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

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Балансувальник задач для динамічної розподіленої системи обчислень
Назва теми

КвРКІ.200239.20.02.14 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 ______ О. В. Матьокін
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 Т.О. Говорущенко

" 10 " 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Матьокіну Олегу Вячеславовичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Балансувальник задач для динамічної розподіленої системи обчислень

Керівник проекту (роботи) Регіда 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)

Аналіз відомих рішень та теорія розподілених систем

Обґрунтування вибору програмних засобів

Програмна реалізація системи обчислень

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

Архітектура клієнт-серверної системи

Блок-схема алгоритма балансувальника

Схема життєвого циклу сервера

6. Консультанти розділів дипломного проекту (роботи)

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

7. Дата видачі завдання «_10_» _01_ 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
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	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2024	виконано
4	Робота над розділом 2 – обґрунтування вибору програмних засобів	01.04.2024	виконано
5	Робота над розділом 3 – програмна реалізація системи розподілених обчислень	30.04.2024	виконано
6	Оформлення пояснювальної записки згідно вимог	31.05.2024	виконано
7	Попередній захист ВКР	30.05.2024	виконано
8	Захист ВКР на засіданні ЕК	Червень 2024 року	

Студент _________ О. В. Матьокін
Підпис Ініціали, прізвище

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

№ рядка	Формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 200239.20.02.14 ПЗ	Пояснювальна записка	71		
			Графічні матеріали			
2		КвРКІ 200239.20.02.14 Е8	Архітектура клієнт-серверної системи	1		
3		КвРКІ 200239.20.02.14 Е8	Блок-схема алгоритма балансувальника	1		
4		КвРКІ 200239.20.02.14 Е8	Схема життєвого циклу сервера	1		

КвРКІ 200239.20.02.14 ВП

Зм	Арк	№ докум	Підпис	Дата
Розробив		Мальокін О.В.		5.06
Перевір.		Регіда П.Г.		5.06
Н. контр.		Засорнова І.О.		5.06
Затв.		Говорущенко Т.О.		6.06

Відомість проекту

Літера	Аркуш	Аркушів
У	1	1

ХНУ, КІ2-20-2

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Балансувальник задач для динамічної розподіленої системи обчислень».

Автор роботи: Матьокін Олег Вячеславович.

Керівник роботи: Регіда Павло Генадійович.

Пояснювальна записка: 71 с., 16 рис., 1 табл., 4 дод., 50 джерел.

Графічна частина: 3 креслення.

БАЛАНСУВАЛЬННЯ ЗАДЧА, КОМУНІКАЦІЙНА АРХІТЕКТУРА, МОНТІРОГИНГ СТАНУ СИСТЕМИ.

Метою роботи є розробка балансувальника задач для розподіленої системи обчислень.

У цій роботі розроблено алгоритм балансування задач для розподіленої системи обчислень на мові JavaScript з використанням платформи NodeJS. Розроблено також гібридну комунікаційну архітектуру, яка включає в себе протоколи HTTP та WebSockets. Крім основного додатку розроблено програму для моніторингу системи, що дозволяє швидко виявляти потенційні проблеми.

Підпис студента

30.05.2024

Дата

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ ВІДОМИХ РІШЕНЬ ТА ТЕОРІЯ РОЗПОДІЛЕНИХ СИСТЕМ ... 10

1.1 Поняття розподілених систем та їх типи 10

1.2 Необхідність балансування задач у розподілених системах та аналіз існуючих алгоритмів 17

Висновки 26

2 ОБҐРУНТУВАННЯ ВИБОРУ ПРОГРАМНИХ ЗАСОБІВ 27

2.1 Вибір мови програмування для реалізації серверної частини 27

2.2 Вибір технологій для розробки засобу моніторингу 32

2.3 Засоби для організації комунікації в системі 35

2.4 Захист передачі даних та їх зберігання 38

Висновки 42

3 ОРГАНІЗАЦІЯ БАЛАНСУВАННЯ ТА КОМУНІКАЦІЇ В РОЗПОДІЛЕНІЙ СИСТЕМІ 44

3.1 Реалізація обраного алгоритма балансування 44

3.2 Реалізація механізмів комунікації 52

3.3 Реалізація засобу моніторинга серверу 61

Висновки 69

ВИСНОВКИ 70

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 72

ДОДАТОК А 76

ДОДАТОК Б 77

ДОДАТОК В 78

ДОДАТОК Г 79

Зм.	Арк.	№ докум.	Підпис	Дата	КвРКІ. 200239.20.02.14 ПЗ		
Виконав		Матьокін О.В.		5.06	Балансувальник задач для динамічної розподіленої системи обчислень	Літера	Аркуш	Аркушів
Перевір.		Регіда П.Г.		5.06			6	71
Н.контр.		Засорнова І.О.		5.06		ХНУ. КІ2-20-2		
Затвер.		Говорущенко Т.О.		6.06				

ВСТУП

В сучасному світі в час швидкого розвитку технологій, коли обчислювальні завдання стають все більш складними, об’ємними та необхідними, динамічні розподілені системи обчислень набувають все більшої популярності. Такі системи дозволяють ефективно використовувати ресурси комп’ютерів особливо багатоядерних, розподіляючи завдання між різними вузлами мережі в залежності від їхньої доступності, продуктивності та поточної завантаженості. Звісно такі системи вимагають досить об’ємних знань для їх розробки та підтримки, їх розробка потребує вирішення багатьох задач, особливо задач балансування. Ефективність такої системи на пряму залежить від методу розподілення завдань з урахуванням можливостей системи.

Тому саме тема балансувальння задач для динамічної розподіленої системи обчислень стає все більш актуальною і вимагає досліджень та розвитку нових підходів. Балансувальник задач для динамічної розподіленої системи обчислень повинен відповідати декільком важливим критеріям. Він має бути реактивним і адаптивним, швидко реагувати на зміни в стані вузлів, такі як збої або зміна продуктивності, а також адаптуватися до нових умов роботи системи. Ефективність використання ресурсів є ще одним критичним аспектом, що передбачає оптимальне використання доступних обчислювальних ресурсів та запобігання простою і перевантаженню окремих вузлів.

Балансувальник повинен бути масштабованим, тобто мати можливість ефективно працювати як у малих, так і у великих системах з різною кількістю вузлів. Важливими також є простота інтеграції, яка забезпечує легкість впровадження та інтеграції з існуючими обчислювальними середовищами та системами, і безпека та надійність, що включають захист від зловмисних дій та стабільну роботу системи навіть у разі відмови окремих компонентів.

Балансувальник задач є критично важливим компонентом для забезпечення ефективної роботи динамічних розподілених систем обчислень. Дослідження та розробка нових підходів у цій сфері відкривають можливості для підвищення продуктивності обчислювальних процесів, що має велике значення для сучасного

інформаційного суспільства. Подальший розвиток методів балансування та їх інтеграція з новітніми технологіями стане ключовим фактором успіху у створенні високопродуктивних розподілених систем обчислень.

Отже, дослідження та розвиток знань у сфері балансування задач для динамічних розподілених систем обчислень є важливим кроком у напрямку оптимізації обчислювальних процесів та підвищення їхньої продуктивності. У результаті, наукові та практичні зусилля в цій області сприятимуть створенню більш ефективних, надійних та продуктивних обчислювальних систем, що відповідають зростаючим вимогам до обробки та аналізу даних.

Мета кваліфікаційної роботи буде забезпечення працездатності динамічної розподіленої системи обчислень за допомогою балансувальника задач.

Поставлена мета досягається реалізацією власного балансувальника на основі існуючих алгоритмів планування роботи з урахуванням особливостей динамічних розподілених систем обчислень, та проведенням відповідних експериментів.

Об'єктом дослідження є процес розподілення завдань за допомогою балансувальника між обчислювальними елементами динамічних розподілених систем.

Предметом дослідження є функціонування серверного додатку що використовує алгоритм балансувальника навантаження в системі, та забезпечує комунікацію елементів системи.

Для досягнення поставленої мети використовуються теоретичні та практичні засади функціонування серверного додатку, методи передачі даних по мережі, підходи до балансування навантаження між учасниками розподіленої системи обчислень.

Практичне значення має спроєктована система для організації розподілених обчислень із асинхронним балансувальником завдань.

1 АНАЛІЗ ВІДОМИХ РІШЕНЬ ТА ТЕОРІЯ РОЗПОДІЛЕНИХ СИСТЕМ

1.1 Поняття розподілених систем та їх типи

Загалом розподілену систему можна розглядати як набір незалежних один від одного комп’ютерів, які користувач сприймає як єдину об’єднану систему. Основними завданнями такої розподіленої системи є збір, обробка, організація ефективного доступу користувачів до інформації [1-2]. Існує багато типів розподілених систем, серед яких слід виділити кластерні розподілені системи обчислень, симетрично-багатопроцесорні розподілені системи обчислень, грід-розподілені системи обчислень та граничні розподілені системи обчислень.

Кластерні розподілені системи обчислень – це системи в яких декілька комп’ютерів, або ж як їх ще називають «Обчислювальні елементи» об’єднуються, щоб діяти як єдиний ресурс [3]. Ці елементи працюють разом для досягнення спільної мети, розділяючи ресурси та завдання. Слід також зазначити, що кластерні розподілені системи працюють в межах локальної мережі, досить часто це якесь приміщення де розміщено багато обчислювальних машин, які і з’єднуються в кластер. Це можна сказати перший тип розподілених систем який приходить на думку на питання що таке розподілена система.

На відміну від деяких інших типів розподілених систем, кластерні системи досить легко масштабувати [4-5]. Також оскільки це кластерна система, то вихід з ладу одного з підключених вузлів не припинить роботу всієх системи, а просто зменшить її потужність. На сьогоднішній день такі системи досить популярні, та застосовуються для вирішення різноманітних задач, останнім часом задачі які активно вирішуються на таких системах включають в себе:

- вирішення задач з галузі біоінформатики, наприклад аналіз геномних та біологічних даних для виявлення генетичних змін, дослідження дії лікарських препаратів , прогнозування захворювань;
- обробка великих обсягів даних або ж як їх ще називають «Big Data». Обробка таких даних включає в себе аналіз великих обсягів структурованих і

неструктурованих даних з метою виявлення залежностей, трендів та визначення патернів;

– обчислення великої продуктивності «High Performance Computing» . Наприклад використання потужних комп’ютерних систем для виконання складних обчислювальних завдань, таких як моделювання клімату, обробка сигналів, геноміки та інших.

Загалом кластерні системи також можуть містити різні допоміжні компоненти, так наприклад кластерна розподілена система обчислень з високою доступністю може містити кластер серверів-диспетчерів, які будуть відповідати за розподіл завдань, робиться це з метою забезпечення безперебійної роботи системи, адже якщо активний сервер виходить з ладу, його функції бере на себе один із резервних [6-7]. Таку систему у її простій формі схематично зображено на рисунку 1.1.

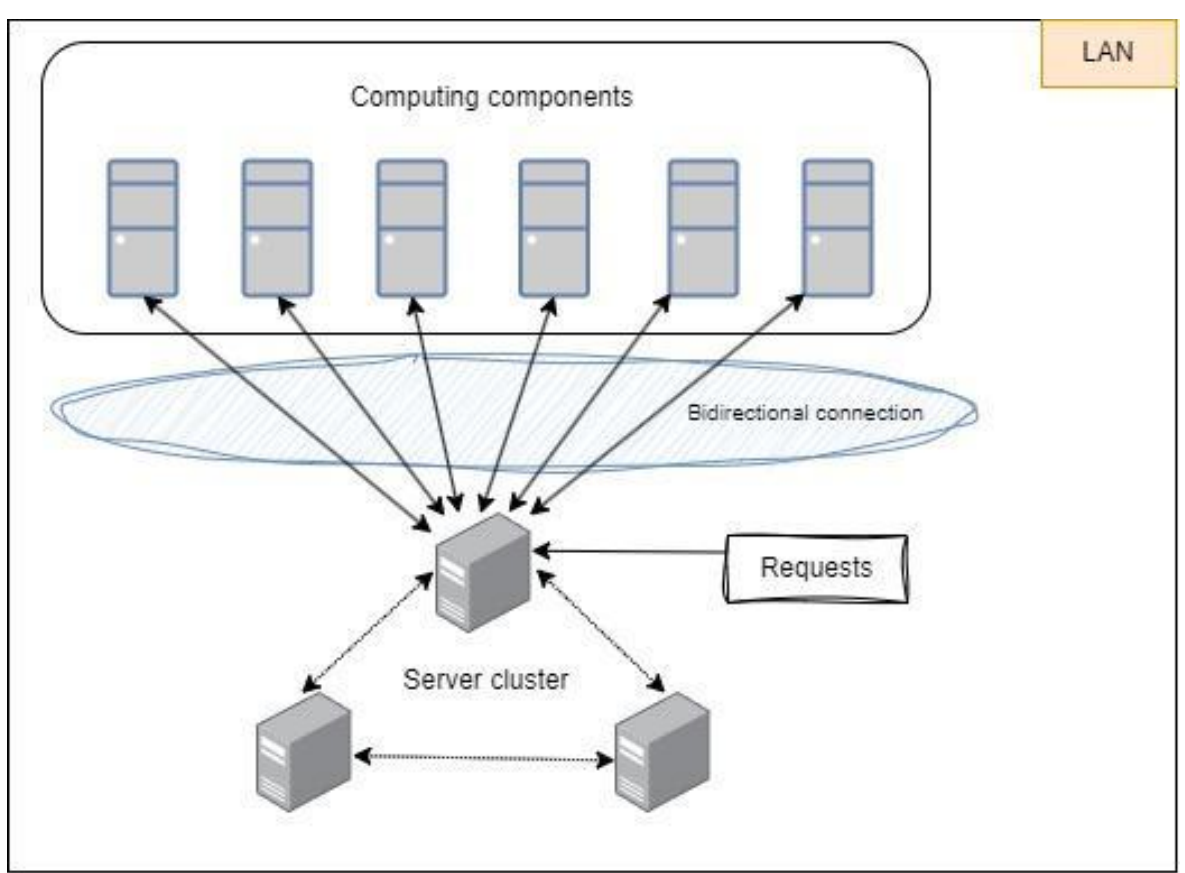


Рисунок 1.1 – Кластерна розподілена система обсчислень з високою доступністю

Як і інші розподілені системи, кластерні розподілені системи потребують високого рівня знань для їх конфігурації. Налаштування того ж кластера потребує досить глибоких знань у сфері мереж і обчислювальних систем. Помилки в конфігурації можуть призвести до неправильної роботи системи або навіть до її відмови. Те ж саме можна сказати і про ефективне розподілення ресурсів між вузлами кластера, адже це також може бути складною задачею, особливо якщо навантаження змінюються динамічно. Тому кластерні системи хоча й досить поширені та ефективні, як і всі інші розподілені системи, вони мають свої недоліки.

Симетрично-багатопроцесорні розподілені системи або ж СБРО, це системи в яких використовується велика кількість процесорів, такі системи з'явилися раніше ніж кластерні по причині того, що раніше мережі були не так розвинуті як зараз, і коли була потреба збільшити продуктивність та обчислювальні можливості системи, її вирішували шляхом додавання до системи додаткових процесорів [8]. Звісно разом з додаванням процесорів потрібно було і вирішувати проблему керування цими процесорами, та передачі даних між ними, тому також до такої системи додавали спільну пам'ять. Звісно така система мала певні недоліки та складнощі в реалізації.

Наприклад проблема синхронізації, як було сказано для таких систем додавалась спільні пам'ять, доступ до якої мали декілька процесорів, що часто призводило до проблем синхронізації доступу. Після чого у системи зменшувалась коректність роботи та продуктивність, що для розподілених систем обчислення є ключовими характеристиками [9]. Звісно при чіткій та правильній маніпуляції даними з відповідною синхронізацією такі системи завдяки паралельній обробці даних значно збільшували продуктивність порівняно з системами з одним процесором;

Ще однією проблемою такої системи, яку слід відмітити є погана масштабованість після певного моменту покращення системи, адже існує практична межа кількості процесорів, які можуть бути ефективними під час виконання певних завдань.

Хоча на сьогодні такі системи використовують не дивлячись на складність їх розробки досить часто, тому що у СБРО ресурси можуть бути дуже ефективно

використані, оскільки завдання можуть розподілятися між вузлами в залежності від їх доступності [10]. Ця перевага робить такі системи потрібними й на сьогоднішній день, і буде тримати їх на плаву ще дуже довго.

Необхідно згадати також про таку характеристику як висока доступність даних систем. Завдяки розподіленому характеру системи, СБРО дозволяють підвищити доступність, оскільки вони можуть продовжувати працювати, навіть якщо один чи кілька вузлів відмовляють. Звісно завдання забезпечення високої доступності таких систем не з легких, але реалізувати його можливо, і при правильному використанні даної характеристики можна будувати дуже надійні системи такого типу. Отож прикладами СБРО зараз є суперкомп'ютери, робочі станції для наприклад відеомонтажу або ж 3D моделювання. Ну і звісно ж СБРО часто використовують як сервери де потрібна висока продуктивність для обробки запитів від користувачів. Схематично структура такої системи зображена на рисунку 1.2.

