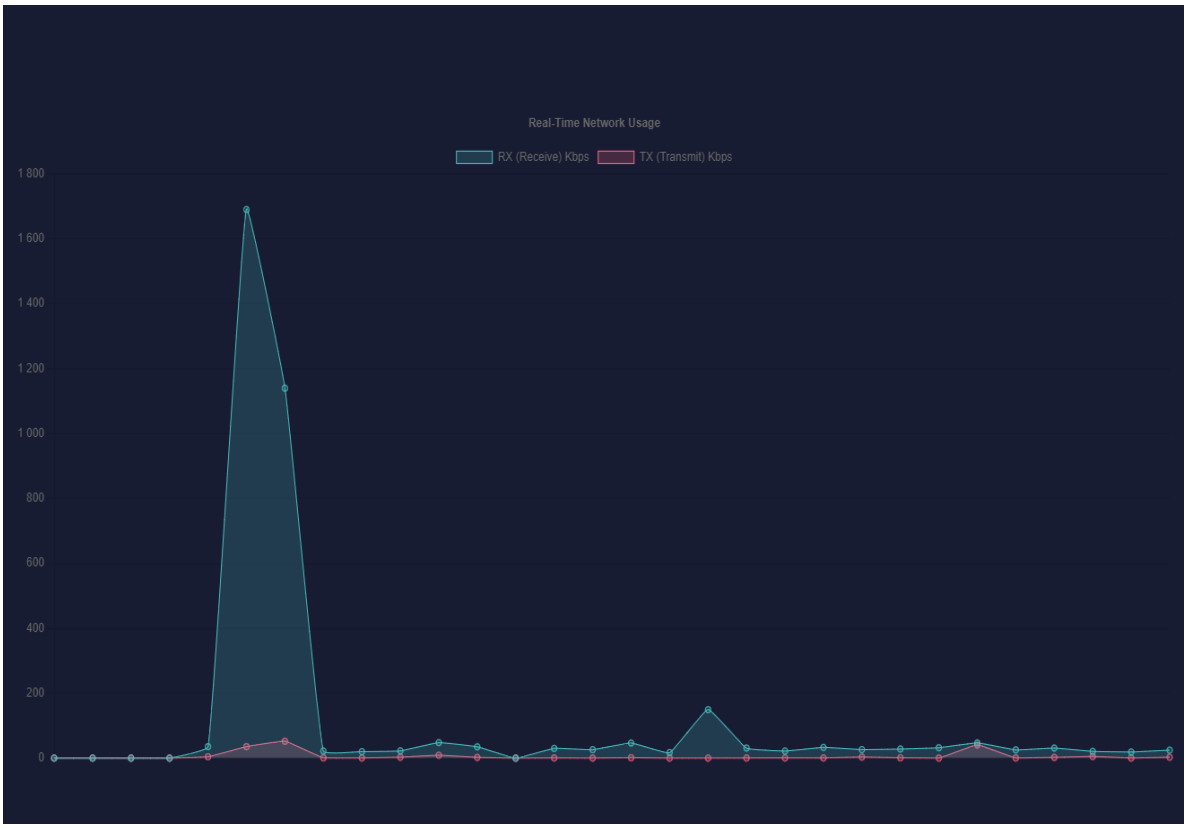


Рисунок 3.17 – Навантаженість мережі для 8 ядер

На початку роботи спостерігався значний пік отриманих даних, який досягав приблизно 1600 Kbps. Цей піковий момент може бути пов'язаний з початковим завантаженням даних, необхідних для обчислення завдань. Після цього навантаження на мережу значно знизилося і стабілізувалося на рівні близько 100-200 Kbps для отриманих даних.

Передача даних (TX) була значно менш інтенсивною протягом всього періоду. Максимальні значення для переданих даних становили близько 200 Kbps, а в середньому коливалися в межах 50-100 Kbps. Це вказує на те, що система передавала менше даних порівняно з отриманням, що може бути обумовлено природою завдань, які вимагали більше вхідних даних для обчислень.

Протягом середньої частини періоду виконання завдань навантаження на мережу залишалося відносно стабільним з невеликими коливаннями. Це свідчило про те, що система підтримувала постійний обсяг обміну даними, необхідний для обчислення завдань і обміну результатами з сервером.

Наприкінці періоду виконання завдань спостерігалися невеликі піки як для отриманих, так і для переданих даних, що могли бути пов'язані з завершенням обчислень і передачею підсумкових результатів. Загалом, обсяг отриманих даних залишався вищим, ніж обсяг переданих даних, що свідчило про домінуюче споживання мережевих ресурсів для отримання інформації.