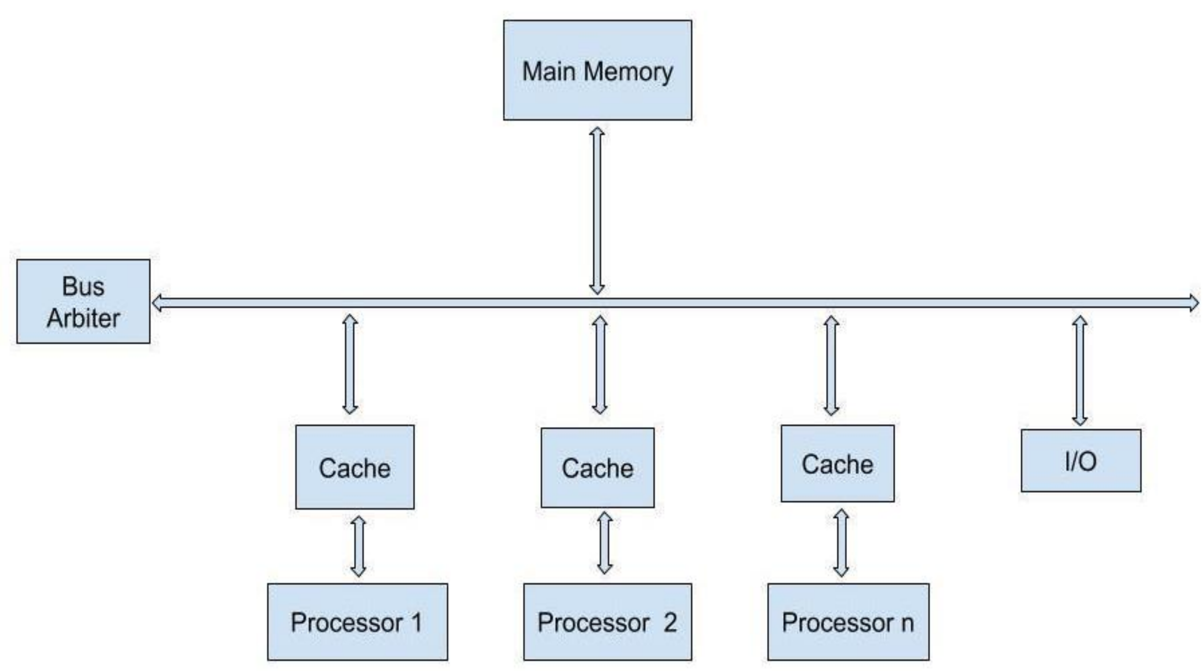


Рисунок 1.2 – Структура симетрично-багатопроцесорної розподіленої системи

Грід-розподілена система обчислень або ж ГРСО. Насправді, ця система певним чином схожа на кластерну розподілену систему, але в неї є одна ключова відмінність, а саме географічне положення вузлів. Тобто це підтип розподіленої системи, де ресурси вузлів можуть бути розподілені географічно між різними комп'ютерами та серверами, і ці ресурси можуть належати різним організаціям або адміністраторам [11-14]. Грід-розподілена система обчислень розглядається як розширення концепції розподіленого обчислення, де вузли мають різну власність і контролюються різними організаціями.

Головна ідея полягає в тому, щоб об'єднати обчислювальні ресурси з різних джерел і різних місць для спільного використання, максимізуючи продуктивність та ефективність. Грід-розподілені системи зазвичай використовуються для великих наукових, інженерних або бізнес обчислень, де потрібно багато ресурсів для вирішення складних завдань.

Як і з кластерними системами, ГРСО досить легко масштабувати, і на відміну від кластерних систем, вони не обмежені однією мережею або приміщенням. Декілька слів потрібно сказати і про ефективність таких систем. Використання обчислювальних ресурсів з різних джерел та місць дозволяє оптимізувати використання цих ресурсів і максимізувати ефективність системи в цілому. Але оскільки така система може включати в себе комп’ютери, які розташовані на великих відстанях це може призвести до збільшення мережевих затримок, що може вплинути на продуктивність деяких обчислювальних завдань. Також оскільки ГРСО чутлива до перерв у мережевому з'єднанні між вузлами, відсутність зв'язку може призвести не тільки до збільшення затримок, а й до втрати доступності або навіть до зупинки роботи деяких операцій. Крім того налагодження та тестування додатків у ГРСО може бути важким через розподілений характер системи та різнорідність середовищ.

Існують також певні нюанси з безпекою в таких системах. Забезпечення безпеки в грід-розподілених системах обчислень може бути складним через розподіленість ресурсів та даних [15-16]. Це може створювати потенційні проблеми з доступом та конфіденційністю. Не слід і забувати про те, що управління ГРСО може бути складним завданням через необхідність координації ресурсів і завдань

між різними власниками та адміністраторами. А оскільки комп'ютерів в такі системі може бути дуже багато, завдання управління стає ще складнішим. Топологію ГРСО наведено на рисунку 1.3.

Розуміння вищенаведених недоліків та складнощів допомагає компаніям враховувати їх при плануванні, розробці та впровадженні грід-розподілених систем обчислень, тим самим уникнувши непередбачуваних проблем. Слід відмітити, що частіше всього це проблеми пов'язані з мережами.

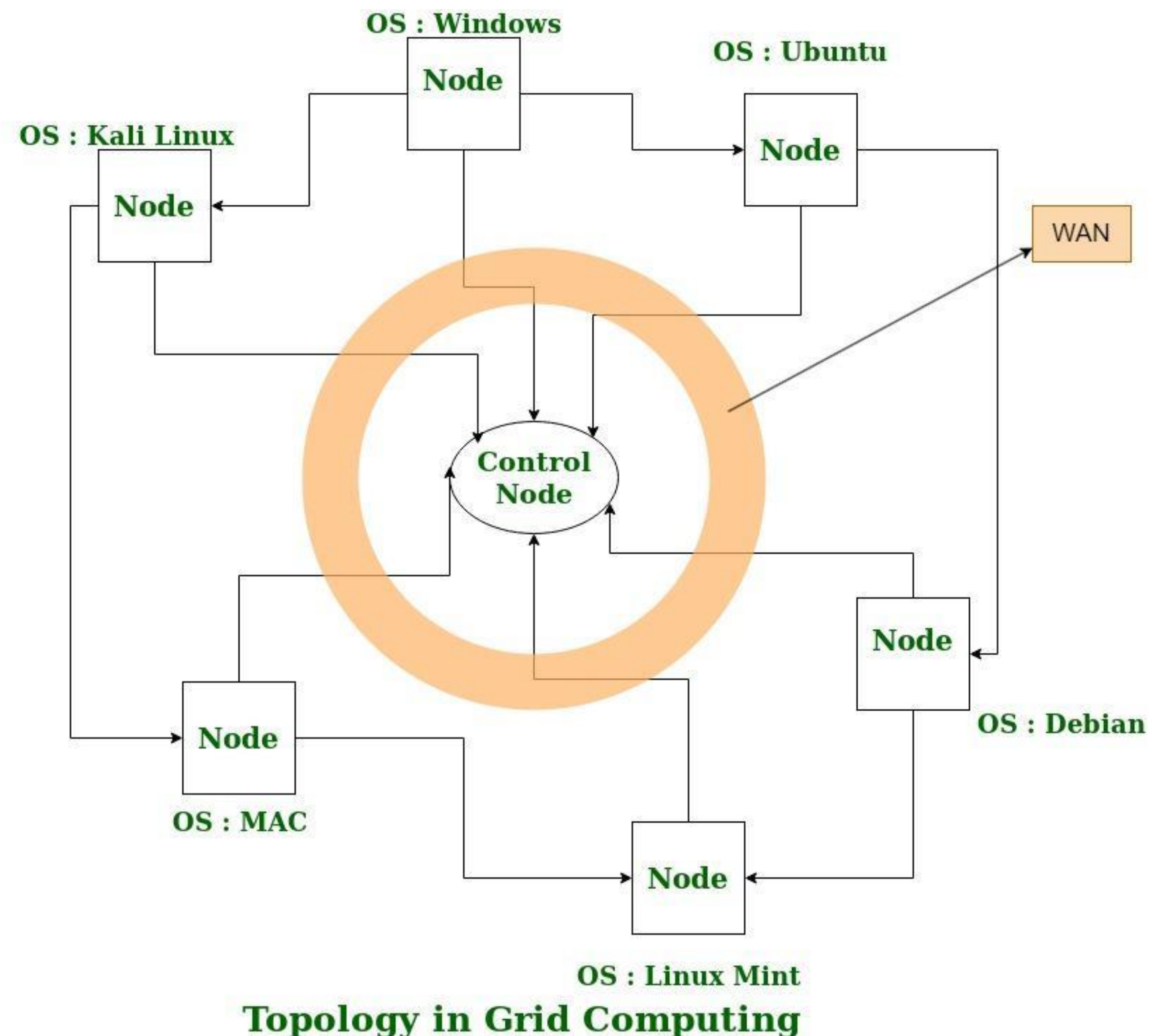


Рисунок 1.3 – Топологія грід-розподіленої системи обчислень

Хоча типів розподілених систем є досить багато, не можна не розглянути граничні розподілені системи обчислень. Це системи, які на сьогоднішній день дуже стрімко набирають популярності, оскільки людство все більше поглиблюється у IoT сферу [17-18]. Граничні розподілені системи обчислень – це

архітектурні рішення, які передбачають розміщення обчислювальних ресурсів та обробку даних близько до місця їх виникнення, тобто на краю мережі. Замість централізованої моделі, де дані передаються на віддалені сервери для обробки, вони обробляються на краю мережі, найближчій до джерела даних або до місця їх споживання. Це може бути використано в широкому спектрі сценаріїв, таких як IoT, мобільних додатках, мережах 5G, і т. д.

«Край» у терміні граничних розподілених систем обчислень відноситься до фізичного місця або межі мережі, де відбувається обробка даних [19]. Це може бути фізичним пристроєм, наприклад, мікроконтролером, міні-сервером або іншою обчислювальною річчю, яка розташована ближче до джерела даних або до місця їх споживання.

Граничні розподілені системи обчислень використовують ці фізичні ресурси, розташовані на краю мережі, для обробки та аналізу даних саме на місці їх виникнення або споживання. Це дозволяє зменшити затримки у передачі даних до централізованих серверів для обробки та аналізу, а також може забезпечити більшу конфіденційність та безпеку, оскільки деякі дані можуть бути оброблені локально, не виходячи за межі пристрою.

Як можна помітити, такі системи мають дійсно необмежений потенціал, в них є все: легка масштабованість, зменшення затримок, зменшення обсягу даних і багато чого іншого. Але, не дивлячись на ці переваги, недоліки такої системи досить очевидні. Розпочнемо з того, що такою системо дуже складно керувати. Розгортання та керування великою кількістю різних вузлів на краю може бути складним завданням, особливо в умовах розподіленості та гетерогенності середовища.

Також не слід забувати, що край, як вже говорилось може бути фізичним пристроєм, тому забезпечення безпеки даних та ресурсів на краю може бути складним, оскільки вони можуть бути більш вразливими до зовнішніх загроз. І також потрібно пам’ятати, що ресурси на краю можуть бути обмеженими в порівнянні з централізованою інфраструктурою, що може вплинути на обсяг та складність обробки даних. Приклад такої системи у випадку використання IoT наведено на рисунку 1.4.

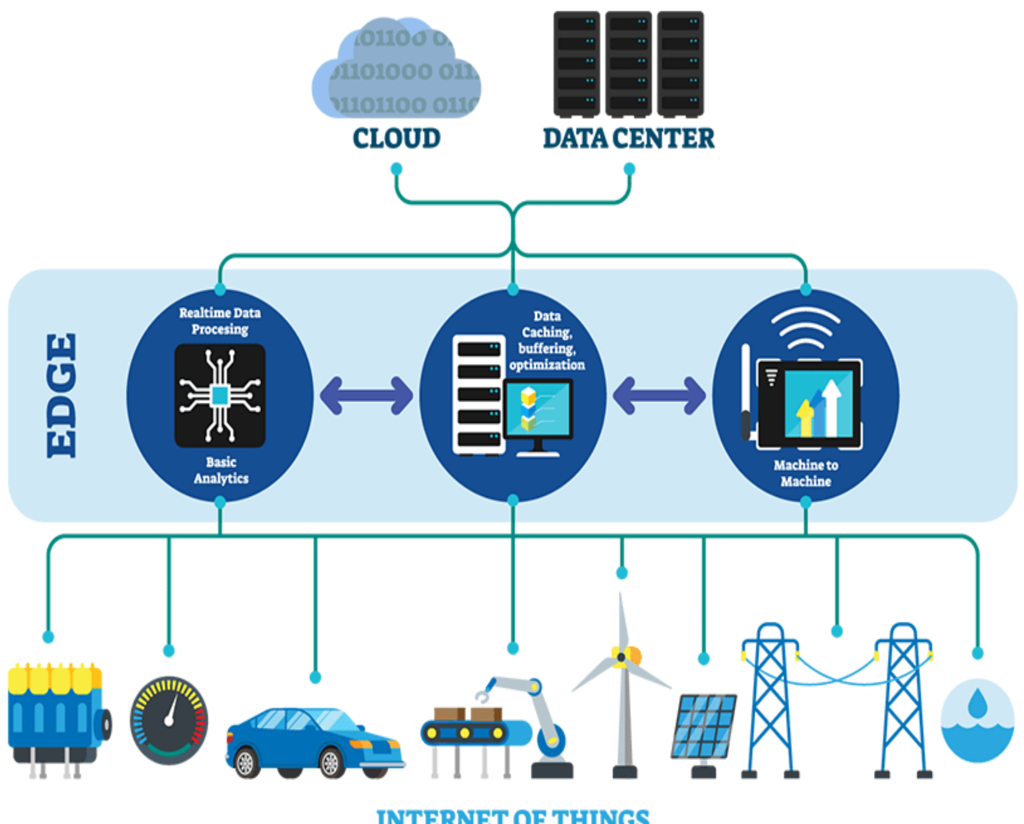


Рисунок 1.4 – Гранична розподілена система обсичлень у зв’язку з IoT

Отже, можна сказати, що вибір яку розподілену систему використовувати залежить від конкретного завдання, яке потрібно виконати. Але ще раз переглянувши всі вищезгадані системи, можна сказати, що граничні розподілені системи виглядають на сьогодні найбільш перспективним варіантом для вивчення, звісно про інші типи систем, також без сумніву не потрібно забувати, та продовжувати розвивати їх.

1.2 Необхідність балансування задач у розподілених системах та аналіз існуючих алгоритмів

Незважаючи на те яка розподілена система буде використовуватися, їх всіх об’єднує одна важлива річ, а саме балансування. В кожній ефективній розподіленій системі обчислень повинно бути реалізовано алгоритм балансування [20-23].

Балансування у розподілених системах є важливим аспектом для забезпечення ефективності, надійності, стійкості цих систем в цілому. Як уже відомо розподілені системи складаються з великої кількості компонентів, які працюють разом для досягнення певної мети. І ось якраз таки балансування такої системи забезпечує, що навантаження розподілятиметься рівномірно між кожним з цих компонентів, і це є ключовим фактором для ефективної та стійкої системи.

Отже, якщо розглянути більш детально необхідність балансування, то воно допомагає уникнути багатьох різних проблем які можуть відбутися з системою, і однією з таких проблем є перенавантаження. Це коли через погано реалізований або не вірний алгоритм балансування відбувається концентрація великої кількості навантаження на одному компоненті, що призводить до його перевантаження, та відповідно знижує його продуктивність тим самим в свою чергу збільшуючи час відповіді для користувача, що є досить критичним при використанні такої системи, наприклад в медичних цілях, адже бувають випадки коли кожна секунда на рахунку. Приклад перенавантаження через несправну роботу балансувальника наведено на рисунку 1.5.

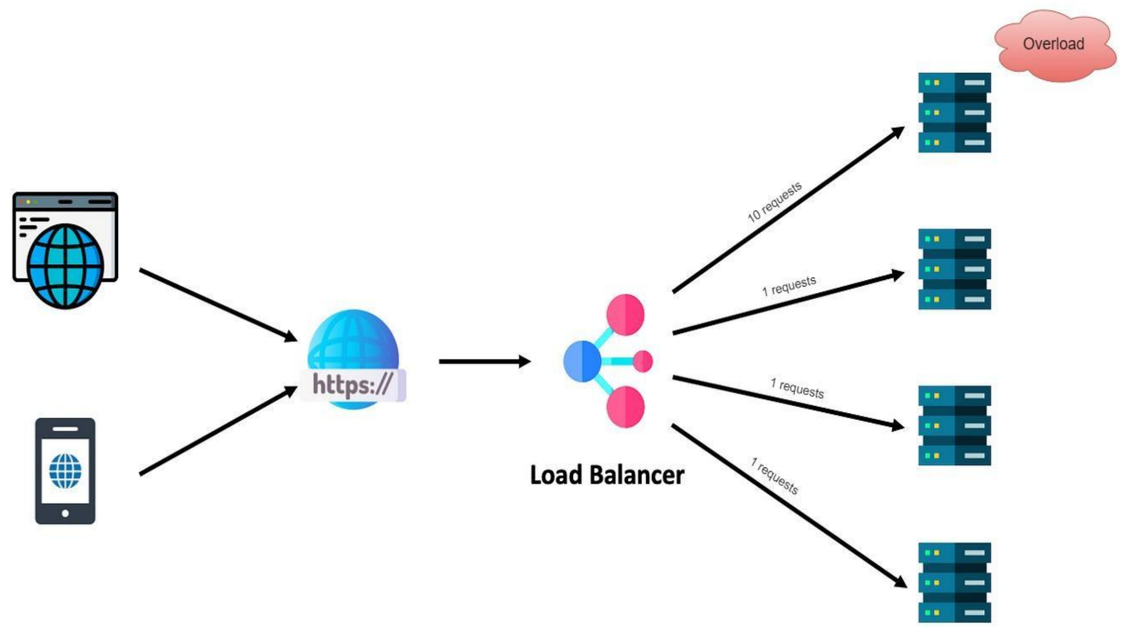


Рисунок 1.5 – Приклад перенавантаження в розподілених системах обчислень

Звісно також не слід забувати й про те що будь-яка розподілена система, повинна бути надійною, тому якщо вийде з ладу один компонент системи, всі інші мають продовжувати працювати. За цей аспект також відповідає балансування системи. Враховуючи алгоритм який реалізовує балансувальник він повинен реагувати на вихід з ладу одного з компонентів по різному. Наприклад якщо алгоритм використовує вагу кожної з систем він повинен формувати чергу відповідно неї, тобто надавати завдання та ставити компоненти у чергу враховуючи їх обчислювальні можливості. Адже якщо система включає в себе багато різних комп'ютерів, то навантаження на кожен комп'ютер повинно бути рівномірне, а для цього комп'ютер який має більше обчислювальних можливостей повинен взяти більше завдань ніж комп'ютер з меншими обчислювальними можливостями. У випадку виходу з ладу компонента такої системи, даний алгоритм повинен перерахувати вагу кожного комп'ютера який залишився у системі.

Слід також згадати про такий аспект розподіленої системи як масштабованість [24]. І для цього питання окрім правильно написаного програмного забезпечення також потрібен балансувальник, який зможе правильно обробляти нові компоненти. Така можливість балансувальника, а саме можливість динамічно обробляти підключення нових компонент дозволяє розподіленій системі дуже легко масштабуватись горизонтально. Горизонтальним масштабуванням називають додавання нових серверів або ж комп'ютерів до системи для розподілу навантаження та забезпечення більшої її ефективності.

Узагальнюючи все вищесказане, балансування у розподілених системах є критичним аспектом для забезпечення їх ефективності, надійності та масштабованості. Процес балансування допомагає оптимізувати використання ресурсів і забезпечити оптимальну продуктивність системи при збільшенні обсягу роботи або зміні умов експлуатації.