Висновки

Особливу увагу було приділено організації розподілу завдань між потоками. Використання WorkerPool забезпечило ефективне розподілення обчислень між доступними ядрами процесора, що дозволило оптимізувати продуктивність системи. На основі створеної діаграми було показано, як завдання розподіляються між потоками, як контролюється їх виконання та як забезпечується збір результатів.

Вплив кількості ядер на продуктивність клієнта було продемонстровано через практичні виміри. Збільшення кількості активних ядер до восьми показало значне покращення продуктивності в порівнянні з двоядерним режимом. Було відзначено, що час обробки задач скоротився на приблизно 23%, що підтверджує ефективність використання більшої кількості ядер для паралельних обчислень. Проте було зауважено, що виміри можуть мати певні похибки через різні фактори, такі як швидкість запису та читання даних, швидкість інтернет-з'єднання та інші.

Також було проведено обрахунки результатів кількості виконаних задач, паралельно, під час виконання основного завдання, що дало змогу продемонструвати більш наявне представлення про продуктивність роботи в різних ядерних режимах.

Було детально розглянуто роботу алгоритму виконання однієї частини завдання для одного потоку, із збереженням в спільну структуру даних, значень, потрібних для подальшого виконання обчислень лінійних алгебраїчних рівнянь, на кожній ітерації.

Завдяки реалізованим засобам моніторингу навантаженості процесору, оперативної пам’яті та навантаженості мережі, було продемонстровано вплив в різних режимах роботи програми, на компоненти системи користувача, що, звісно ж, дає змогу біль достовірно оцінити продуктивність роботи застосунку, повторюючись, в залежності від вибору режиму роботи застосунку.

Таким чином, третій розділ роботи демонструє успішну реалізацію програмного забезпечення, що включає розробки інтерфейсів, організації мережевої комунікації, розподілу завдань та оптимізації продуктивності. Це забезпечує високу ефективність та зручність у використанні програмного засобу для обробки поведінкових даних обчислювальних елементів розподіленої системи.

ВИСНОВОК

У цьому дослідженні було розглянуто методологію опрацювання поведінкових даних обчислювальних елементів розподіленої системи, зокрема на основі Node.js. Дана тема є надзвичайно актуальною в сучасному світі, де обсяги даних швидко зростають, а швидкість та ефективність їх обробки є критичними для успіху бізнесу та забезпечення безпеки і стабільності систем.

Node.js, як рішення для створення серверних додатків з використанням JavaScript, дозволяє розробникам легко створювати потужні програмні засоби для опрацювання поведінкових даних у розподілених системах. Його невисока вартість відносно інших технологій, велика спільнота розробників та широкі можливості зробили Node.js одним з популярних виборів для цієї задачі.

Програмні засоби для опрацювання поведінкових даних обчислювальних елементів розподіленої системи на основі Node.js дозволяють збирати, аналізувати та реагувати на дані в реальному часі. Вони допомагають організаціям виявляти аномальні поведінки, прогнозувати виникнення проблем та управляти ресурсами ефективніше.

Однак, важливо враховувати, що успішне впровадження програмних засобів для опрацювання поведінкових даних обчислювальних елементів розподіленої системи вимагає не лише технічних знань, але й вміння вірно оцінювати потреби бізнесу та вибирати найбільш підходящі рішення для конкретної ситуації.

Отже, використання програмних засобів для опрацювання поведінкових даних обчислювальних елементів розподіленої системи на основі Node.js є обгрунтованим кроком для організацій, що прагнуть оптимізувати свої процеси, підвищити ефективність та залишатися конкурентоспроможними в умовах постійних змін на ринку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. De Assuncao, Marcos Dias, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing. *A survey on resource elasticity and future directions. Journal of Network and Computer Applications*. 2018. №103. P. 1-17.
2. Kersting, William H. *Distribution system modeling and analysis. Electric power generation, transmission, and distribution*. CRC press, 2018. P. 15-26.
3. Lamport, Leslie. Time, clocks, and the ordering of events in a distributed system. Concurrency: the Works of Leslie Lamport. 2019. 234 p.
4. Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016. 300 p.
5. Florio, Luca. Decentralized self-adaptation in large-scale distributed systems."Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015. 1112 p.
6. Giannoula, Christina, et al. Syncron: Efficient synchronization support for near-data-processing architectures. International Symposium on High-Performance Computer Architecture (HPCA). 2021. 324 p.
7. Sriram, Sundararajan, and Shuvra S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. 2018. №2. P. 14-18.
8. Fox, Geoffrey C., Roy D. Williams, and Guiseppe C. Messina. Parallel computing works! 2014. 976 p.
9. Golub, Gene H., and James M. Ortega. Scientific computing: an introduction with parallel computing. Elsevier, 2014. 442 p.
10. Hockney, Roger W., and Chris R. Jesshope. Parallel Computers 2: architecture, programming and algorithms. CRC Press, 2019. 642 p.
11. Vasile, Mihaela-Andreea, et al. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Generation Computer Systems*. 2015. №51. P. 61-71.

12. Ge, Xiaohua, Fuwen Yang, and Qing-Long Han. Distributed networked control systems: A brief overview. Information Sciences. 2017. 151 p.
13. Chen, Tianqi, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015. 401 p.
14. Chen, Rong, et al. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. ACM Transactions on Parallel Computing. 2019. 47 p.
15. Sari, Arif, and Murat Akkaya. Fault tolerance mechanisms in distributed systems. *International Journal of Communications, Network and System Sciences.* 2015. P. 471-482.
16. Verbraeken, Joost, et al. A survey on distributed machine learning. *Acm computing surveys*. 2020. P. 1-33.
17. Cheng, John, Max Grossman, and Ty McKercher. Professional CUDA c programming. 2014. 466 p.
18. Qiu, Hongjian, et al. Yafim: a parallel frequent itemset mining algorithm with spark. *international parallel & distributed processing symposium workshops*. 2014. №5. P. 140-167.
19. Venkataramani, Swagath, et al. Approximate computing and the quest for computing efficiency. *Proceedings of the 52nd Annual Design Automation Conference*. 2015. №1.2. P. 41-99.
20. Csuhaj-Varjú, Erzsébet, et al. Grammar systems: a grammatical approach to distribution and cooperation.2018. 254 p.
21. Anderson, Ross. Security engineering: a guide to building dependable distributed systems. 2020. 1182 p.
22. Himmel, M. Azua, and F. Grossman. Security on distributed systems: Cloud security versus traditional IT. *IBM Journal of Research and Development*. 2014. №58. P. 1-3.
23. Brown, Philip N., Holly P. Borowski, and Jason R. Marden. Security against impersonation attacks in distributed systems. IEEE Transactions on Control of Network Systems. 2018. №6.1. P. 440-450.

24. Mbanaso, U. M., and G. A. Chukwudebe. Requirement analysis of IoT security in distributed systems. *IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON).* 2017. P. 777-781.
25. Willis, H. Lee. Distributed power generation: planning and evaluation. 2018. 616 p.
26. Adefarati, Temitope, and Ramesh C. Bansal. Integration of renewable distributed generators into the distribution system: a review. *IET Renewable Power Generation.* 2016. №10.7. P. 873-884.
27. Blaabjerg, Frede, et al. Distributed power-generation systems and protection. *Proceedings of the IEEE*. 2017. №105.7. P. 1311-1331.
28. Ametepe, Wolali, et al. Data provenance collection and security in a distributed environment: a survey. *International Journal of Computers and Applications*. 2021. №43.1. P. 11-25.
29. Chang, Victor, and Muthu Ramachandran. Towards achieving data security with the cloud computing adoption framework. *IEEE Transactions on services computing*. 2015. №9.1. P. 138-151.
30. Aldin, Hesam Nejati Sharif, et al. "Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications." arXiv preprint arXiv:1902.03305 (2019).
31. Van Der Aalst, Wil, and Wil van der Aalst. Data science in action. 2016. 321 p.
32. Cliff, Norman. Ordinal methods for behavioral data analysis. Psychology Press. 2014. 212 p.
33. Kambatla, Karthik, et al. Trends in big data analytics. *Journal of parallel and distributed computing*. 2014. №74.7. P. 2561-2573.
34. Wilson, Robert C., and Anne GE Collins. "Ten simple rules for the computational modeling of behavioral data." Elife 8 (2019): e49547.

35. Chen, CL Philip, and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information sciences* . 2014. №275. P. 314-347.
36. Thurner, Leon, et al. pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems. IEEE Transactions on Power Systems. 2018. №33.6. P. 6510-6521.
37. Saritha, Vara, Namuduri Srinivas, and N. V. Srikanth Vuppala. Analysis and optimization of coagulation and flocculation process. *Applied Water Science*. 2017. №7. P. 451-460.
38. Bhosekar, Atharv, and Marianthi Ierapetritou. Advances in surrogate based modeling, feasibility analysis, and optimization: A review. *Computers & Chemical Engineering*. 2018. №108. P. 250-267.
39. Rao, Marada Srinivasa, et al. Application for Mood Detection of Students Using TensorFlow and Electron JS. *Machine Learning and Big Data Analytics*. 2022. P. 235.
40. Fedosejev, Artemij. React. js essentials. Packt Publishing Ltd, 2015. 118 p.
41. Le, Thai. *Comparing State Management Between Redux And Zustand In React*. 2023. P. 16-43.
42. Wohlgethan, Eric. Supportingweb development decisions by comparing three major javascript frameworks: Angular, react and vue. js. Diss. *Hochschule für Angewandte Wissenschaften Hamburg*. 2018. 95 p.
43. Duldulao, Devlin Basilan. ASP. NET Core and Vue. js: *Build real-world, scalable, full-stack applications using Vue. js 3, TypeScript,. NET 5, and Azure. Packt Publishing Ltd*. 2021. 56 p.
44. Rose, Alexander S., and Peter W. Hildebrand. NGL Viewer: a web application for molecular visualization. *Nucleic acids research*. 2015. №43.1. P. 576-579.
45. Skvorc, Dejan, Matija Horvat, and Sinisa Srbljic. Performance evaluation of Websocket protocol for implementation of full-duplex web streams.