Оскільки балансування системи як вже обговорювалося напряму впливає на її ефективність, надійність та масштабованість, то алгоритми балансування навантаження повинні обиратись відповідно до типу системи яку планується розробляти. Існує безліч алгоритмів балансування завдань для розподілених систем обчислень, проте будуть вибрані лише найпопулярніші, описані їх принцип роботи,

розглянуто плюси та мінуси кожного з них. Також буде виділено основні характеристики кожного алгоритму які подібні між собою, таким чином відбуватиметься порівняння даних алгоритмів. Також слід відмітити що розглядатися алгоритми будуть здебільшого в контексті планування процесора, адже майже всі вони застосовуються і для балансування навантаження у мережі.

Розпочнемо з найпопулярнішого алгоритму планування а саме «FCFS». First Come First Serve - найпростіший алгоритм планування процесора, який планує виконання завдання відповідно до часу надходження його у чергу. Алгоритм планування за принципом перший прийшов – перший обслуговувався стверджує, що процес, який першим запитує процесор, отримує його в першу чергу. Він реалізується за допомогою черги. Коли процес потрапляє до черги готовності, його друкована плата з'єднується з хвостом черги. Коли процесор звільняється, він виділяється процесу на початку черги. Процес, що виконується, видаляється з черги. FCFS – це алгоритм планування без витіснення. Переваги даного алгоритму наступні:

- найпростіша і базова форма алгоритму планування процесора;
- простота реалізації: FCFS як дуже простий алгоритм планування, легко реалізувати та зрозуміти. Він не потребує складних обчислень або аналізу для визначення який процес буде запущений наступним;
- він добре підходить для систем, де довші періоди часу для кожного процесу цілком задовільні, адже процеси які потребують багато часу на виконання, можуть почати працювати швидше, ніж в інших алгоритмах;
- мінімальне використання ресурсів планування, даний алгоритм не вимагає значних обчислювальних ресурсів для свого виконання.

Тепер перейдемо до недоліків даного алгоритму. Серед основних недоліків даного алгоритму виділимо лише найбільш значущі, хоча даний алгоритм балансування має право на життя, в розподілених системах він використовується досить рідко, адже він не покриває розгляд деяких критичних факторів розподілених систем [25]. Отож недоліки даного алгоритму наступні:

- високе значення середнього часу очікування. Це один з найбільших недоліків даного алгоритму, і полягає він у тому, що короткі процеси можуть

очікувати поки завершаться довгі процеси, які вже розпочали виконання. Це може призвести до збільшення середнього часу очікування для коротких процесів;

- не можна не відзначити відсутність пріоритетів процесів. Це означає, що критичні або важливі процеси можуть чекати, поки завершаться менш важливі або менш критичні процеси;
- у випадку коли навантаження на систему змінюється, алгоритм може виявитися нестабільним. Якщо, наприклад велика кількість довгих процесів надходять один за одним, то процеси які очікують можуть виконуватися дуже довго.

Отже, хоча First Come First Serve простий для реалізації та зрозумілий, він має досить суттєві обмеження, які можуть призводити до неефективного використання ресурсів та затримок у виконанні процесів.

Наступним буде розглянуто також дуже популярний алгоритм планування – Round Robin, який вже досить часто використовується у балансувальниках для деяких типів розподілених систем, наприклад гомогенних систем, які вирішують однакові завдання. Round Robin – це алгоритм планування процесора, де кожному процесу циклічно призначається фіксований часовий інтервал. Це витісняюча версія алгоритму планування процесора First Come First Served. Як і у випадку з попереднім алгоритмом, даний алгоритм є досить простим у реалізації, але як вже зрозуміло з визначення, він певним чином вирішує одну з основних проблем вищезгаданого алгоритму балансування, а саме проблему зайняття довгим завданням процесорного часу [26]. Звісно як і всі алгоритми RR також має свої переваги та недоліки. До переваг слід віднести наступні:

- у порівнянні з іншими алгоритмами, які можуть надавати перевагу довгим процесам, RR зменшує час очікування для коротких процесів, оскільки кожен процес отримує свою порцію обслуговування протягом фіксованого періоду;
- рівномірне розподілення часу. RR забезпечує розподілення часу обслуговування між різними процесами. Кожен процес отримує однаковий квант часу для виконання, що дозволяє вирівняти навантаження на систему;

– даний алгоритм не потребує складного управління або моніторингу стану процесів. Це робить його ідеальним для швидкої реалізації та використанні в системах з обмеженими ресурсами;

– хоча RR не є справжньою паралельною обробкою, він може ефективно керувати кількома процесами, що дозволяє операційним системам обробляти багатозадачність та задовольняти потреби користувачів, які вимагають одночасного виконання декількох завдань.

Не дивлячись на те що даний алгоритм є дуже популярним та має на перший погляд дуже багато переваг, недоліки в ньому також наявні. Серед основних недоліків алгоритму RR можна виділити наступні:

– втрата продуктивності через перемикання контексту. RR використовує часті перемикання контексту між процесами, що може призвести до зниження загальної продуктивності системи, особливо якщо час перемикання контексту стає значною частиною загального часу виконання;

– недооцінка реальних потреб процесів. RR може недооцінювати реальні потреби деяких процесів. Наприклад, якщо один процес потребує більше часу процесора для обробки важливої задачі, йому може бути виділено недостатньо часу перед перемиканням до наступного процесу;

– неефективність для деяких типів завдань. Деякі типи завдань можуть бути неефективними для RR. Наприклад, задачі з великою кількістю вводу-виводу можуть бути краще вирішені іншими алгоритмами планування, такими як алгоритм планування з пріоритетом або алгоритм з однорідним квантом часу.

Узагальнюючи, RR є простим та рівномірним алгоритмом балансування навантаження, який може бути ефективним для деяких сценаріїв використання, але він також має свої обмеження та недоліки.

Тепер від більш простих та базових алгоритмів, переходимо до алгоритму балансування, який ще частіше використовується у розподілених системах ніж вищезгадані алгоритми, а саме до Weighted Round Robin. Як видно з назви WRR є розширенням звичайного Round Robin, різниця полягає в тому, що в даному алгоритмі кожному процесу або завданню надається вага, що визначає його

пріоритет або важливість. Оскільки цей алгоритм є досить поширеним, буде розглянуто кожну з його переваг та недоліків більш детально.

Після аналізу базового визначення даного алгоритму можна дійти висновку що він дає можливість використовувати так звану вагу кожного елемента. В контексті розподіленої системи вагою називають пріоритетність або важливість елементів таких як вузли, сервери, процеси або ресурси [27]. Вага може використовуватись для розподілу навантаження, ресурсів або обчислювального часу між різними частинами системи. Наприклад вага вузла у розподілених обчислювальних системах вказує на його потужність або доступні ресурси. Використовуючи ці ваги, система може розподіляти завдання або обчислення таким чином, щоб вони були більш рівномірно розподілені між усіма вузлами. У веб-серверах або серверних кластерах вага може використовуватися для розподілу трафіку між серверами. Сервери з вищими вагами можуть отримувати більше запитів або навантаження, щоб забезпечити кращу продуктивність або виконати важливіші завдання. Отож, коли розглядається даний алгоритм, в першу чергу потрібно використати його можливість підтримки пріоритетів, з якої вже можна розкрити більше позитивних сторін даного алгоритму.

Як не дивно використання пріоритетів дає також можливість більш ефективного використання ресурсів. WRR дозволяє ефективно використовувати ресурси, надаючи більше часу тим процесам, які потребують більше обчислювальних ресурсів. Саме цього так не вистачало простому алгоритму RR – можливості надання важливішим процесам більше часу на обробку. Також слід відзначити, що вага це лише число, а яку характеристику системи ним описувати вирішує розробник, тому змінити вагу означає просто змінити число. Це також означає, що ваги можуть бути змінені в реальному часі в залежності від потреб системи [28]. Крім того це надає гнучкість в управлінні процесами та ресурсами, що без сумніву є ще одним позитивним аспектом даного алгоритму.

Не дивлячись на те що даний алгоритм є розширенням алгоритму RR, у реалізації він є не настільки простим. І діло не в самому написанні алгоритму, складність в визначені правильної ваги для кожного процесу. Налаштування ваг для кожного процесу може бути складним завданням, особливо в системах з

великою кількістю цих самих процесів. Також розрахунок та управління вагами може призвести до додаткових обчислень та витрат процесорного часу, але це є необхідним якщо в системі використовуватиметься даний алгоритм. Повертаючись до питання налаштування ваги, слід запам'ятати, якщо ваги неправильно налаштувати це може призвести до нестабільності пріоритетів та виконання процесів, що може вплинути на продуктивність системи, і зробити результати її виконання непередбачуваними.

Окремо слід виділити проблему голодування в системах, які використовують даний алгоритм. Якщо один процес має надто велику вагу, це може призвести до голодування інших процесів, які мають менші ваги. Таким чином втрачається ефективність нашої системи.

Підбиваючи підсумки на рахунок даного алгоритму балансування навантаження, можна сказати, що в цілому WRR може бути ефективним алгоритмом для систем, де потрібно надавати пріоритети різним процесам або задачам, і це дійсно так, адже даний алгоритм використовують багато розподілених систем, які працюють над найрізноманітнішими завданнями. Але він також має свої обмеження та потенційні недоліки, але в основному всі вони зв'язані з ризиками при налаштуванні ваги, та при витрачені процесорного часу на її розрахунок.

Всього було розглянуто три алгоритми балансування, звісно їх існує значно більше, але багато наступних алгоритмів, певним чином повторюють логіку попередніх. Тому на завершення буде розглянуто ще лише один алгоритм балансування, який називається Least Response Time. Це алгоритм балансування навантаження, який вибирає клієнта на основі найменшого часу відповіді на попередні запити. Головна ідея LRT полягає в тому, щоб направляти нові завдання до того клієнта, який найшвидше обробляє запити, з метою оптимізації часу відповіді для кінцевих користувачів. Тут вже можна побачити певний пріоритет, тобто у даному алгоритмі спостерігається відбиток WRR. Дійсно, насправді даний алгоритм можна розглянути як спеціалізований випадок WRR, де вагою є швидкість відповіді на попередні запити.

В основному даний алгоритм є також досить непоганим рішенням для деяких типів розподілених систем, а недоліками даного алгоритму є моніторинг часу та

чутливість до змін. Для ефективної роботи LRT потрібно відстежувати час відповіді кожного клієнта, що може вимагати додаткових ресурсів та обробки даних. Також зміни у часі відповіді клієнтів можуть призвести до нестабільності системи та непередбачуваності вибору клієнта для нових завдань.

Далі буде наведено таблицю 2.1, яка дозволяє порівняти алгоритми балансування за додатковими характеристиками.

Таблиця 2.1 – Порівняння алгоритмів балансування

Характеристика	FCFS	RR	WRR	LRT
Наявність пріоритетів	Ні	Ні	Так	Так
Час очікування	Великий	Помірний	Помірний	Помірний
Пропускна здатність	Середня	Середня	Середня	Середня
Рівномірність навантаження	Низька	Висока	Висока	Висока
Складність алгоритму	Простий	Простий	Складний	Складний
Стійкість до перевантажень	Низька	Середня	Висока	Висока
Підтримка різних типів завдань	Середня	Середня	Висока	Висока
Адаптивність до змінних умов	Низька	Середня	Висока	Висока

Висновки

У розділі було розглянуто декілька алгоритмів балансування навантаження, таких як FCFS (First-Come, First-Served), Round Robin, Weighted Round Robin

(WRR), та Least Response Time (LRT). Основними критеріями оцінки цих алгоритмів були наявність пріоритетів, час очікування, пропускна здатність, рівномірність навантаження, складність алгоритму, стійкість до перевантажень та адаптивність до змінних умов. З проведеного аналізу вищезгаданих алгоритмів також стає очевидним, що вибір алгоритму повинен бути тісно пов'язаний з типом розподіленої системи та її характеристиками. Це важливо для забезпечення оптимальної продуктивності та ефективності системи, оскільки кожен алгоритм має свої переваги та недоліки залежно від конкретного контексту застосування.

Наприклад, FCFS простий у реалізації, але може спричиняти високий час очікування для коротких процесів. У цьому алгоритмі завдання обробляються у порядку їх надходження, що робить його прозорим і легким для розуміння, але він не враховує пріоритети завдань та може бути неефективним у середовищах з неоднорідними навантаженнями. Round Robin, в свою чергу, розподіляє завдання циклічно між усіма доступними ресурсами. Це забезпечує рівномірний розподіл навантаження, але не враховує різницю в продуктивності різних вузлів, що може призвести до перевантаження менш потужних ресурсів. Round Robin підходить для систем з однорідними ресурсами, де кожен вузол має схожу продуктивність.

З іншого боку, WRR та LRT пропонують кращу продуктивність за рахунок більш складної реалізації. Однією з ключових переваг WRR є його здатність враховувати пріоритети процесів, що дозволяє забезпечити більш рівномірний розподіл навантаження між вузлами системи. Завдяки цьому, WRR краще підходить для систем з неоднорідними обчислювальними ресурсами, оскільки він може динамічно регулювати навантаження, враховуючи потужність та продуктивність кожного вузла. Це дозволяє максимізувати продуктивність системи та мінімізувати час очікування для критичних процесів.

2 ОБҐРУНТУВАННЯ ВИБОРУ ПРОГРАМНИХ ЗАСОБІВ

2.1 Вибір мови програмування для реалізації серверної частини

Для виконання даного диплому було розглянуто декілька мов програмування, а саме: C# та JavaScript. Звісно, так як розподілена система потребує сервер який буде приймати та розподіляти завдання між обчислювальними приладами то ці мови розглядалися з їх серверними можливостями, тому говорячи про JavaScript мається на увазі платформа NodeJS, яка дає змогу JavaScript взаємодіяти з операційною системою.

Спершу було виділено основні позитивні сторони C#. Однією з ключових сторін мови програмування C# при розробці серверних додатків є її висока продуктивність і ефективність. C# підтримує різноманітні парадигми програмування, включаючи об'єктно-орієнтоване програмування та асинхронне програмування, що дозволяє розробникам створювати потужні та масштабовані серверні застосунки. Мова має багато функцій, таких як обробка винятків, автоматичне управління пам'яттю та розширені можливості управління типами, що сприяє зручності та безпеці кодування.

Що до NodeJS, то він зі своєї сторони надає швидку реакцію на події завдяки своїй асинхронності, що дозволяє оптимізувати обробку багатьох одночасних з'єднань [29]. Його легка масштабованість дозволяє також будувати великі та ефективні серверні додатки з високою продуктивністю. Node.js в придачу виявився чудовим вибором для веб-додатків з інтенсивним обміном даними завдяки можливості роботи з WebSocket і великою кількістю сторонніх модулів. Його менеджер пакетів, зокрема npm, пропонує широкий вибір модулів, що спрощує розробку та розширення функціональності серверних додатків. Відкритість та активна спільнота сприяють швидкому розвитку та вирішенню проблем, що виникають під час розробки.

З першої точки зору ці дві мови можуть показатися однаково привабливими для розробки серверу для розподіленої системи, але все ж таки вибір пав на NodeJS. Саме NodeJS за останніми новинами зробив величезний крок у розвитку свого

середовища шляхом покращення продуктивності та швидкодії своїх додатків. Завдяки постійній оптимізації та вдосконаленню платформи, Node.js став більш продуктивним і швидшим, що не скажеш про C#. Хоча він також розвивається, розробники даної мови більшу увагу приділяють на саме розширення функціоналу їхньої платформи .NET, добавляючи в неї можливості для створення клієнтської частини сайтів, та багато інших цікавих речей.

Повертаючись до NodeJS, окрім збільшення продуктивності, ця платформа чудово підтримує технологію, через яку і буде здійснюватись взаємодія між клієнтом і сервером, а саме технологію WebSocket. Хоча NodeJS ще тільки збираються додати можливості WebSocket у свій загальний пакет, через велику підтримку зі сторони користувачів, існують багато сторонніх пакетів, які допомагають працювати з ними.

Для кращого розуміння як працює NodeJS також було розглянуто його внутрішню будову, але більше уваги приділялося Event Loop. Реалізація Event Loop (циклу подій) є ключовою особливістю платформи NodeJS, яка забезпечує його асинхронну, неблокуючу поведінку. Сама внутрішня будова NodeJS наведена на рисунку 2.1.

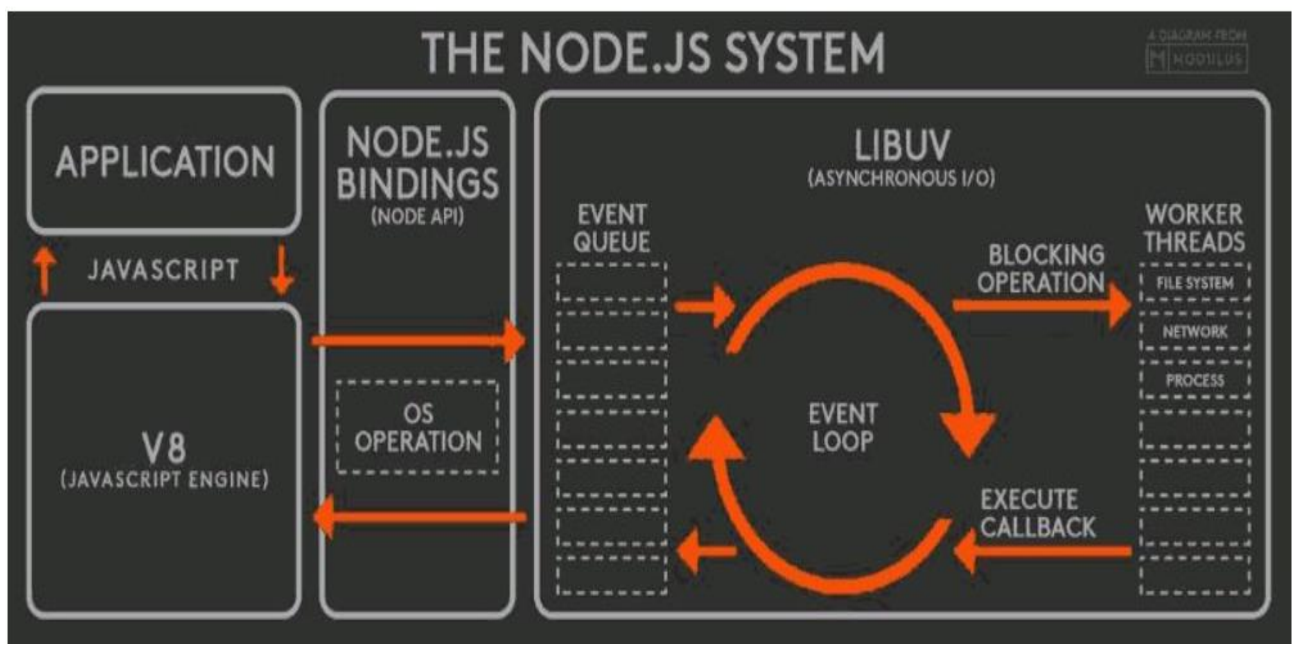


Рисунок 2.1 – Внутрішня будова NodeJS

Як видно з рисунка сама реалізація Event Loop знаходиться в бібліотеці libuv, яка разом з рушієм V8 є основою NodeJS. Node не працює за принципом виділення окремого потоку для кожного запиту, натомість, весь JavaScript код виконується в єдиному потоці, в якому опрацьовується цикл подій [30]. Ця особливість є як недоліком так і перевагою даної платформи, оскільки при роботі з дуже багатьма потоками в інших мовах, втрачається більшість часу на переключення контекстів, в випадку ж NodeJS такого не відбувається адже в основному розробник працює з одним головним потоком. Але мінус в такому підході також є, оскільки є лише один потік, то операції які займають багато часу будуть блокувати його не даючи можливості іншим функціям продовжувати свою роботу. Але для деяких модулів NodeJS потоки все ж таки створюються, бібліотека libuv створює пул потоків для виконання асинхронних операцій, які використовуються тільки чотирма вбудованими частинами NodeJS, а саме модулями fs, crypto, zlib та для DNS-пошуку. Всі інші операції використовують основний потік, в якому опрацьовується цикл подій. При старті, Node ініціалізує Event Loop, виконує початковий код, що запускає таймери, робить запити і т.д, а тоді переходить до опрацювання циклу подій.