International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). 2014. №37. P. 14-56.

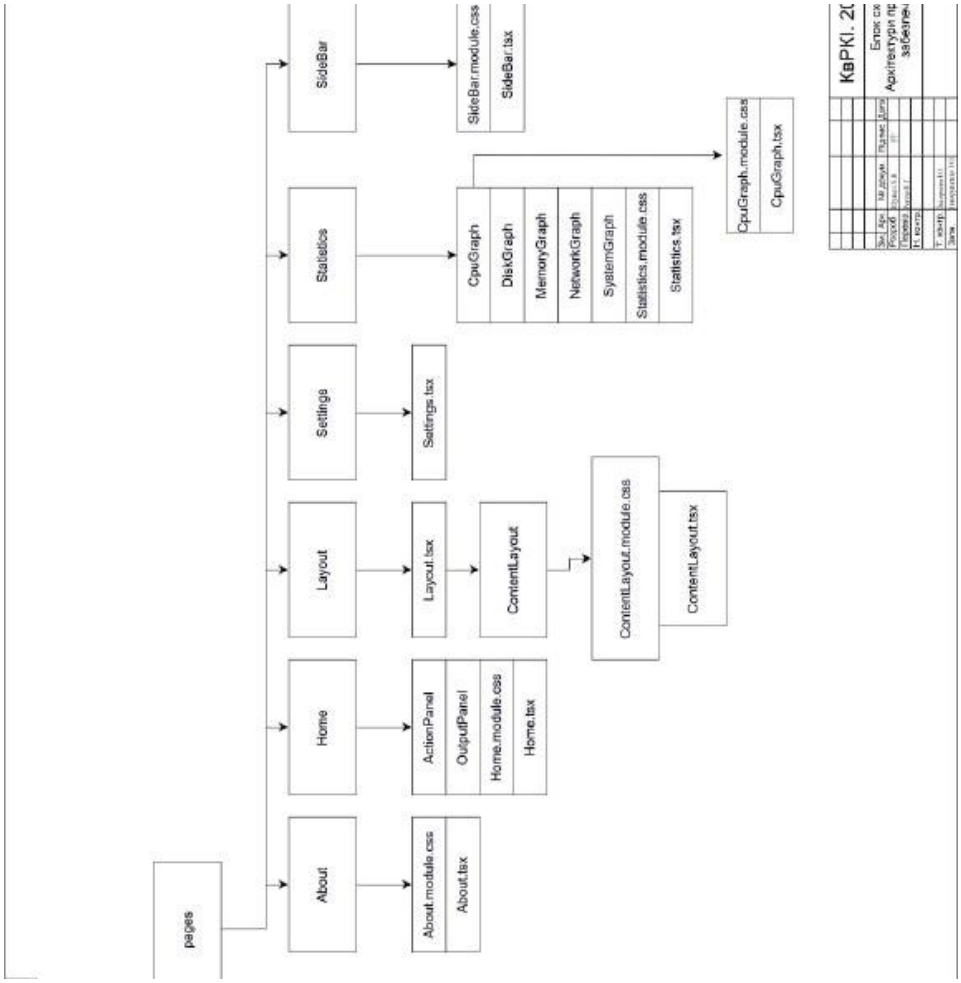
46. Muller, Gabriel L. HTML5 WebSocket protocol and its application to distributed computing. 2014. 352 p.
47. Karlström, Juuso. The WebSocket protocol and security: best practices and worst weaknesses. 2016. 251 p.
48. Ogundeyi, K. E., and C. Yinka-Banjo. WebSocket in real time application. *Nigerian Journal of Technology*.2019. №38.4. P. 1010-1020.
49. Abdullah, Ako Muhamad. Advanced encryption standard (AES) algorithm to encrypt and decrypt data. *Cryptography and Network Security*. 2017. №16.1. P. 11.
50. Subramanyan, Pramod, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2015. P. 152-187.
51. Abikoye, Oluwakemi Christiana, et al. Modified advanced encryption standard algorithm for information security. *Symmetry*. 2019. №11.12. P. 1484.
52. Altigani, Abdelrahman, et al. A polymorphic advanced encryption standard–a novel approach. *IEEE Access*. 2021. №9. P. 20191-20207.
53. Teng, Lin, et al. A Modified Advanced Encryption Standard for Data Security. *Int. J. Netw. Secur.* 2020. №22.1. P. 112-117.
54. D'souza, Flevina Jonese, and Dakshata Panchal. Advanced encryption standard (AES) security enhancement using hybrid approach. *International Conference on Computing, Communication and Automation (ICCCA)*. 2017. №31. P. 1420-1429.
55. Zhang, Qi, and Qun Ding. Digital image encryption based on advanced encryption standard (AES). *Fifth International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC). IEEE*. 2015. №5.7. P. 142-332.

56. Tonde, Ashwini R., and Akshay P. Dhande. Implementation of Advanced Encryption Standard (AES) Algorithm Based on FPGA. *International Journal of Current Engineering and Technology*. 2014. №4.2. P.112-139.
57. Bhanot, Rajdeep, and Rahul Hans. A review and comparative analysis of various encryption algorithms. *International Journal of Security and Its Applications*. 2015. №9.4. P. 289-306.
58. Mushtaq, Muhammad Faheem, et al. A survey on the cryptographic encryption algorithms. *International Journal of Advanced Computer Science and Applications.* 2017. №8.11. P. 87-94.
59. Wu, Hongjun, and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. Selected Areas in Cryptography—SAC. 2014. №12. P. 41-49.
60. Subramanyan, Pramod, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST).* 2015. №3. P. 32-39.

ДОДАТОК А

(обов’язково)

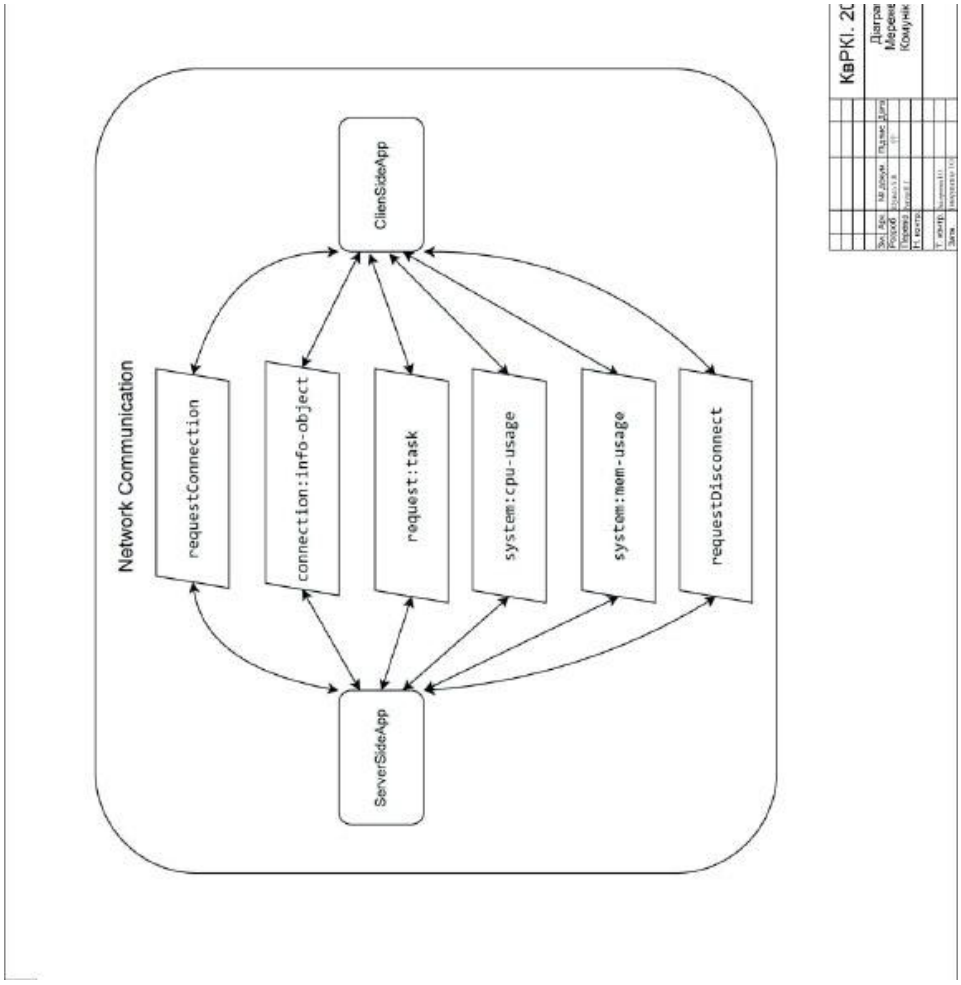
Копія креслення схеми архітектури програмного забезепечення