Для повного розуміння Event Loop потрібно зрозуміти як і для чого працюють всі його фази, а всього їх є 7, але дві з них не виконують ніяких явних функцій на цих фазах виконуються внутрішні операції libuv. Кожна з фаз представляє собою FIFO-чергу колбеків, коли Event Loop переходить в конкретну фазу, він виконує операції пов’язані з нею та колбеки в черзі, поки вони не закінчаться [31].

Першою йде фаза опитування, вона ж Poll. Фаза опитування виконує виклики, пов'язані з введенням/виведенням. Це фаза, в якій код програми найчастіше виконується. Коли основний код починає працювати, він виконується у цій фазі.

Далі йде фаза перевірки або Check, на якій перевіряється чи є у черзі зворотні виклики які були оголошені через setImmediate(), це функція, які присутня тільки в NodeJS. Вона і відповідає за перенесення виконання функції в цю фазу, а робиться це в основному для того, щоб розгрузити основний потік.

Наступною йде фаза Close. Назва цієї фази все про себе говорить сама, на цій фазі виконуються зворотні виклики через EventEmitter, які підписані на подію close. Наприклад, коли TCP-сервер net.Server закривається, він видає подію close, яка запускає зворотній виклик на цій фазі.

Передостанньої йде фаза Timers, знову ж таки назва говорить за себе. На цій фазі виконуються зворотні виклики заплановані за допомогою setTimeout() та setInterval().

І остання фаза називається Pending, вона ж фаза очікування. На цій фазі виконуються спеціальні системні події, наприклад коли TCP-сокет повертає помилку, вона повернеться на цій фазі.

Слід додати один важливий момент, асинхронний код в NodeJS також планується за допомогою Promise. Promise поміщає обробник функції в чергу мікрозадач, яка виконується в кінці кожної з вищезгаданих фаз. Тобто щоб не блокувати головний потік, можна планувати обробку результатів функції в кінці виконання фази, адже сама функція виконується асинхронно разом з нашим головним кодом [32]. Загальний порядок виконання фаз зображено на рисунку 2.2.

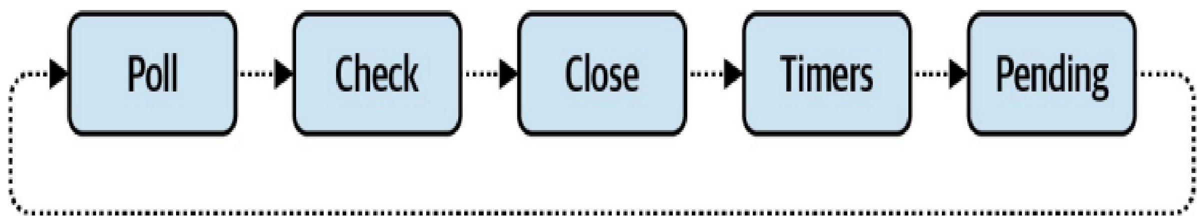


Рисунок 2.2 – Порядок виконання фаз Event Loop

Слід додати що у вищезгаданих функціях для роботи з асинхронним кодом не було згадано таку функцію як process.nextTick(). Функція process.nextTick() у Node.js є однією з основних особливостей для управління асинхронними викликами. Однак, при розробці комплексних асинхронних додатків, особливо у контексті розподілених систем, її застосування потребує обережності та розуміння її впливу на процес виконання програми.

На відміну від інших методів асинхронного планування, таких як setTimeout() чи setImmediate(), які дозволяють відкласти виконання до наступної фази подій або

до наступного циклу подій відповідно, process.nextTick() планує виконання функції негайно після завершення поточного операційного стеку, але перед будь-якими подальшими асинхронними подіями. Це означає, що виклики process.nextTick() можуть призвести до рекурсії, яка не переривається подіями I/O, таймерами або іншими системними викликами. Якщо така рекурсія є необмеженою, вона може призвести до переповнення стеку викликів та зрештою до зависання програми [33].

Ця особливість робить process.nextTick() вкрай потужним інструментом у випадках, коли потрібно швидко відреагувати на подію без затримок, що вносяться іншими методами асинхронного планування. Проте, в контексті розподілених систем, де завдання та процеси можуть бути розподілені серед різних вузлів і потребувати взаємодії між ними, необхідно забезпечити, що ці взаємодії не призведуть до надмірних затримок або блокування. У таких випадках використання process.nextTick() може бути недоречним, оскільки це може надмірно ускладнити управління потоком подій та асинхронним виконанням.

Крім того, важливо розуміти, що process.nextTick() має вищий пріоритет у порівнянні з setImmediate(), тому задачі заплановані за допомогою process.nextTick(), будуть виконані до будь-яких setImmediate() викликів в одному та тому ж циклі подій. Це також вносить ризики переповнення стеку, особливо при неправильному або надмірному використанні. Посилаючись на все вищесказане було прийнято рішення не використовувати функцію process.nextTick() під час розробки розподіленої системи та балансувальника.

Звісно слід згадати і про середовище розробки, в даному випадку яким є Visual Studio Code. Під час роботи використовувався саме Visual Studio Code, тому що він є одним з найбільш впливових та популярних редакторів коду в сучасному світі розробки програмного забезпечення. Він відкритий, безкоштовний та розроблений командою Microsoft. Він підтримує майже всі основні мови програмування. Visual Studio Code працює дуже швидко і забезпечує високу продуктивність навіть при роботі з великими проектами. Це дозволило ефективно працювати над своїм завданням без затримок. Також слід відмітити те що він крос-платформний, Visual Studio Code підтримується на всіх основних операційних системах таких ях Windows, macOS та Linux. Програма Visual Studio Code має

велику кількість налаштувань зовнішнього вигляду редактора коду. Користувач може налаштувати тему оформлення програми, розмір вікон, видимість відкритих панелей інструментів, а також розмір та тип шрифтів. Узагальнюючи можна сказати, що Visual Studio Code - це потужний, зручний та розширюваний редактор коду, який допомагає розробникам бути більш продуктивними та ефективними у своїй роботі.

2.2 Вибір технологій для розробки засобу моніторингу

Моніторинг роботи розподіленої системи це також важлива складова всієї системи загалом, яка допомагає в подальшій розробці та в виправлені помилок, тим самим заощаджуючи багато часу. Оскільки для написання серверної частини використовується NodeJS, то було б непогано використати цю саму мову програмування для написання додатку який буде слідкувати за станом нашої системи, наприклад переглядом логів, відображенням графіків навантаженості клієнтів та відображенням загальної статистики системи. Саме по цій причині було вирішено обрати такий фреймворк як Electron.

Electron - це фреймворк для розробки настільних застосунків із використанням HTML, CSS і JavaScript. У двійковий код Electron уже вбудовані Chromium і NodeJS, і це дає змогу мені підтримувати лише JavaScript код і створювати крос-платформні додатки, що працюватимуть як на Windows, так і на macOS та Linux. Але оскільки Electron дає змогу лише запускати браузер та відображати в ньому html сторінку під видом настільного додатку, він не дає ніяких додаткових можливостей по створені самих html сторінок. Тому окрім Electron, було також вирішено обрати одну з найбільш популярних бібліотек для роботи з UI, а саме React.

React - це передовий інструмент для створення веб-інтерфейсів, який наділяє розробників можливістю створювати складні та інтерактивні додатки шляхом розбиття їх на повторно використовувані компоненти. Використання віртуального DOM сприяє оптимізації рендерингу, забезпечуючи швидкість та продуктивність. Однак, React є досить складним для вивчення через необхідність освоїти сучасні

підходи та концепції, такі як JSX та управління станами компонентів. Хоча, при розумінні вищезгаданих складнощів, React пропонує неймовірно гнучку, бистру та потужну роботу з класичним набором css та html. React дає змогу динамічно міняти html сторінку, тим самим пропонуючи так званий Single Page Application [34]. Тобто на відміну від класичної роботи з html, коли на кожен запит клієнт отримував нову сторінку, завантажував нові скрипти та стилі, React дає змогу повертати клієнту одну й ту ж саму сторінку зі внесеними в неї змінами. Тим самим лише перше завантаження сторінки буде займати певний час, а всі подальші зміни будуть відбуватися майже миттєво без перезавантаження сторінки, що звісно позитивно впливає на клієнтів. Окремо слід виділити роботу з так званими хуками у React. Хуки в React надають безліч можливостей, включаючи якраз таки повідомлення ядру, де саме відбулась зміна в документі. Також хуки дають можливість виконувати побічні ефекти.

Побічний ефект React є важливим компонентом будь-якого React-додатку. Він допомагає встановити зв'язок із зовнішніми ресурсами, такими як локальне сховище, та підтримує використання API. Тобто саме хук надає змогу звернутися до якогось віддаленого серверу за даними при певних подіях, таких як перше завантаження сторінки, що і використовувалося під час розробки додатку. Тому з всього вищесказаного можу сказати, що роботі з React було приділено досить багато часу. Звісно люди які вже працювали з React напевно знають що використовувати його без менеджера санів дуже незручно. Адже компонент це окрема функція, яка повертає html розмітку, а зберігати всі дані на рівні компонента та передавати їх в дочірні компоненти, дуже незручно та призводить до багатьох помилок при роботі з React. Тому як менеджер станів було вирішено обрати бібліотеку Zustand.

Як вже говорилось Zustand - це бібліотека для управління станом в додатках React. Вона надає простий та зручний спосіб зберігати та оновлювати стан додатка, використовуючи найновіші можливості JavaScript, такі як хуки. За допомогою Zustand можна легко створювати ізольовані контейнери стану, що дозволяє зменшити складність та підвищити читабельність коду. Крім того, Zustand підтримує асинхронні операції та має невеликий розмір, що дійсно є дуже

важливим моментом. У порівнянні з іншими менеджерами управління станом Zustand дає змогу працювати з асинхронними операціями використовуючи найновіші можливості самого JavaScript, без необхідності підключення додаткових пакетів. Також Zustand надає змогу створювати безліч незалежних один від одного сховищ, що набагато покращує вигляд та читання коду.

До прикладу найпопулярніший менеджер станів на сьогодні який називається Redux не може запропонувати нічого з вищесказаного. Окрім великого розміру та концепції єдиного сховища, Redux для роботи з асинхронністю змушує нас встановлювати додаткові пакети такі як Redux-Thunk. Звісно Redux має свої позитивні сторони, але його популярність зв’язана з тим, що він на ринку вже досить довго, і люди які працювали з ним роками, просто не мають бажання розглядати нові технології які виконують ідентичне завдання, навіть якщо вони роблять це простіше.

Хоча у кожного менеджера станів є свій функціонал та синтаксис, принцип їх роботи не відрізняється. І тому в них є одна спільна річ, а саме робота з незмінними даними. Тобто всі дані в сховищі повинні бути immutable, або ж простіше, незмінюваними [35]. Коли менеджер станів змінює дані в сховищі він повинен повідомити про це React, після чого другий в свою чергу відобразить ці зміни шляхом зміни стану додатку. Але проблема в тому, що ядро React під час рендерингу сторінки порівнює попередній та поточний стан даних за допомогою оператора глибокого порівняння (shallow comparison). Це означає, що React перевіряє, чи змінилися посилання на об'єкти або масиви даних, а не їх вміст або значення. Тобто при передачі даних в сховище повинно завжди передаватися нове посилання, що інколи досить важко при роботі з великими об’єктами даних, адже якщо потрібно лише змінити одне поле з тридцяти доступних, потрібно спочатку скопіювати решту полів в новий об’єкт замінити потрібне, та передати в сховище. Тому, щоб полегшити роботу з незмінюваними даними, використовуватиметься пакет Immer.

Immer - крихітний пакет, який дозволяє працювати з незмінним станом у більш зручний спосіб. Тобто він гарантує те, що при передачі об’єкта даних, на виході завжди отримуємо його повну копію з новим посиланням, незалежно від

того що було зроблено з даним об'єктом в ході роботи. Якщо розглядати алгоритм роботи Immer, він просто отримує початковий об'єкт, копіює його, та виконує над ним певні дії які передаються в нього як callback. Після чого Immer просто повертає повий об'єкт з новим посилання, який вже можна передати в сховище. В загальному можна сказати, що Immer позбавляє рутинної роботи, яка просто забирає час.

2.3 Засоби для організації комунікації в системі

Для того щоб сервер міг приймати підключення від багатьох клієнтів, та обмінюватися з ними повідомленнями було обрано технологію WebSocket.

На відміну від TCP або HTTP де з'єднання зазвичай ініціюється клієнтом, WebSocket дозволяє ініціювати та утримувати з'єднання з обох сторін. Таким чином, обидва клієнт та сервер можуть починати спілкування в будь-який момент, що робить його більш гнучким і підходящим для сценаріїв реального часу та багаторівневих комунікацій [36-41]. А оскільки наш сервер буде обслуговувати розподілену систему, та буде виступати як балансувальник навантаження, то миттєва та двостороння комунікація між клієнтом та сервером, ідеально для цього підходять.

Окрім роботи в реальному часі та двостороннього обміну даними, WebSocket також має низьку затримку та навантаження на мережу. Знову ж таки якщо порівнювати з HTTP, WebSocket має меншу затримку оскільки він не потребує повторних запитів та відповідей, що якраз таки випливає з його роботи двостороннього з'єднання. Це ще одна з причин вибору даного виду комунікації, адже розподіленій системі якраз таки важлива швидкодія та мінімізація навантаження на мережу.

Також не зважаючи на всі явні переваги WebSocket, є ще один дуже важливий момент який привертає увагу, а саме зручність використання. WebSocket дозволяє передавати як структуровані, так і неструктуровані дані у реальному часі без необхідності створення нового з'єднання для кожного запиту. Окрім цього всі дані передаються події, а сам WebSocket побудований на подійно орієнтованій моделі,

тобто для передачі та прийому даних використовується такий клас як EventEmmiter. За допомогою EventEmitter можна створювати власні об'єкти, які можуть генерувати події, а також підписуватися на ці події та реагувати на них. Це робить код більш зрозумілим та підтримуваним, особливо в контексті реалізації алгоритмів навантаження для розподілених систем.

Однак як вже було сказано WebSocket не підтримується в NodeJS як загальний модуль, однак завдяки великій підтримці користувачів, знайти сторонній пакет було не складно, загалом на майже всіх формах та статтях рекомендували пакет Socket.IO, тому було вирішено звернути увагу в першу чергу на нього.

Socket.IO – це бібліотека обміну повідомленнями в реальному часі для розробників JavaScript, яка базується на WebSocket. З визначення даної бібліотеки вже можна дійти висновку, що Socket.IO лише базується на WebSocket, а не є їх реалізацією, хоча всередині вона використовує цю технологію. Річ у тому, що Socket.IO при підключенні та запитах прикріплює додаткові метадані, яких в класичних WebSocket відсутні. З одного боку, це може виглядати як недолік, адже клієнт WebSocket не зможе успішно підключитися до сервера Socket.IO, і клієнт Socket.IO також не зможе підключитися до простого сервера WebSocket. Проте, якщо архітектура комунікації розробляється з нуля, і розробник впевнений що клієнт буде написаний з використанням Socket.IO, це надасть багато додаткових можливостей і пришвидшить розробку.

Однією з суттєвих переваг Socket.IO над іншими пакетами є його багатомовність. Socket.IO підтримує багато мов програмування, що надає розробникам значну гнучкість у реалізації проектів. Серед усіх доступних мов, на яких може бути написаний клієнт Socket.IO, слід виділити такі: C++, Java, NodeJS, JavaScript, Python, Kotlin, PHP, C#. Така різноманітність підтримуваних мов програмування робить Socket.IO універсальним інструментом для розробки додатків реального часу на різних платформах і в різних середовищах.

Крім основних можливостей WebSocket, дана бібліотека пропонує багато додаткових функцій. Наприклад, якщо з’єднання втрачено, Socket.IO намагається автоматично відновити його. Таким чином, використовуючи лише Socket.IO,

розробник отримує механізм автоматичного перепідключення. Це значно підвищує надійність і стійкість додатків до збоїв мережі.

Socket.IO також додає буферизацію пакетів, що дозволяє зберігати та обробляти їх у відповідному порядку. У багатьох випадках це є дуже важливим, оскільки забезпечує коректність передачі даних і послідовність виконання операцій. Завдяки буферизації пакетів, розробники можуть бути впевнені, що всі повідомлення будуть доставлені та оброблені в тому порядку, в якому вони були надіслані, що є критичним для багатьох застосувань, де правильна послідовність даних має велике значення.

Іншою важливою функцією, яка привертає увагу при огляді цієї бібліотеки, є мультиплексування або ж простір імен. Socket.IO дозволяє створювати простори імен, які дозволяють розділити з'єднання на різні категорії. Це особливо корисно для організації логічної структури додатку і зменшення складності коду. Оскільки потрібно було розробити додаток для моніторингу системи, можливість використання просторів імен виявилася дуже корисною. Це дозволило розбити програму на дві частини: одна відповідала за загальні підключення клієнтів та виконувала функції балансування, а інша частина відповідала за роботу додатку моніторингу системи. Таким чином, дві частини не перетиналися в одному коді, що значно спрощує розробку і підтримку додатку. Єдині функції, які з'єднували ці частини, забезпечували взаємодію між ними, що підвищувало модульність і масштабованість системи.