ДОДАТОК Б

(обов’язково)

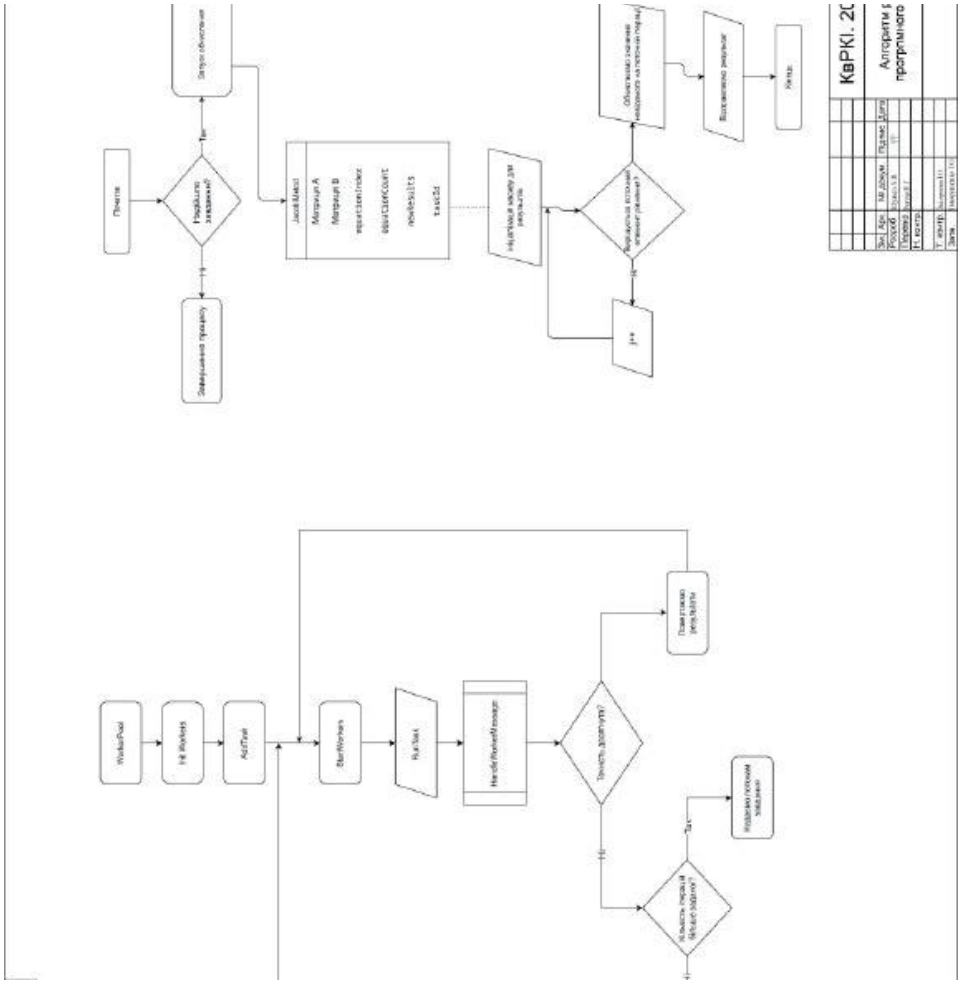
Копія креслення мережевої комунікації



ДОДАТОК В

(обов’язково)

Копія креслення алгоритму роботи програмного додатку



ДОДАТОК Г

(обов’язково)

Кодування WorkerPool.js

```
const path = require("node:path")
const { Worker } = require("node:worker_threads")
const { v4 } = require("uuid")
const fs = require("fs")
const perf = require("execution-time")()

export class WorkerPool {
```

```
constructor(
    workerScript,
    maxWorkers,
    logCallback,
    endCallBack,
    sendLogForMeasurments
) {
    this.workerScript = workerScript
    this.maxWorkers = maxWorkers
    this.workers = []
    this.taskQueue = []
    this.currentEqNum = 0
    this.tolerance = 0.01
    this.activeTasks = new Map()
    this.currentActiveTasks = 0
    this.completedTasks = 0 // Track the number of completed tasks
    this.maxIterations = new Map()
    this.interval = undefined
    this.logCallback = logCallback
    this.logForMeasurments = sendLogForMeasurments
    this.endWorkStatus = endCallBack

    this.startTime = null // Track start time of tasks

    this.startTaskReporting() // Start reporting task completion every 2 seconds
}

// Start reporting completed tasks every 2 seconds
startTaskReporting() {
    this.interval = setInterval(() => {
```

```
        if (this.completedTasks > 0) {
            this.logToFile(
                `The number of completed tasks per 0.2 seconds: ${this.completedTasks}`,
                "onTask.txt"
            )
            this.completedTasks = 0 // Reset the count after reporting
        }
    }, 200)
}

// Function to log messages to a file
logToFile(message, name = "log.txt") {
    const logFilePath = path.join(__dirname, name)
    fs.appendFile(logFilePath, message + "\n", (err) => {
        if (err) {
            console.error("Failed to write to log file:", err)
        }
    })
}

initWorkers() {
    for (let i = 0; i < this.maxWorkers; i++) {
        this.createWorker(i)
    }
}

// Створення воркера
createWorker(id) {
    const worker = new Worker(this.workerScript)
```

```
    worker.on("message", (result) => {
        this.handleWorkerMessage(result, worker)
    })

    worker.on("error", (err) => {
        console.error(err)
    })

    worker.on("exit", (code) => {
        console.log(`Worker ${id} stopped with exit code ${code}`)
        this.workers[id] = null
    })

    this.workers[id] = worker
}

// Обробка повідомлення від воркера
handleWorkerMessage(result, worker) {
    const task = this.activeTasks.get(result.taskId)
    if (!task) {
        console.error(`Task with ID ${result.taskId} not found.`)
        this.logToFile(`Task with ID ${result.taskId} not found.`)

        return
    }
    task.newResults[result.index] = result
    task.currentEqNum++
    if (task.currentEqNum === task.tasks[0].equationCount) {
        task.iterations++
        if (
```

```
            Math.max(...task.newResults.map((res) => res.toleranceDif)) <
            this.tolerance
        ) {
            const logMessage = `The ascent was achieved after ${task.iterations} iterations for the task ${task.taskId}`
            this.logToFile(logMessage)
            const finalRes = [...task.newResults.map((res) => res.x_new)]
            this.logToFile(`Final results: ${finalRes}`)

            this.completedTasks++ // Increment the count of completed tasks

            this.activeTasks.delete(task.taskId)
            if (this.activeTasks.size <= 0) {
                this.logToFile(
                    `\n ======================== END OF REQUEST ======================== \n`
                )
                this.endWorkStatus()
                const results = perf.stop()
                this.logToFile(
                    `End of time on request in mlseconds: ${results.time}`
                )
            }
        } else {
            if (task.iterations > this.maxIterations.get(task.taskId)) {
                const logMessage = `The ascent was not achieved after ${task.iterations} iterations for the task ${task.taskId}`
                console.log(logMessage)
                this.logToFile(logMessage)
                const finalRes = [
```

```
                ...task.newResults.map((res) => res.x_new)
            ]

            this.logToFile(`Final results: ${finalRes}`)

            this.completedTasks++ // Increment the count of completed tasks

            this.activeTasks.delete(task.taskId)
            if (this.activeTasks.size <= 0) {
                if (this.logCallback)
                    this.logToFile(
                        `\n ========== END OF REQUEST ========== \n`
                    )
                this.endWorkStatus()
                const results = perf.stop()
                this.logToFile(
                    `End of time on request in mlseconds: ${results.time}`
                )
            }
            this.startWorkers()
            return
        }
        task.currentEqNum = 0
        task.tasks.forEach((item, index) => {
            this.taskQueue.push({
                ...item
            })
        })
        task.tasks.length = 0
```

```
            task.prevResults = [...task.newResults.map((res) => res.x_new)]
            this.startWorkers()
        }
    }

    if (this.taskQueue.length > 0) {
        this.runTask(worker, this.taskQueue.shift())
    }
}

// Запуск задачі на воркері
runTask(worker, task) {
    if (!this.activeTasks.has(task.id)) {
        this.activeTasks.set(task.id, {
            taskId: task.id,
            tasks: [task],
            newResults: [],
            iterations: 0,
            currentEqNum: 0,
            prevResults: []
        })
    } else {
        this.activeTasks.get(task.id).tasks.push(task)
    }
    const prevTask = this.activeTasks.get(task.id)
    worker.postMessage({
        ...task,
        newResults: prevTask.prevResults
    })
}
```

```
    startWorkers() {
        // if (!this.startTime) {
        //         this.startTime = Date.now() // Set start time when workers start in milliseconds
        // }
        for (let i = 0; i < this.maxWorkers; i++) {
            if (this.workers[i] && this.taskQueue.length > 0) {
                this.runTask(this.workers[i], this.taskQueue.shift())
            }
        }
    }

    addTask(taskData, maxIterations = 100) {
        const taskId = v4() // Унікальній ідентифікатор задачі
        this.taskQueue.push(
            ...taskData.map((data, index) => ({
                id: taskId,
                equationIndex: index,
                ...data,
                equationCount: taskData.length
            }))
        )
        this.maxIterations.set(taskId, maxIterations)

        perf.start()
        this.startWorkers()
    }

    // Завершення всіх воркерів
```

```
terminateAll() {
    this.workers.forEach((worker) => {
        if (worker) {
            worker.terminate()
        }
    })
    this.workers = []
    clearInterval(this.interval)
                                                    }
                                                }
```