Дивлячись на всі ці можливості та несуттєві в даному випадку недоліки, було прийнято рішення використовувати саме Socket.IO як бібліотеку для роботи з WebSocket. Це рішення дозволило скористатися всіма перевагами даної технології, забезпечуючи надійну, ефективну і гнучку платформу для розробки додатків реального часу. У результаті, використання Socket.IO сприяло підвищенню продуктивності розробки та якості кінцевого продукту, що є важливим фактором при створенні надійних систем.

2.4 Захист передачі даних та їх зберігання

Оскільки система повинна виконувати завдання від клієнтів та повертати їм результати, можливо ці завдання будуть критично важливі для деяких системи, і тому щоб запобігти крадіжці цих даних було прийнято рішення впровадити шифрування.

Як алгоритм шифрування використовувався Advanced Encryption Standard. Advanced Encryption Standard (AES) є одним із найважливіших інструментів у сфері кібербезпеки, що використовується для захисту конфіденційної інформації. Він розроблений як симетричний блоковий шифр і є стандартом, що вибрано урядом США для захисту важливих даних. Його універсальність дозволяє використовувати AES як у програмному, так і в апаратному забезпеченні на глобальному рівні [42-45]. Однією з ключових переваг AES є його здатність розбивати повідомлення на блоки по 128 біт і проводити декілька раундів шифрування, що робить його значно безпечнішим від старіших методів.

В загальному список переваг даного алгоритму шифрування досить великий і включає в себе наступні пункти:

- стійкість до відомих текстів. AES відповідає на відомі текстові атаки, що робить його надійним для захисту даних;
- стійкість до диференціального та лінійного криптоаналізу. AES виявляється міцним проти цих атак;
- активний аналіз та перевірка криптографічною спільнотою. Даний алгоритм пройшов інтенсивний аналіз та перевірку, що підтверджує його надійність.

Звісно слід також назвати одну з найголовніших перевага даного алгоритму шифрування - сама ж швидкість шифрування. AES вирізняється не тільки своєю безпекою, але й швидкістю шифрування. Це критично важливо для ефективної роботи розподілених систем, де обробка великих обсягів даних має відбуватися швидко. Висока швидкість шифрування і розшифрування, що забезпечується AES, дозволяє уникнути зайвих затримок у передачі та обробці інформації. Це лише частина з переваг які надає даний алгоритм шифрування.

І ще однією перевагою та однією з головних причин вибору даного алгоритму є те що він присутній в загальному пакеті NodeJS, і для його використання достатньо лише підключити один з модулів Node.

Інтеграція AES з різними технологіями, такими як NodeJS, є однією з його ключових переваг. Використання модулю crypto у NodeJS забезпечує легкість у реалізації шифрування у різних додатках. Підтримка AES багатьма мовами програмування та криптографічними бібліотеками сприяє його широкому впровадженню та використанню у різноманітних інформаційних системах. Можливість розширення системи з використанням додаткових модулів, написаних на C++ або інших мовах, також надає гнучкості при розробці складних і багатофункціональних систем.

На рисунку 2.3 наведено просту схему шифрування за допомогою алгоритму AES.

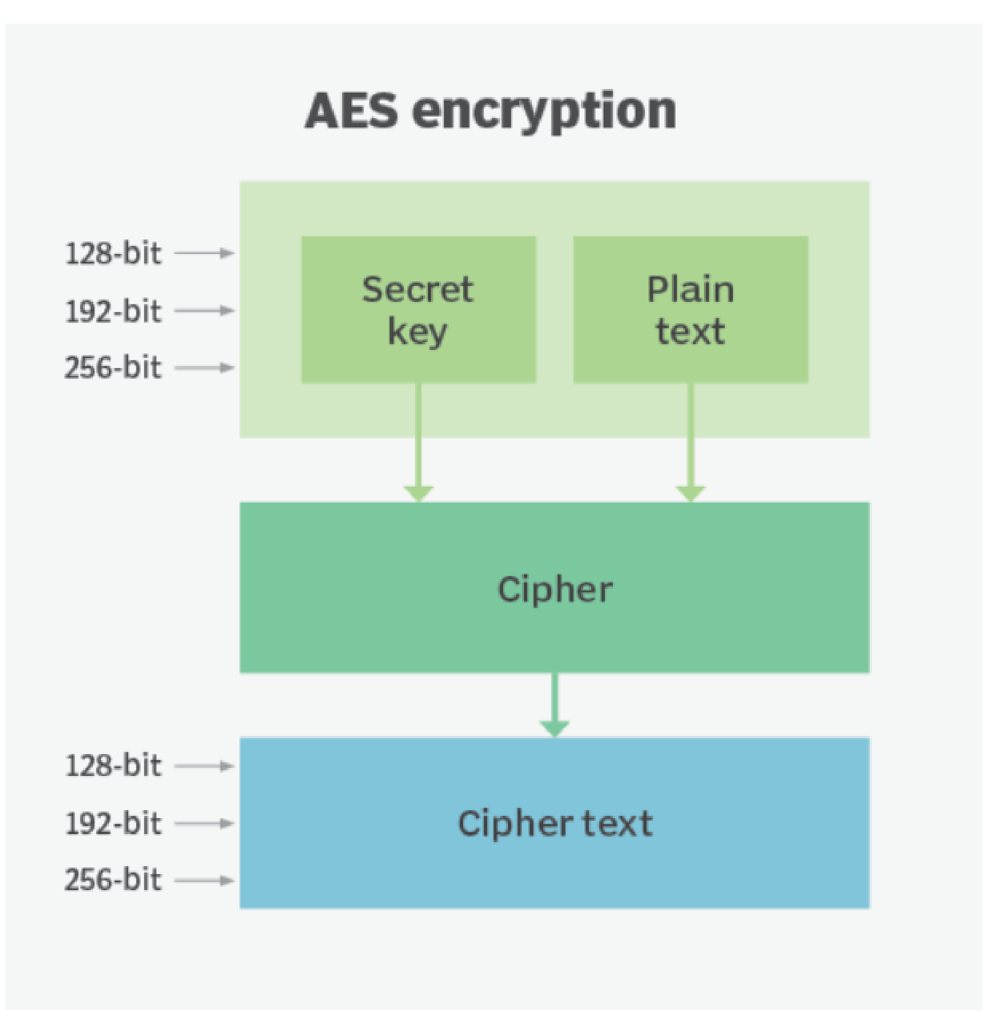


Рисунок 2.3 – Схема шифрування алгоритмом AES

Окрім шифрування, також потрібно було обрати місце для зберігання даних. Тобто обрати базу даних, та її тип. Переглядаючи відомі бази даних такі як MySQL, Postgress SQL та MongoDB, перевагу було надано MongoDB.

MongoDB була обрана для зберігання даних у вигляді логів та статистичних даних роботи системи, особливо з огляду на її здатність працювати з нереляційними даними, швидкість обробки та високу масштабованість [46-48]. Це рішення особливо вигідне у контексті відсутності потреби в управлінні користувацькими сесіями, що вказує на відносно прості вимоги до зберігання даних.

У MongoDB відсутність фіксованої схеми даних дозволяє зберігати інформацію в документах, що можуть легко змінюватися та адаптуватися без необхідності перебудови всієї бази даних. Ця гнучкість ідеально підходить для зберігання логів, які можуть мати різні формати в залежності від джерела чи типу події, що логується. Логи можуть включати такі дані, як час створення запису, інформаційне повідомлення від сервера та ідентифікатор клієнта, що дозволяє системі швидко аналізувати та реагувати на потенційні проблеми.

Статистичні дані, які зберігаються в MongoDB, можуть включати різні типи інформації, такі як кількість завдань, що були оброблені системою, час, затрачений на кожне завдання, та інші метрики продуктивності. Гнучкість MongoDB дозволяє зберігати ці дані у вигляді структурованих документів, що полегшує збір, аналіз та візуалізацію статистики, а також спрощує звітність.

Ще однією значущою перевагою MongoDB є її здатність до швидкої індексації та пошуку по документах, що робить можливим швидке отримання доступу до необхідних даних. Це критично важливо для реагування на зміни у реальному часі при навантаженні на систему для виявлення аномалій.

Можливості агрегації даних в MongoDB дозволяють виконувати складні запити на аналіз та отримання важливих узагальнень без залучення додаткових аналітичних інструментів. Функціонал агрегації може бути використаний для виявлення трендів в роботі системи, оптимізації ресурсів та вдосконалення процесів обробки даних. Таким чином, вибір MongoDB в контексті зберігання логів та статистичних даних не тільки забезпечує високу продуктивність та

ефективність, але й сприяє гнучкому управлінню даними в динамічних та високонавантажених середовищах.

Окрім вже згаданих переваг, MongoDB пропонує ще кілька ключових функцій, які роблять її ідеальним вибором для динамічних систем, де зберігання та аналіз даних мають критичне значення. Наприклад відмінні можливості реплікації, тобто MongoDB підтримує реплікацію даних, забезпечуючи високу доступність і надійність. Реплікація даних дозволяє створювати копії даних на різних серверах, що знижує ризик втрати даних у випадку збоїв та забезпечує безперервність обслуговування. Це особливо важливо в критичних застосуваннях, де час простою має бути мінімізований.

Слід також згадати про те що MongoDB дозволяє виконувати не тільки стандартні CRUD-операції (створення, читання, оновлення, видалення), але й складніші операції, такі як текстові пошуки та геопросторові запити [49-50]. Ну і звісно ж гнучка інтеграція з сучасними мовами програмування та платформами також виглядає дуже перспективно. Як і у випадку з алгоритмом шифрування AES про який було згадано раніше, гнучка інтеграція з сучасними мовами програмування та платформами також виглядає дуже перспективно. MongoDB пропонує великий вибір драйверів для інтеграції з різними мовами програмування, зокрема з NodeJS, Python, Java та іншими. Це забезпечує легку інтеграцію MongoDB у більшість сучасних архітектур та додатків, тому при додаванні сервісів на інших мовах програмування, робота з базою даних особливо не зміниться, що економить багато часу та в цілому полегшує розробку.

На додаток до зазначених можливостей, MongoDB підтримує горизонтальне масштабування, що дозволяє ефективно обробляти великі обсяги даних і високі навантаження. Використання шардингу дозволяє розподіляти дані по кількох серверах, що забезпечує високу продуктивність і стійкість системи навіть при значному збільшенні обсягів даних.

На рисунку 2.4 який зображено нижче наведено просту схему роботи нереляційної бази даних.

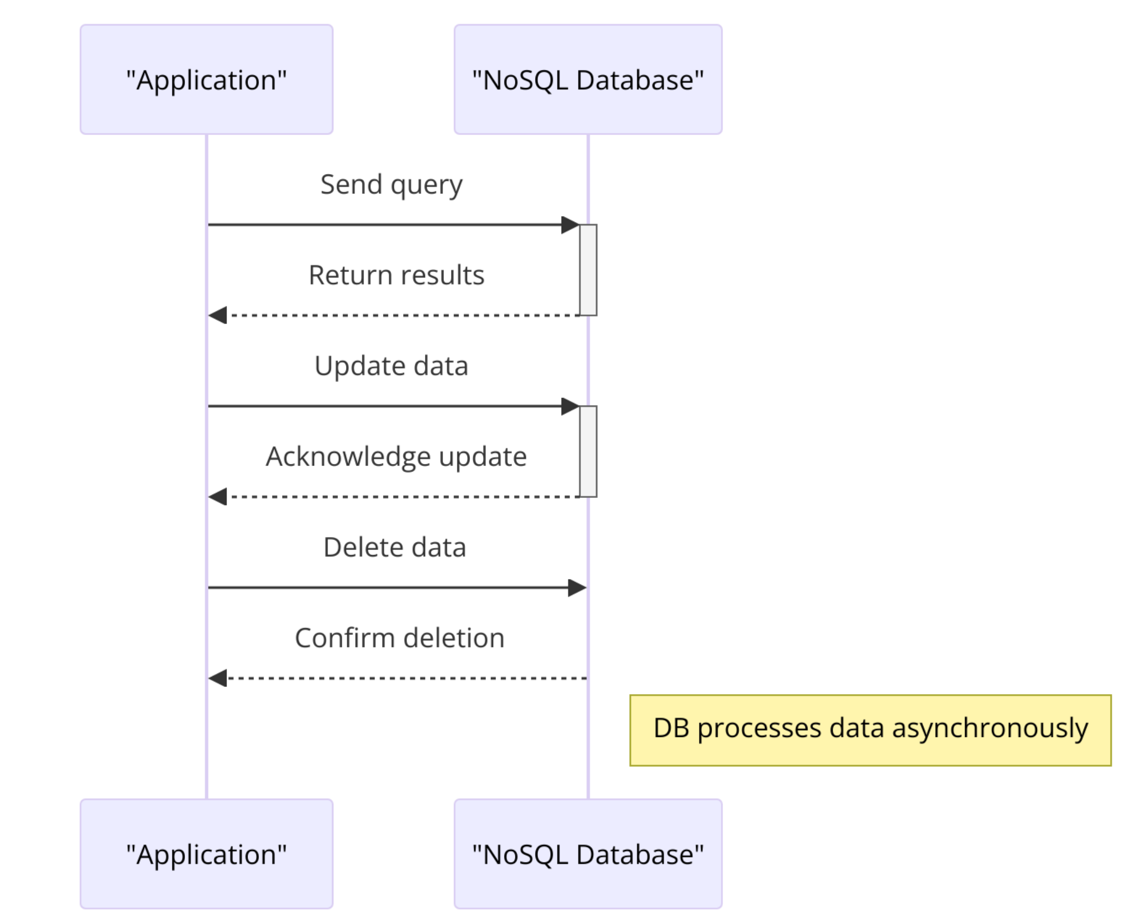


Рисунок 2.4 – Схема роботи нереляційної бази даних

Висновки

Для реалізації серверної частини розподіленої системи було обрано платформу Node.js. Цей вибір обґрунтований високою продуктивністю та асинхронною обробкою подій, що забезпечує швидку реакцію системи на запити клієнтів. Крім того, Node.js підтримує WebSockets, що є критичним для забезпечення двостороннього зв'язку між сервером та клієнтами.

Згідно з дослідженням, вибір MongoDB для зберігання логів та статистичних даних дозволить ефективно керувати великими обсягами інформації. MongoDB, як документно-орієнтована база даних, забезпечує гнучкість у зберіганні та обробці даних завдяки своїй структурі JSON-подібних документів. Це дозволяє легко масштабувати систему, додаючи нові вузли, коли це необхідно, без значних змін у кодовій базі чи архітектурі. Крім того, MongoDB надає можливості для індексації даних, що значно покращує швидкість запитів та зменшує час доступу до

необхідної інформації. Вбудовані функції агрегації дозволяють виконувати складні операції з даними без необхідності перенесення їх до окремих аналітичних систем, що економить ресурси і знижує затримки.

Таким чином, вибір MongoDB для зберігання логів та статистичних даних забезпечить високу продуктивність системи, можливість її масштабування та адаптацію до зростаючих потреб бізнесу, що є критичним для підтримки конкурентоспроможності та оперативного прийняття рішень.

Для забезпечення безпеки даних під час передачі між клієнтами та сервером було впроваджено алгоритм шифрування AES (Advanced Encryption Standard). AES є стандартом шифрування, прийнятим урядом США, і широко використовується у всьому світі для захисту конфіденційної інформації.

Алгоритм AES забезпечує високий рівень безпеки завдяки використанню симетричного шифрування, де один і той самий ключ використовується для шифрування та дешифрування даних. Це робить AES ефективним та швидким, що є важливим для систем, які обробляють великі обсяги даних у режимі реального часу. Особливо це проявляє себе при впроваджені даного алгоритму шифрування у розподілені системи, які зазвичай є дуже навантаженими.

Шифрування даних за допомогою AES гарантує, що навіть якщо зловмисник перехопить передачу даних, він не зможе їх розшифрувати без відповідного ключа. Це значно знижує ризик витоку конфіденційної інформації, такої як особисті дані користувачів, фінансова інформація або корпоративні дані.

В результаті, використання Node.js для серверної частини та MongoDB для зберігання даних, а також впровадження шифрування за допомогою AES, забезпечує створення високопродуктивної, гнучкої та безпечної розподіленої системи, яка здатна ефективно обробляти великі обсяги даних та адаптуватися до змінних потреб бізнесу. Такий підхід дозволяє забезпечити високу якість обслуговування користувачів, швидкість реагування на запити та надійність зберігання і передачі даних, що є критично важливим у сучасних надійних системах розподілених обчислень.

3 ОРГАНІЗАЦІЯ БАЛАНСУВАННЯ ТА КОМУНІКАЦІЇ В РОЗПОДІЛЕНІЙ СИСТЕМІ

3.1 Реалізація обраного алгоритма балансування.

Реалізація алгоритму була розпочата з вирішення питання місця розташування коду. Оскільки реалізація алгоритму балансування була однією з ключових у розподіленій системі, її потрібно було виділити з решти коду, для покращення читання оскільки в майбутньому можливо потрібно буде змінити алгоритм, або розширити його. Тому було прийнято рішення для реалізації алгоритму створити окремий клас LoadBalancer, який буде в собі містити інформацію про наявні підключення та список завдань які необхідно обробити. Також цей клас одразу було вирішено розширити можливістю підключення до адміністратора, тобто щоб була можливість надсилати оброблені дані та логи одразу до адміністратора в реальному часі. Для цього було додано властивість adminNamespace, яка встановлювалася завдяки методу setAdminNamespace(namespace). Решта ж підключень, які будуть виконувати завдання зберігалися у двох властивостях – workers та sockets.

Причиною створення двох окремих властивостей для зберігання підключень було бажання відокремити реалізацію об’єкта працівника від екземпляра підключення. Адже сам працівник містить в собі багато додаткових даних які потрібно перераховувати та модифікувати що потребує певних ресурсів, і збереження в цьому об’єкті екземпляра підключення робить його набагато більшим, що значно збільшує час обробки.

Але при такому розділені працівник нічого не буде знати про встановлене з'єднання, тобто про сам сокет, тому в собі він просто зберігає текстовий рядок який є унікальним ідентифікатором за яким буде здійснюватися пошук сокета при необхідності надіслати дані.