UNICHECK
by Turnitin

Ім'я користувача:
Кафедра КІ

Дата перевірки:
04.06.2024 23:09:34 EEST

Дата звіту:
05.06.2024 07:50:11 EEST

ID перевірки:
1016321104

Тип перевірки:
Doc vs Internet + Library

ID користувача:
100005591

Назва документа: Шульга_Програмний засіб для опрацювання поведінкових даних обчислювальних елемен...

Кількість сторінок: 75 Кількість слів: 13044 Кількість символів: 103166 Розмір файлу: 3.72 MB ID файлу: 1016119474

7.38%
Схожість

Найбільша схожість: 0.59% з Інтернет-джерелом (http://elar.khmnu.edu.ua/jspui/bitstream/123456789/12227/1/%d0%93.

7.05% Джерела з Інтернету 863 ..... Сторінка 77

1.71% Джерела з Бібліотеки 67 ..... Сторінка 82

0% Цитат

Не знайдено жодних цитат

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

Wed Jun 05 06:54:10 EEST 2024, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 9%**

ID: 128361 Назва: БКР Програмний засіб для опрацювання поведінкових даних обчислювальних елементів розподіленої системи Додано в БД: 2024-06-05 Автора: Х. В. Шульга Керівники: П.Г Регіда Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	92478	733	710 (1%)	10 (1%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Шульга Харітон Вячеславович

Тема: Програмний засіб для опрацювання поведінкових даних обчислювальних елементів розподіленої системи

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 Кількість сторінок записки 71

1. Короткий зміст роботи та прийнятих рішень: В рамках дипломного проекту розглядалася розробка програмного засобу для опрацювання поведінкових даних обчислювальних елементів розподіленої системи. Основною метою роботи було забезпечення ефективного збору та аналізу даних, що відображають функціонування обчислювальних компонентів у розподіленій системі, а також реалізацію програмного додатку для виконання обчислювальних задач на хості.

2. Висновок про відповідність ДП дипломному завданню: Дипломний проект у повній мірі відповідає поставленому завданню як в теоретичній, так і в практичній частині даного проекту.

3. Характеристика виконання кожного розділу проекту, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі розглянуто принципи та типи паралельних обчислень, а також методи синхронізації для ефективного використання ресурсів. У другому розділі обгрунтовано вибір Node.js та systeminformation для збору поведінкових даних, використання HTTP та WebSocket протоколів для комунікації, а також технологій React, Zustand та Electron для реалізації візуального інтерфейсу програмного додатку. Усі розділи демонструють використання сучасних технологій для забезпечення ефективності, масштабованості та надійності проекту.

4. Позитивні сторони роботи: Проект є актуальним завдяки зростаючій потребі в ефективних засобах для обробки поведінкових даних в розподілених системах та організації розподілених обчислень в цілому. Проект містить інструменти для

детального моніторингу та аналізу стану системи, що сприяє оптимізації її роботи та підвищенню ефективності обчислювальних процесів.

5. Негативні сторони роботи: Негативними сторонами проекту є похибки, що виникають під час підрахунку результатів виконання, що зменшує точність вимірів результатів тестування. А також містить незначні недоліки оптимізації під час формування обчислювальних потоків, що може впливати на продуктивність.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: Розглянувши позитивні та негативні сторони представленої дипломної роботи вважаю, що робота заслуговує оцінки «відмінно», 4.75 (А).

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) _____

доцент кафедри АКІТтаР Кофцова Л.О.

05 06 2024 р. _____ (підпис)

Завідувачу кафедри КНС
д-р.техн.наук, проф. Говорущенко Т. О.

Шульги Харітона Вячеславовича

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-20-2

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений(а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

3 червня 2024 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Програмний засіб для опрацювання поведінкових даних обчислювальних елементів розподіленої системи

Автор: Шульга Харітон Вячеславович

Спеціальність: 123– Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Регіда Павло Генадійович, ст. Викладач

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) усі запозичення фрагментарні, або мають належним чином оформлені посилання;
3) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 20-67 джерелами на один фрагмент речення;
4) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ідентичності/схожості, складає 7.38% і адресується до 930 першоджерела, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи П.Г. Регіда

Гарант ОП С.М. Лисенко

Завідувач кафедри КНС Т. О. Говорущенко