Після оголошення вищезгаданих методів та властивостей, розпочалися кроки до реалізація самого алгоритму балансування Weighted Round Robin. Але як вже відомо даний алгоритм балансування потребує так званої ваги кожного

підключення між якими він буде розподіляти завдання, для досягнення максимальної ефективності. Щоб отримати вагу було реалізовано допоміжну функцію calculateWeight. Слід наголосити що це саме допоміжна функція, і вона ніяким чином не відноситься до самого класу балансувальника, адже вагу можна обчислювати різними способами, її міг надсилати і сам працівник, тому це залежить від архітектурних рішень при розробці системи. Таким чином дана функція має наступну сигнатуру calculateWeight(coreCount, baseCpuPerformance, ramAvailable). Дана функція враховує три основні параметри, що надсилаються працівником при встановлені з'єднання:

- coreCount (кількість ядер процесора): Кількість ядер процесора визначає здатність клієнта обробляти паралельні завдання. Чим більше ядер, тим більше завдань клієнт може виконувати одночасно;
- baseCpuPerformance (базова продуктивність процесора): Цей параметр відображає продуктивність одного ядра процесора. Він може бути оцінений у бенчмаркових тестах або як у даному випадку взятий з технічних характеристик процесора клієнта;
- ramAvailable (доступна оперативна пам'ять): Оперативна пам'ять впливає на здатність клієнта обробляти велику кількість даних одночасно, зберігати тимчасові дані та кешувати інформацію.

Загальна формула (1) має наступний вигляд:

$$\text{weight} = \alpha \cdot \text{baseCpuPerformance} + \beta \cdot \text{coreCount} + \gamma \cdot \text{ramAvailable} \quad (1).$$

Вибір коефіцієнтів α β та γ здійснюється на основі вагомості кожного з параметрів у загальному контексті продуктивності клієнта. В даній реалізації вони мають наступні значення:

- $\alpha = 1.0$: коефіцієнт для базової продуктивності процесора. Він вказує на те, що базова продуктивність процесора має значний, але не домінуючий вплив на загальну вагу;
- $\beta = 2.0$: коефіцієнт для кількості ядер процесора. Цей параметр має подвоєний вплив порівняно з базовою продуктивністю одного ядра, оскільки

можливість виконувати паралельні задачі є критично важливою для обробки завдань;

– $\gamma = 0.5$: коефіцієнт для доступної оперативної пам'яті. Цей параметр має менший вплив порівняно з іншими двома, оскільки оперативна пам'ять є важливою, але її значення є менш критичним порівняно з процесорними характеристиками в контексті обробки завдань для даної системи.

Коефіцієнти були обрані на основі експериментальних даних та досвіду з реальних навантажень клієнтів. У процесі тестування різних конфігурацій клієнтських систем було виявлено, що кількість ядер процесора (coreCount) найчастіше має більший вплив на здатність клієнта обробляти паралельні завдання, ніж продуктивність одного ядра (baseCpuPerformance). Доступна оперативна пам'ять (ramAvailable) також важлива, але її вплив є меншим порівняно з процесорними характеристиками в контексті обробки завдань.

Після того як було отриману вагу кожного клієнта, можна було переходити до реалізації подальших кроків алгоритму. Сам алгоритм уже реалізовувався як метод класу LoadBalancer і був названий просто WRR . На даному етапі реалізації вже потрібно було враховувати особливості самої платформи NodeJS. Адже як вже сказано JavaScript однопотоковий язик програмування а сама платформа NodeJS реалізує неблокуючу модель вводу/виводу. Отже відповідно дл особливостями язика даний метод було оголошено як async, тобо асинхронним. Адже в майбутньому потрібно буде додавати нові завдання та підключення, видаляти виконані завдання та сокети які було закрито, і всі ці дії не повинні блокувати основний потік виконання. Після цього розпочинається реалізація самого тіла функції WRR.

Першим ділом потрібно перевірити чи є під'єднані працівники які можуть виконувати завдання, тому спочатку йде перевірка на довжину масиву працівників, яка в разі хибного результату одразу завершує роботу алгоритму, повертаючи управління основному потоку, адже не можна розподіляти завдання, якщо немає активних працівників. У разі позитивного результату переходимо до наступної перевірки, а саме перевірки на довжину завдань. Якщо список завдань порожній то його довжина відповідно буде дорівнювати нулю, у такій ситуації також повинно

бути завершено виконання алгоритму, тому по тій же схемі повертаємо управління основному потоку. Цей крок лише з двома перевірками дає змогу мінімізувати появу потенційних помилок. На наступному кроці потрібно було вирішити питання обходу усіх завдань та надсилання цих завдань працівникам. Для обходу усіх завдань було вирішено використати цикл while з перевіркою на довжину масиву з завданнями. Тобто цикл працюватиме доти поки усі завдання не будуть оброблені. Але перед циклом потрібно буде створити змінну workerIndex зі значенням 0.

Потрібно це для звернення до конкретного працівник в масиві, адже цикл while не містить в собі індексу на відміну від for. В тілі циклу першим ділом з черги доступних завдань вибирається перше завдання та записується у змінну task, після чого відповідно видаляємо його зі списку. Дану зміну разом із поточним індексом передаємо у метод sendTaskToWorker, який має наступну сигнатуру: async sendTaskToWorker(index, task).

Як зрозуміло з назви даний метод надсилає завдання до працівника. В тілі метод звертається до відповідного працівника за індексом який він отримав в якості першого параметру і витягує ідентифікатор сокету який після цього використовується для пошуку відповідного підключення вже в списку сокетів. Знайшовши відповідний сокет, метод надсилає за ним завдання, та підписується на відповідь, так званий callback, в якому перевіряється статус відповіді, який може або ok або error.

В залежності від статусу додаємо одиницю до показника кількості успішних або не успішних виконаних завдань відповідного працівника. Після чого незалежно від успішності виконання завдання, інкрементується загальна кількість завдань надісланих працівнику. В результаті маємо 3 властивості працівника які показують його відсоток успішності виконання завдань. Звісно слід відмітити що даний метод також є асинхронним, тому як тільки відбувся його виклик то одразу зменшуємо вагу даного працівника на одиницю відповідно до алгоритму WRR, після чого цикл починається знову.

Тепер переходимо до одного з найважливіших кроків, а саме реалізації переходу до іншого працівника коли вага попереднього дорівнює нулю. Для цього після початку циклу додається ще один цикл while , який буде працювати поки вага

працівника за індексом який було оголошено перед першим циклом буде дорівнювати нулю. Але оскільки вібувається модифікація при кожному обході циклу поточної ваги працівника, в один момент потрібно буде повернутися до її початкового значення, тому працівник має властивість originalWeight , в якому записана початкова вага. Отже коли умова для входу у вкладений цикл задовольняється це означає, що поточний працівник має нульову вагу, і він більше не може приймати завдання, тому відбувається інкрементування індексу працівника workerIndex.

Після чого йде перевірка на входження індексу працівника в доступний діапазон, тобто чи не виходить індекс за доступні межі які вимірюються шляхом взяття довжини масиву працівників. Якщо індекс не виходить за межі, тоді просто продовжується виконання вищезгаданих операцій, і в результаті усі працівники отримують завдання відповідно до їх ваги.

Після виходу індексу за межі створюється цикл for, який проходить по всіх працівниках, і для кожного працівника скидає вагу до початкового значення, шляхом копіювання у weight значення яке знаходиться у властивості originalWeight .

Після чого також скидається індекс працівників workerIndex і відбувається перенесення наступних обходів у іншу фазу event loop. Тобто на цьому моменті враховується неблокуюча модель NodeJS, що дозволяє виконатися синхронному коду, подіям які були викликані і так далі. Саме у цей час можуть бути додані наступні нові завдання та підключення працівників. Перенесення відбувається шляхом повернення і очікування виконання Promise, який в собі викликає setImmediate(resolve) що говорить NodeJS виконати весь наступний код у кінці наступної фази check. Сама ж конструкція є не дуже складною і виглядає наступним чином await new Promise((resolve) => setImmediate(resolve)). Коли синхронний код завершив виконання і прийшла черга мікрозадач де виконується Promise продовжується виконання циклу на моменті його зупинки, де одразу після цього викликається break, що завершує вкладений цикл while.

Після чого продовжується виконання зовнішнього циклу який знову надсилає завдання кожному працівнику за вищезгаданими кроками. На цьому реалізація алгоритму закінчена а блок схему алгоритму наведено на рисунку 3.1.

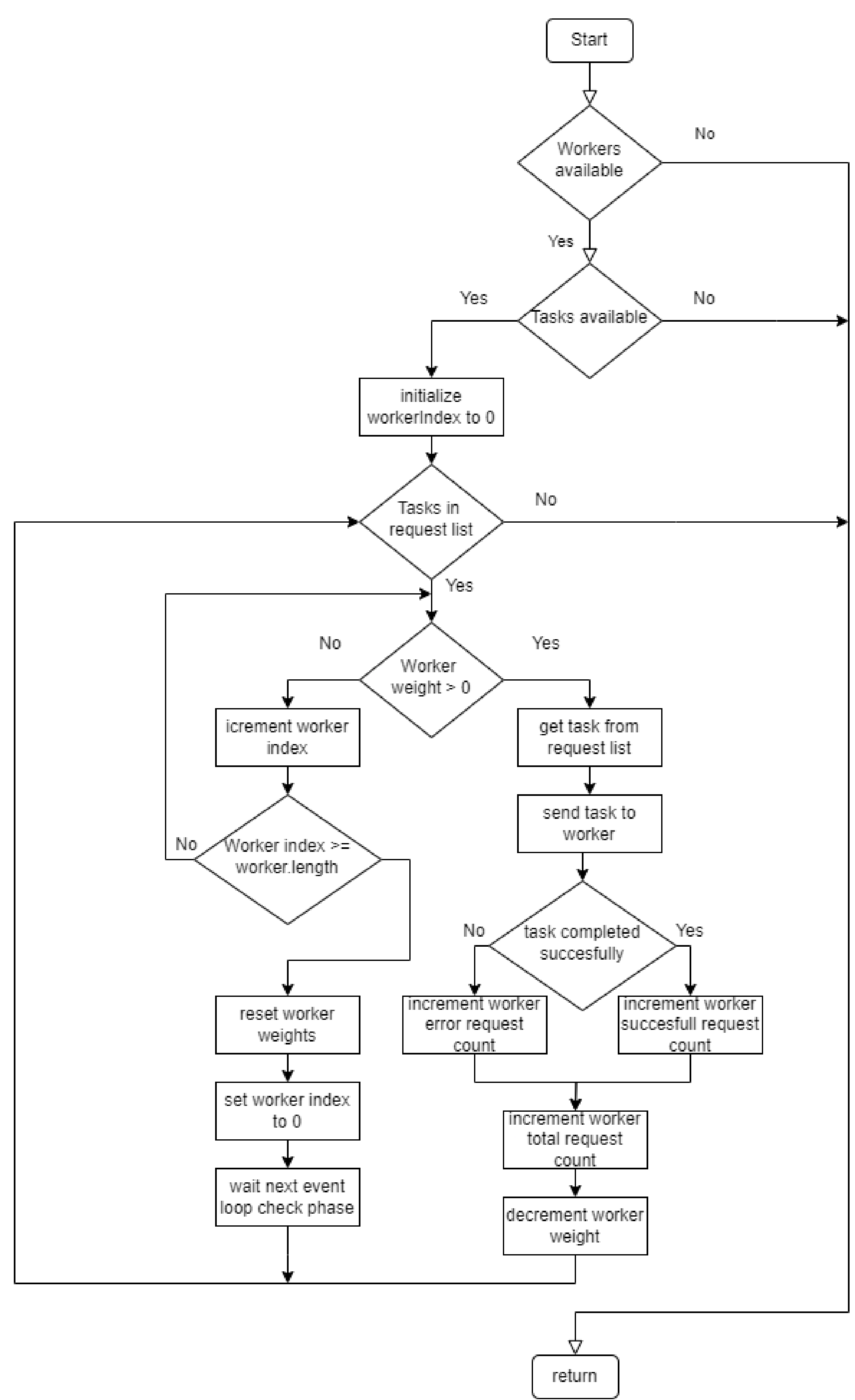


Рисунок 3.1 - Блок-схема алгоритму балансування

Хоча алгоритм вже написано, залишилося ще одне питання - де викликати даний алгоритм, та що робити коли відбувається його завершення по згаданим вище причинам.

Перш за все потрібно сказати, що повертатися при підключені модуля з даним класом буде одразу його екземпляр, в NodeJS це гарантує що екземпляр буде лише один. Це дасть можливість зменшити залежність усіх інших компонентів від даного класу, та керувати ним буде значно легше.

З додаванням та видаленням підключень все просто, підключення будуть додаватися в момент з'єднання з сокетом працівника після обчислення його ваги. Закриватись сокет буде відповідно у момент втрати з’єднання з працівником.

Нові завдання для обчислення ж будуть додаватися клієнтами які будуть використовувати дану систему, або ж адміністратором, тому для цього буде використовуватися http запит, в обробнику якого й буде відбуватися парсинг завдання після чого його буде додано до черги. З викликом ж самого алгоритму все складніше. Слід врахувати, що алгоритм працює асинхронно, і може завершуватися в любий момент.

Тому потрібно реалізувати механізм який буде очікувати завершення алгоритму, та в момент якщо алгоритм відпрацював, запустити його знову.

Також слід врахувати що цей механізм не повинен блокувати головний потік виконання, тому він буде асинхронним. Отож було реалізовано допоміжну функцію execute, яка має наступну сигнатуру: async function execute(balancer).

Передбачається що дана функція в майбутньому може використовуватися не тільки для даного алгоритму, тому як параметр вона буде приймати екземпляр балансувальника та в тілі запускати метод балансування, очікуючи на результат виконання.

Тобто, функція буде очікувати завершення алгоритму балансування, після чого ставить у чергу через таймер з нульовим часом очікування саму ж себе з тим же екземпляром класу у виді параметра, таким чином виконуючи рекурсивний виклик.

Умовою виходу з цієї рекурсії буде закінчення виконання програми, але переповнення стеку викликів не буде, оскільки функція завжди спершу очікує завершення попереднього виклику роботи алгоритму. Зображення схеми роботи функції запуску алгоритму балансування наведено на рисунку 3.2 нижче.

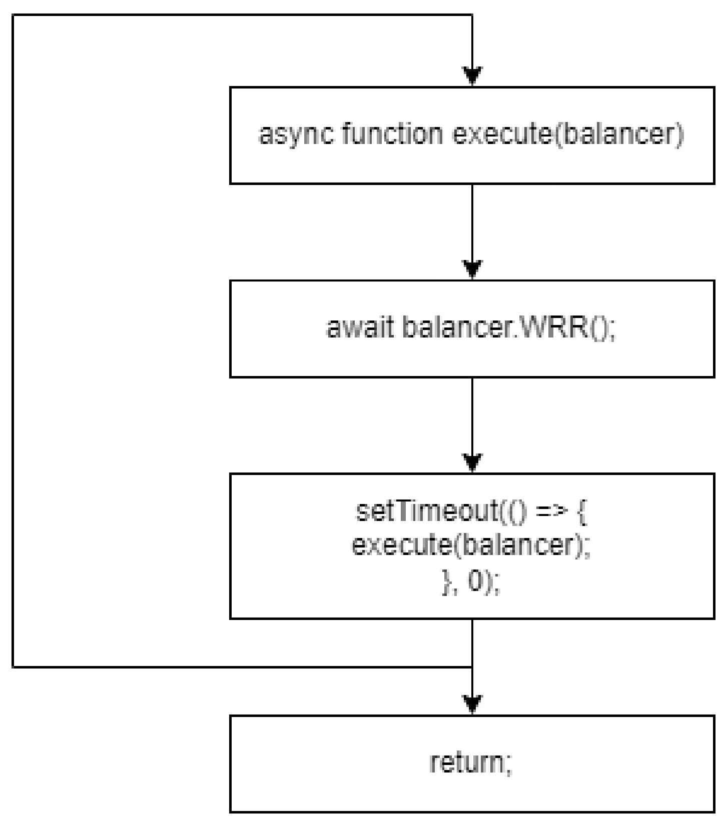


Рисунок 3.2 - Блок-схема роботи функції запуску алгоритму балансування

Так як статистичні дані з серверу будуть переглядатися в окремій програмі, то для неї потрібно буде пересилати результати роботи працівників, тому одразу в LoadBalancer класі було реалізовано метод startSendingRequestInfo, що при виклику створює інтервал, який через кожну секунду буде надсилати сокету адміністратора інформацію про кожного працівника.

Оскільки це інтервал, то викликати метод потрібно буде лише один раз в основному файлі програми. На цьому поточна реалізація класу LoadBalancer завершена, хоча в майбутньому цей клас може бути розширено додатковим функціоналом, таким як підтримка різних типів підключень.

Наприклад одне підключення буде для адміністратора який матиме змогу редагувати роботу сервера, шляхом видалення або додавання певних даних, а друге лише для моніторингу без можливості втручання у роботу системи.

3.2 Реалізація механізмів комунікації

Так як завдання було створити ефективну та гнучку розподілену систему, до якої є можливість під'єднатися з будь-якого куточка країни, як вже обговорювалося прийнято рішення використати WebSockets. Хоча WebSockets і мали багато переваг, основною ідеєю все ж таки було дати можливість працівникам без додаткових налаштувань їх комп'ютера просто та бистро приєднатись до системи.

Для цього головний сервер до якого вони будуть приєднуватися ставиться на хостинг, а працівники будуть знати його адресу. Тому перш за все в основному файли програми потрібно було створити сервер.

У випадку з NodeJS використовувався модуль HTTP та метод createServer, після чого в функції обробнику підключень додаються відповідні заголовки, які дозволяли кросс-доменні запити. Після цього йде підписка на подію request в яку передається знову ж таки функція обробник. Також даному серверу задається порт через метод listen(), який викликається в самому кінці основного файлу.

Слід відмітити, що під час перегляду потенційного функціоналу системи, було вирішено впровадити гібридну комунікаційну архітектуру. Тобто дати можливість серверу спілкуватися як через HTTP, так і через WebSockets.

Зроблено це було з метою оптимізації роботи самого сервера, адже велика кількість запитів у реальному часі досить сильно навантажують систему, тому певні менш важливі запити були винесені у HTTP, в основному це відноситься до запитів які надходять від додаткових сервісів та програм, таких як програма для слідкування за станом системи.

Так як архітектура комунікації буде гібридна, очікувалось досить велика кількість коду, тому спершу впроваджувалася маршрутизація та використання middlewares, які виносилися у окремі файли.

Всього middleware на момент розробки планувалося створити 4. Щоб почати працювати з даними які надходять до сервера у тілі HTTP запита, їх потрібно було звіти дістати шляхом використання stream. Для цього було написано bodyParserMiddleware функцію, яка у тілі створює порожній об'єкт, після чого починає зчитувати chunk (порції даних у байтах). Після закінчення зчитування відбувається парсинг байтів у JSON, який після цього записується у об'єкт request.body та викликається функція next().

Слід додати що кожен middleware приймає однакові параметри, такі як: request, response, next. Тіло відповід та тіло запиту - request та response відповідають за зчитування та відправку даних при запиті з клієнта.

Що до next() то це функція яка, говорить програмі перейди до виконання наступного middleware, адже ці функції повинні виконуватись у чітко заданому порядку, і викликається next() в кінці кожного middleware. Наступні три middlewares виконують наступне:

– jsonParserMiddleware - розширює об'єкт response шляхом додавання в нього нового метода send() , який в свою чергу до відповіді додає заголовок 'Content-Type': 'application/json', що говорить клієнту приймати відповідь у форматі JSON. Та власне й надсилає відповідь до клієнта у цьому форматі;

– функція loggerMiddleware веде логування HTTP-запитів. Вона виводить у консоль поточну дату та час, метод запиту і URL;

– urlParserMiddleware розбирає URL запит і зберігає його компоненти у властивості request.params. Це дозволяє легко отримувати доступ до різних частин URL для подальшої обробки.

Розроблені проміжні функції (middleware) забезпечують ефективну обробку HTTP-запитів, починаючи від зчитування та парсингу даних тіла запиту, до логування та розбору URL. Використання цих функцій дозволяє організувати структуровану і зручну для розширення обробку запитів у Node.js-додатку.

Оскільки middlewares розроблені лише для парсингу даних для подальшого їх опрацювання, потрібно також додати кінцеві точки, в яких буде відбуватися опрацювання даних по заданному URL, і саме для цього створювалися маршрути.

Для зручного керування маршрутами та їх винесення в окремі файли, також було створено простий клас Router. Якщо говорити простіше, то цей клас дозволяє реєструвати різні маршрути та обробляти запити, направляючи їх до відповідних обробників на основі шляху URL.

Клас Router має властивість routers, яка є екземпляром Map. Ця структура даних використовується для зберігання пар ключ-значення, де ключем є назва маршруту (pathname), а значенням - функція-обробник для цього маршруту. Крім того даний клас має два методи: registerRouter та startRouting.

Метод registerRouter має наступну сигнатуру registerRouter(pathname, router) та використовується для реєстрації нового маршруту. Він приймає рядок, який представляє шлях URL, за яким буде оброблятися запит та функцію-обробник, яка буде викликана, якщо шлях запиту співпадає з зареєстрованим шляхом.

Метод startRouting(req, res) використовується для запуску маршрутизації запитів. Він приймає два параметри: об'єкт запиту, що містить дані про HTTP-запит та об'єкт відповіді, що використовується для відправки відповіді клієнту.

Цей метод спершу отримує шлях з об'єкта запиту req.params.pathname та розбиває його на частини, витягує першу частину шляху, яка визначає маршрут та перевіряє наявність цього маршруту у властивості routers.

Якщо маршрут не знайдено, відправляє клієнту повідомлення про помилку. Якщо маршрут знайдено, викликає відповідну функцію-обробник для цього маршруту. Екземпляр цього класу створюється у головному файлі програми, після чого викликається метод registerRouter та передаються маршрут разом з функцією обробником.

Самі ж функції обробники, це анонімні функції які виносяться в окремі файли, для очищення основного коду, та примають знову ж таки request та response. В собі ці обробники перевіряють наступну частину маршруту та метод який надходить з клієнта, це може бути один з методів які використовують при обробці запитів у HTTP таких як GET, POST, PUT або DELETE.

У випадку якщо обробник може опрацювати такий URL з відповідним методом то він виконує необхідні операції за даним запитом та повертає результат назад до клієнта завдяки функції response.send() яку надав middleware.

Якщо ж відповідного маршруту не знайдено, то клієнту повертається про це повідомлення. Тепер коли всі основні механізми комунікації через HTTP реалізовано, потрібно вирішити питання їх виклику. Виклик усіх middlewares та вибір обробника для маршруту буде здійснюватися під час спрацювання події request об'єкта httpServer. При цьому це лише подія на яку можна підписатись, на момент створення самого серверу, тому самі маршрути можна зареєструвати нижче, що не скажеш про middlewares, адже вони будуть викликатися перед обробкою маршрута, тому їх потрібно оголосити раніше.

Саме ж послідовний виклик middlewares відбувається у функції runMiddleware, яка викликається при спрацюванні події request. Функція runMiddleware відповідає за послідовне виконання проміжних функцій та самого обробника маршрута.

Вона працює рекурсивно, викликаючи кожну проміжну функцію по черзі та передаючи їй об'єкти req, res та функцію next. Перед запуском функції runMiddleware ініціалізується змінна middlewareIndex, яка відслідковує поточний індекс середовища у ланцюжку.

Початкове значення цієї змінної - 0. Функція перевіряє, чи поточний індекс middlewareIndex менший за кількість обробників у масиві middlewares. Якщо умова виконується, викликається проміжна функція відповідно до індексу. Проміжна функція з масиву middlewares викликається з трьома параметрами про які вже говорилося раніше: request, response та next().

На цьому моменті слід сказати, що функція next() являє собою callback в тілі якого за допомогою можливості JavaScript - замиканню, інкрементується поточний індекс що відповідає за вибір функції з масива middlewares, після чого рекурсивно викликається runMiddleware функція.

Якщо всі middlewares були виконані (тобто middlewareIndex досяг значення, що дорівнює кількості функцій у масиві middlewares), функція runMiddleware передає управління маршрутизатору router, викликаючи його метод startRouting з об'єктами request та response. Блок-схема на якій зображено алгоритм роботи даної функції наведено на рисунку 3.3.

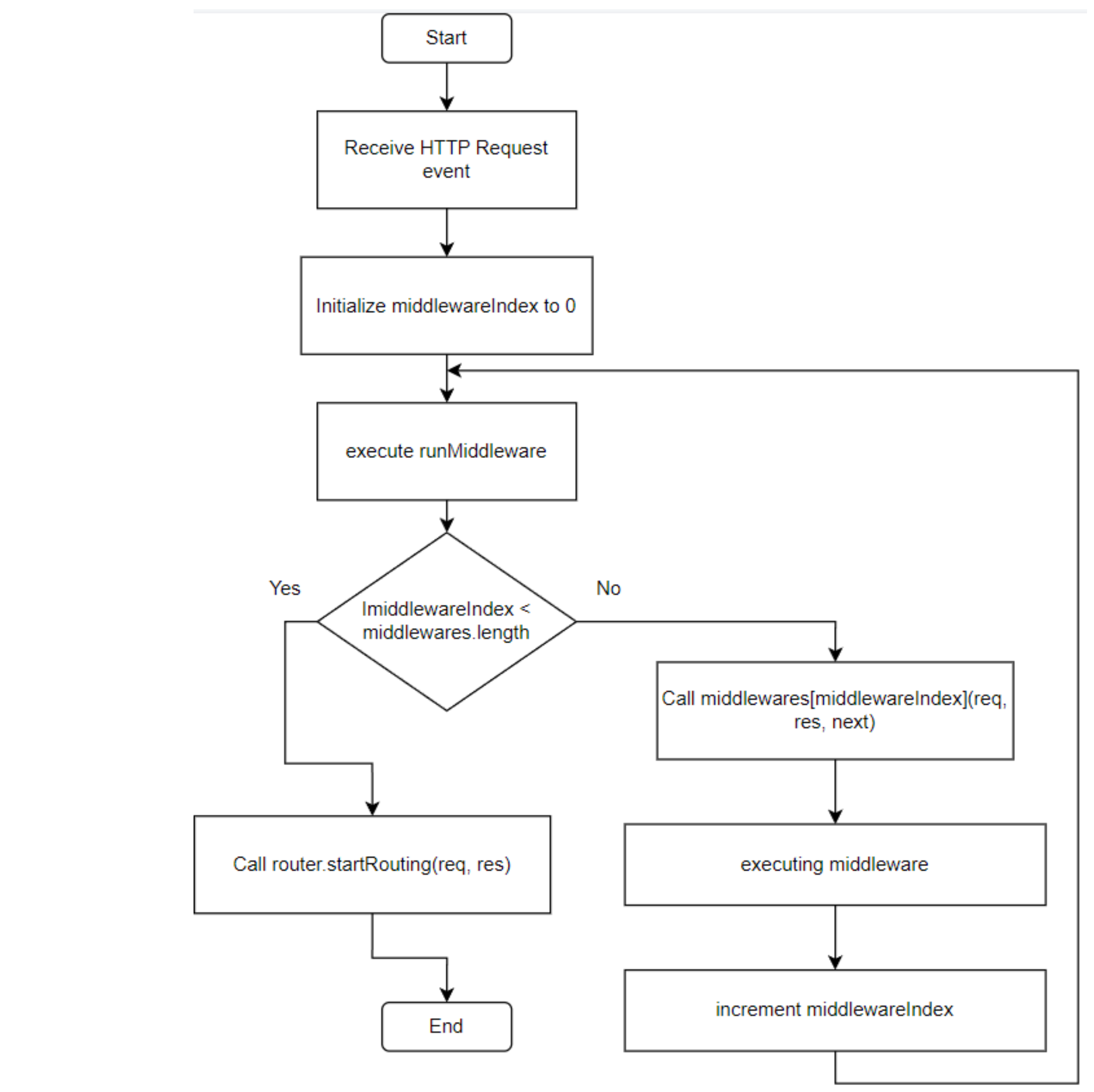


Рисунок 3.3 - Алгоритм роботи функції runMiddleware

Хоча на цьому етапі розробка функцій для з’єднання та комунікації через HTTP була завершена, як вже було сказано реалізувати потрібно було саме гібридну комунікаційну архітектуру, тому тепер розпочалася робота з WebSockets. Незважаючи на те що в проєкті і використовується бібліотека socket.io, яка надає дуже багато можливостей при роботі з сокетами з коробки, знати як вона працює все ж таки необхідно. Тому перш за все потрібно розібратися як працює Engine.IO – рушій даної бібліотеки. WebSocket – це передова технологія, яка дозволяє відкривати двосторонній інтерактивний сеанс зв'язку між користувачем і сервером, але розпочинається все з звичайного HTTP запиту, який називається WebSocket handshake. На початку з'єднання з Engine.IO сервер надсилає наступну інформацію:

– sid - ідентифікатор сесії, його потрібно включати в параметр запиту sid у всіх наступних HTTP-запитах;

– масив upgrade, що містить список усіх “кращих” типів зв’язку, які підтримуються сервером;

– значення pingInterval та pingTimeout, які використовуються у механізмі heartbeat (серцебиття).

Клієнт надсилає спеціальний HTTP запит до сервера з заголовком Upgrade, який вказує на бажання перейти від HTTP до WebSocket протоколу. Це називається «WebSocket handshake». Після чого сервер, отримавши цей запит, відповідає з підтвердженням, що він підтримує перехід до протоколу WebSocket, і надсилає відповідний заголовок у відповіді. Після успішного обміну заголовками клієнт і сервер переходять від використання HTTP до WebSocket протоколу. Відтепер з'єднання стає постійним і двонаправленим, що дозволяє обмінюватися даними в реальному часі. З цього можна зробити висновок що WebSocket потребує спершу HTTP з’єднання, саме тому socket.io має можливість в якості параметра конструктора класу Server який надає можливість роботи з сокетом приймати екземпляр класу створений за допомогою http.createServer(), що надає NodeJS. Отже після створення HTTP серверу та реєстрування обробника події request, відбувається створення екземпляру класу Server, що надає socket.io. Новий сервер окрім вищезгаданого першого параметру в якості якого виступає екземпляр HTTP сервер, потрібно додати другий параметр - об’єкт що дозволяє кросс-доменні запити.

Тепер потрібно сказати що як і у випадку з HTTP сервером, потрібно додати функції обробники, які будуть слухати відповідні події. Спершу створюється функція onConnection що параметром приймає сокет, який буде передаватися у кожен обробник автоматично. Виконання цієї функції встановлюється на подію «connectio», яка викликається кожен раз коли підключається новий користувач. Ця функція в тілі реєструє вже інші обробники, які в свою чергу також отримують сокет та екземпляр класу Server. Зроблено це виключно для очищення коду та не є обов’язковим. Саме на цьому етапі в LoadBalancer через метод addSocket додаються сокети, які будуть отримувати завдання. Після чого на тільки що під’єднаний сокет

одразу надсилається запит на отримання інформації про систему користувача, для подальшого обчислення її ваги, та моніторингу. Інший обробник який реєструється в тілі також опрацьовує події, такі як disconnect, connection:send-info та інші. Для підключення користувачів які не виступають в ролі працівників, відведено окремі кімнати, які socket.io надає змогу створювати. Для цього звертаємося до екземпляра класу Server та викликаємо метод of() , в який передаємо назву кімнати що потрібно створити. В тілі даний метод створює новий сокет, який прив'язаний до заданого імені та повертає його.

У випадку з створенням сокету для програми моніторингу виклик виглядає наступним чином: io.of('/admin'), де io екземпляр класу Server.

Новий сокет також має ідентичні події, і на виклик події connection програмі моніторинга надсилається список під'єднаних клієнтів. Зроблено це тому що програма може запуститися в момент коли клієнти вже приєдналися, і в цей момент їй потрібно отримати інформацію про вже підключених користувачів. Звісно при під'єднані нового користувача про нього на цей сокет надсилається інформація.

Таким чином забезпечується постійна актуальність даних при моніторингу системи. Хоча основний механізм комунікації реалізовано, залишилося створити, сервіси що будуть працювати з базою даних та забезпечити шифрування даних.

Для можливості використання бази даних потрібно додати модуль «mongodb», що надає в наше користування клас MongoClient який в подальшому буде використовуватись для управління віддаленою базою даних. В якості параметру цей клас приймає стрічку з даними про підключення, яку можна отримати на головному сайті mongo. Як і у випадку з роутером було створено окремий клас MongoService, який в конструкторі встановлює з'єднання з базою і повертає екземпляр з'єднання у властивість client даного класу.

Крім цього MongoService має ще два методи: getMongoClient() та closeConnection(), які відповідно повертають екземпляр клієнта та закривають з'єднання з базою. Екземпляр класу MongoService створюється одразу в основному файлі програми після чого коли створюється додатковий сервіс який потребує екземпляр бази даних, він отримується за допомогою функції getMongoClient() та

передається як параметр у цей сервіс. У даному проєкті такий сервісів всього два: LoggerService та StatisticService.

Ці сервіси відповідають за запис та зчитування даних з бази, і їх винесення в окремі файли потрібні лише для покращення читабельності коду та забезпечення зручного подальшого його розширення.

Кожен з цих сервісів при створені у конструкторі потребує клієнт, який отримуємо з класу підключення до бази даних. Працюють вони ідентично, тільки відповідають за зчитування та запис різних даних, як зрозуміло з назви LoggerService відповідає за роботу з лагами у базі даних в цей як StatisticService відповідає за зчитування та запис статистичних даних сервера. Оскільки операції з базами даних досить часозатратні, то додавання логів було вирішено зробити з затримкою в 10 секунд.

Так як логів у системі дуже багато, наприклад клієнт приєднався, надіслав відповідь до завдання, від’єднався тому всі вони накопичуються у відповідному масиві який знаходиться у класі Store - це таке собі глобальне сховище тимчасових даних, в ньому зберігаються як тимчасові логи так і статистика системи.

Таким чином реалізується буферизація логів (або батчинг логів) - це техніка збирання логів у буфер протягом визначеного періоду часу або до досягнення певного обсягу даних, а потім відправлення цих логів як однієї порції (батчу) до бази даних.

Це дозволяє зменшити кількість звернень до бази, що покращує продуктивність системи та знижує навантаження на саму базу даних. Після надсилання даних, тимчасове сховище очищується. У тілі LoggerService використовується основні CRUD операції. NoSQL бази даних, такі як MongoDB, Cassandra, Redis, як і більшість підтримують CRUD-операції, але синтаксис і підхід відрізняються. Наприклад, у MongoDB ці операції виконуються наступним чином:

- create: db.collection.insertOne({ key: value });
- read: db.collection.find({ key: value });
- update: db.collection.updateOne({ key: value }, { $set: { key2: value2 } });
- delete: db.collection.deleteOne({ key: value }).

StatisticService працює ідентично, різниця лише у даних які він відправляє у базу та шлях за яким він створює документ, для LoggerService це “dest-server logs” а для статистичних даних це “dest-server statistic”. Після завершення роботи з кодом бази даних, потрібно було забезпечити захист передачі даних через HTTP та WebSockets. Так як socket.io не надає вбудованих варіантів захисту даних, це потрібно вирішити на момент формування даних при передачі.

Для шифрування даних було створено функцію encrypt, яка приймає текст для шифрування та сам ключ яким буде відбуватись шифрування. Ключ знаходиться у файлі .env, що використовується у проєктах для зберігання змінних середовища. Замість файлу можна було би додавати ключ при кожному запуску програми через командний рядок, але це не зручно.

В тілі функції відбувається генерація Initialization Vector – випадковий або псевдовипадковий набір байтів, який використовується разом із секретним ключем для шифрування даних. IV забезпечує унікальність зашифрованого тексту, навіть якщо той самий відкритий текст шифрується багаторазово з використанням того самого ключа.

Після цього за допомогою модуля crypto що надає NodeJS відбувається саме шифрування з використанням функції crypto.createCipheriv() . Параметрами ця функція приймає назву шифру, у даному випадку це aes-256-cbc, текст для шифрування у вигляді масиву байтів та IV. Для переведення тексту у масив байтів використовується Buffer. Далі шифр інтерпретується як “utf-8” символи та записується у hex форматі. Повертається з даної функції сам шифр з дописаним до нього IV через знак двох крапок. Всі ці операції відбуваються у тілі функції encrypt, яка і переводиться як зашифрувати.

Шифрування – це добре, але тепер потрібно навчитись розшифровувати ці дані. Для цього створюється функція decrypt яка приймає зашифрований текст та ключ який він був зашифрований.

У тілі функція спочатку розділяє зашифрований текст по знаку двох крапок що додаються при шифруванні та роблять текст унікальним. IV в подальшому конвертується у “hex” та передається разом з шифрованим текстом у функцію

crypto.createDecipheriv(). Повертається з функції decrypt розшифрований текст у форматі “utf-8”.

Коли функції шифрації та дешифрації готові, настав час скористатись можливостями раніше розроблених middlewares, а конкретніше jsonParserMiddleware, який розширює вбудований метод res.end() і повертає відповідь на HTTP запит у форматі JSON.

Перед поверненням відповіді викликається тільки-що створений метод encrypt в який передається відповідь, і тепер вже зашифрований результат роботи функції шифрування повертається клієнту.

Таким простим розширенням було забезпечено шифрування будь-якої відповіді на HTTP запит, що зекономило дуже багато лишніх стрічок коду. Таке ж розширення задається і для bodyParserMiddleware що в свою чергу за допомогою виклику функції decrypt забезпечує розшифровування кожного HTTP запиту який надходить до серверу.

На даний момент реалізації такого механізму для обробки WebSocket запитів немає, тому для них необхідно буде шифрувати дані на кожне відсилання даних вручну. Те саме відноситься і для дешифрування даних. На цьому етапі реалізація гібридної комунікаційної архітектури завершено.

3.3 Реалізація засобу моніторинга серверу

Так як при поточна версія програми повинна виконувати лише моніторинг системи та видалення логів, додаткових маніпуляцій з electron робити не потрібно. Для початку роботи вистачає створення шаблону electron з React за допомогою npm команди. В preload.js скрипті який виконується при запуску програми створюється красивий loader який буде виводитись користувачеві під час підвантаження основної сторінки.

Оскільки React використовується для створення Single Page Applications, то достатньо створити лише одни index.html файл в якому створюється один блок. Цей html файл відображається при завантаженні програми завдяки функції

win.loadFile(), яку надає electron. Після таких базових налаштувань настав час створення самого ReactDOM.

Початкова ініціалізація самого React відбувається у файлі main, де також задаються відповідні маршрути завдяки використанню пакета react-router-dom. Загалом буде доступно п’ять сторінок, за відображення яких відповідають компоненти ConnectionsPage, ConnectionIdPage, LogsPage, StatisticPage та HomePage.

Перехід до цих сторінок буде здійснюватись за допомогою посилань, що описані на боковій панелі, що знаходиться у компоненті MainLayout та є контейнером в який поміщаються всі наступні компоненти описані у маршрутах. При завантаженні програми першою користувач бачить сторінку з привітанням та вступною інформацією для кращого розуміння роботи програми. За це відповідає компонент HomePage. Дана сторінка відіграє виключно інформаційну роль, тому більшу увагу було приділено іншим функціональним сторінкам.

Перша функціональна сторінка на яку користувач зможе перейти шляхом вибору пункту Connections на боковій панелі називається ConnectionsPage. Цей компонент забезпечує відображення списку активних з'єднань у реальному часі. Також він використовує збереження стану з'єднань за допомогою socketStore і відображає список з'єднань, якщо вони є, або повідомлення про їх відсутність. Компонент Connections має наступні функції:

– підключення до стану з'єднань: Використовується функція socketStore для отримання списку з'єднань зі сховища стану;

– відображення з'єднань: Якщо у списку є активні з'єднання, компонент відображає їх за допомогою дочірнього компоненту ConnectionList. Цей компонент приймає масив з'єднань як параметри і відображає їх у вигляді списку;

– повідомлення про відсутність з'єднань: Якщо активних з'єднань немає, компонент відображає повідомлення «There are no active connections».

Особливістю даного компонента як і більшості наступних є динамічне відображення даних, адже компонент автоматично оновлює список з'єднань у реальному часі, використовуючи підписку на зміни у сховищі socketStore.

Що до socketStore, то він реалізує сховище стану для управління з'єднаннями і логами, а також ініціалізацію підключення до сервера за допомогою socket.io. О

сновні функції включають додавання, видалення та оновлення з'єднань, а також обробку різних подій, що надходять від сервера.

Так як додаток моніторингу є клієнтським, то він використовує для WebSocket з’єднання socket.io-client. На рисунку 3.4 наведено схему яка зображає потік роботи даного React компонента.

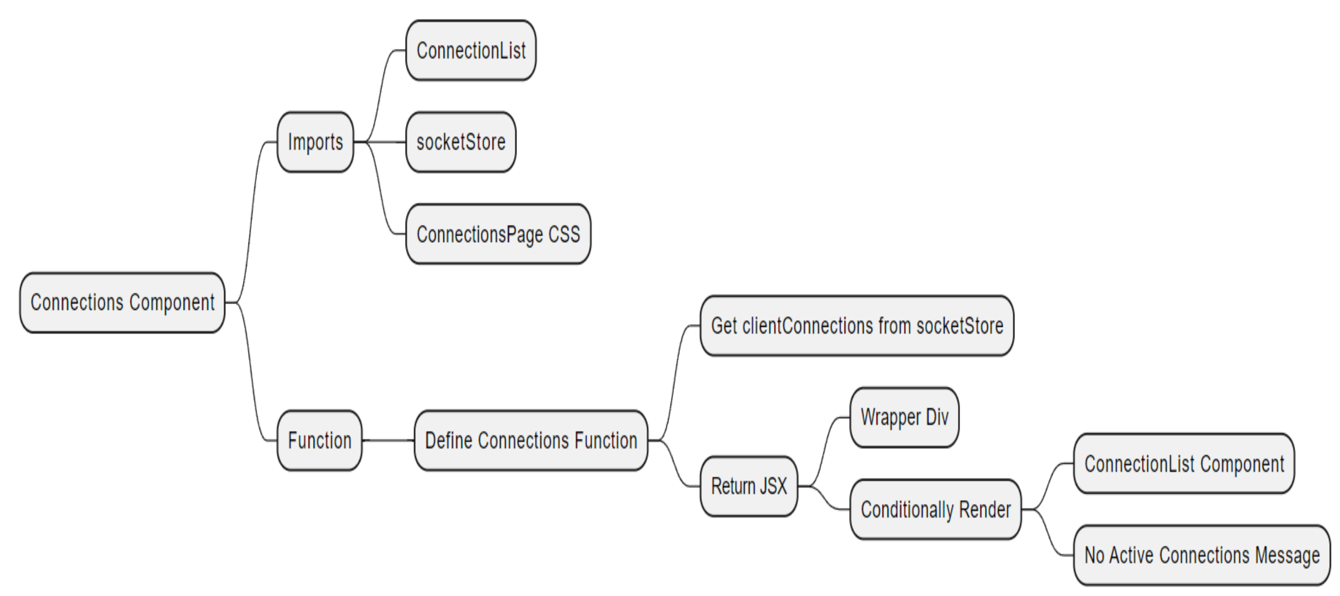


Рисунок 3.4 – схема потоку роботи компонента ConnectionsPage

Наступний не менш важливий компонент це LogsPage. LogsPage є важливим компонентом, який забезпечує зручне відображення та управління логами на відповідній сторінці додатку.

Він дозволяє користувачам ефективно шукати, відображати та видаляти логи з сервера. Використання API для отримання та видалення логів, а також обробка введення користувача для пошуку конкретних записів робить цей компонент надзвичайно функціональним та зручним.

Компонент LogsPage має різні стани, такі як масив логів, отриманих з сервера, стрічка для пошуку логів, булева змінна для визначення активності

режиму видалення логів, а також стани для управління процесом отримання та видалення логів з сервера.

Використання хуків useFetching та useEffect дозволяє при завантаженні сторінки викликати функцію fetchLogs, яка отримує всі логи з сервера за допомогою сервісу LogService. Це забезпечує актуальність даних та їх оновлення в реальному часі. Коли сторінка завантажується або коли користувач виконує певні дії, які впливають на логи, ці хуки автоматично оновлюють стан компоненту, викликаючи нові запити до сервера для отримання актуальних даних.

Функціонал видалення логів реалізовано через компонент LogDeleteForm, який дозволяє користувачам видаляти логи за допомогою запиту на сервер. Це забезпечує можливість видаляти як окремі логи, так і діапазон логів, залежно від потреби користувача. Коли користувач активує режим видалення, він може вибрати конкретні логи для видалення або вказати діапазон, який потрібно видалити. Після підтвердження видалення, відповідний запит надсилається на сервер, і логи видаляються з бази даних.

Відфільтровані та відсортовані логи відображаються на сторінці з повідомленням про завантаження даних або відсутність логів, що забезпечує зрозумілий інтерфейс для взаємодії з логами. Якщо логи ще завантажуються, користувач бачить індикатор завантаження, що інформує його про поточний стан. Якщо ж логи відсутні, на сторінці з’являється відповідне повідомлення.

Завдяки цим функціям та особливостям, компонент LogsPage забезпечує потужний та гнучкий інструмент для управління логами, що є невід'ємною частиною ефективної роботи додатка.

Це дозволяє адміністраторам легко відстежувати події, діагностувати проблеми та підтримувати високу продуктивність системи. Схема потоку роботи даного компонента наведена на рисунку 3.5. Ця схема візуалізує процеси отримання, відображення та видалення логів, а також взаємодію різних компонентів та сервісів у межах компонента LogsPage.

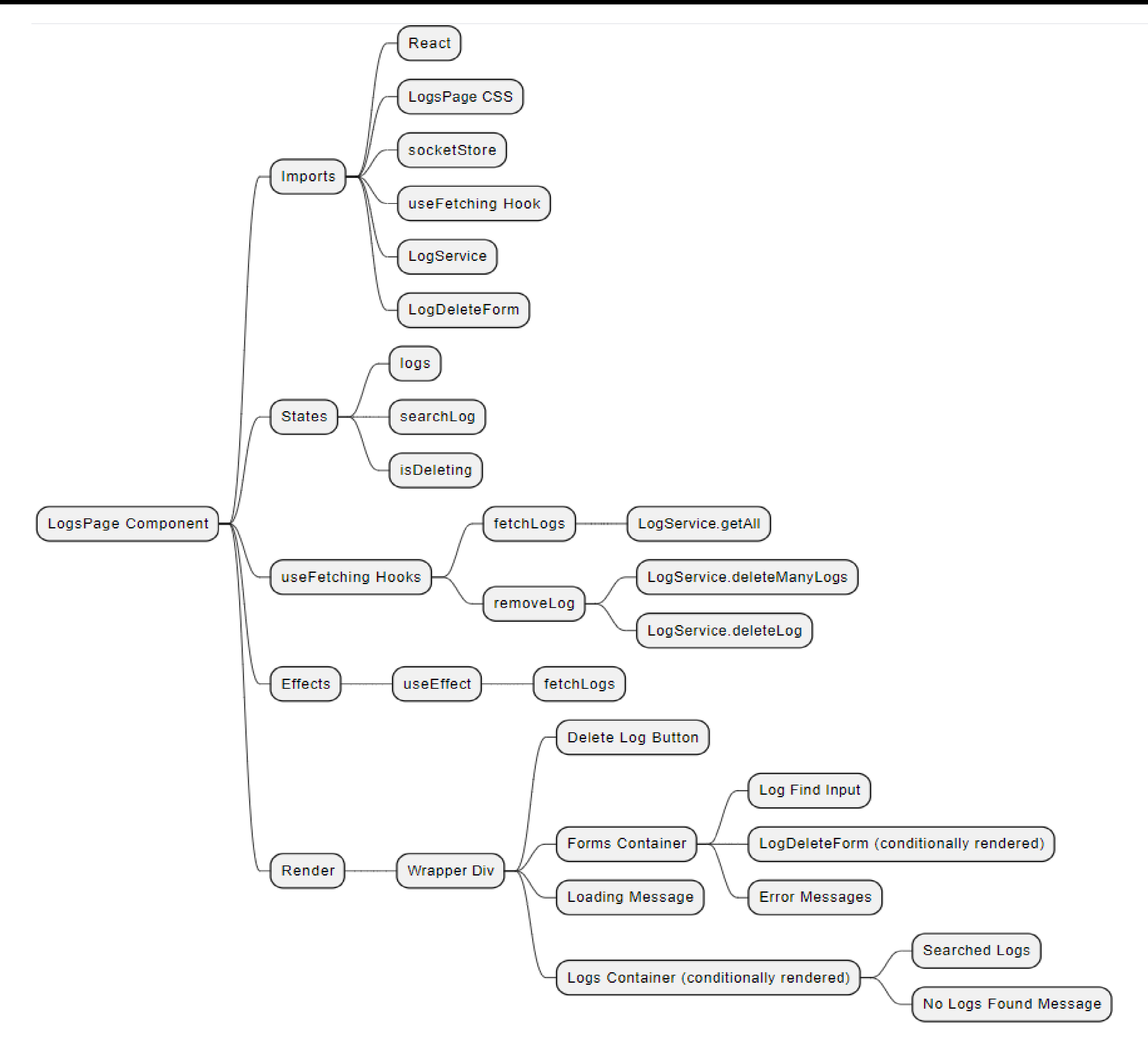


Рисунок 3.5 – схема потоку роботи компонента LogsPage

Компонент ConnectionIdPage відображає детальну інформацію про конкретне з'єднання, таку як системні характеристики, графіки запитів, завантаження процесора та мережевий трафік. Він дозволяє користувачам переглядати та аналізувати дані з'єднання у реальному часі.

Цей компонент використовує хук useParams для отримання параметрів маршруту, зокрема id з'єднання, яке буде відображено. Цей id являє собою ідентифікатор сокета працівника який під’єднаний до серверу. Компонент має також один локальний стан chartsVisible - об'єкт, що відстежує видимість різних графіків (запитів, завантаження процесора).

Сама інформація про працівника знаходиться у глобальному сховищі socketStore, і на основі id, яке передається через параметри маршруту, вона вибирається з масиву.

Для показу графіків використовуються кнопки, які створюються завдяки додатковій компоненті ShowChartButton. Відображає графіки запитів RequestChart та завантаження процесора CpuChart виноситься також в окремі.

Компоненти SystemInfoList використовуються для відображення системної інформації про з'єднання, включаючи інформацію про систему, процесор, операційну систему та пам'ять. Якщо підсумувати особливості даного компоненту то вони будуть виглядати наступним чином:

- детальна інформація про з'єднання: відображає детальну інформацію про конкретне з'єднання;
- графічні представлення: дозволяє відображати різні графіки, такі як графік запитів, завантаження процесора та мережевого трафіку;
- керування видимістю графіків: користувач може перемикати видимість графіків за допомогою кнопок;
- реальний час: дані оновлюються в реальному часі, що дозволяє користувачам бачити актуальну інформацію про з'єднання;
- інтерактивність: користувач може шукати та сортувати інформацію за допомогою інтерактивного інтерфейсу.

Даний компонент є ключовим у аналізі роботи конкретного клієнта, що дозволяє легко побачити стан та продуктивність з'єднання.

Аналітичні дані, представлені в компоненті, допомагають приймати обґрунтовані рішення щодо управління системою та її оптимізації під час роботи з конкретними з’єднаннями.

Схема потоку роботи компонента ConnectionIdPage наведена на рисунку 3.6. Ця схема візуалізує процеси збору, обробки та відображення даних про з'єднання, а також взаємодію різних підкомпонентів та модулів у межах ConnectionIdPage. Вона демонструє, як дані передаються від сервера до користувацького інтерфейсу та як користувачі можуть взаємодіяти з цими даними.

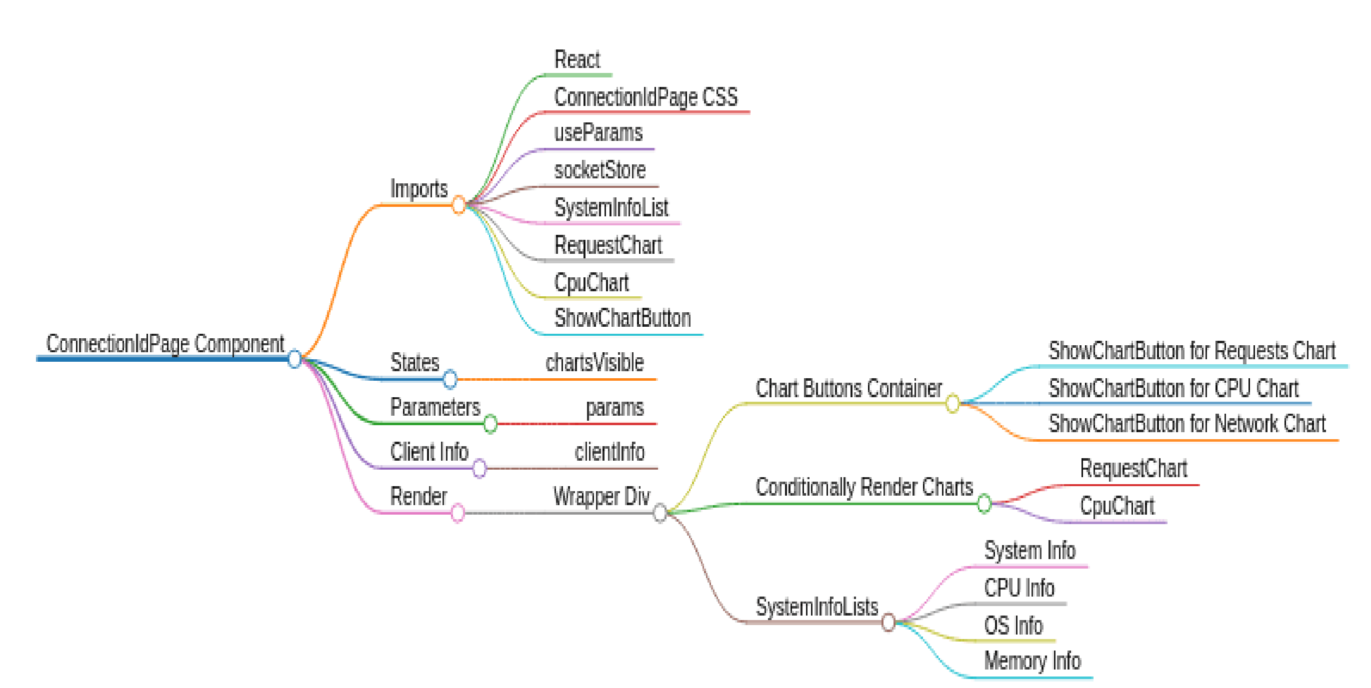


Рисунок 3.6 – схема потоку роботи компонента ConnectionIdPage

Останній дуже важливий компонент StatisticPage. Даний компонент виконує функції схожі на ConnectionIdPage. Різниця в тому що даний компонент виконує аналіз та збір даних про сам сервер. Якщо попередній компонент надавав можливість слідкувати за продуктивністю роботи клієнтів, то даний компонент надає можливість перевіряти роботу самого сервера. Особливості даного компонента наступні:

– агрегація даних: компонент відображає важливу статистичну інформацію про систему в одному місці;

– графіки в реальному часі: включає графіки для моніторингу мережевого трафіку, дискового вводу-виводу та використання пам'яті;

– користувацькі підкомпоненти: використовує спеціалізовані підкомпоненти, такі як NetworkChart, DiskChart та MemoryChart, для візуалізації різних аспектів системної статистики;

– модульність: легко розширювальний компонент завдяки використанню окремих підкомпонентів для кожного типу статистики;

– оновлення в реальному часі: дані оновлюються через певні інтервали часу, забезпечуючи актуальність інформації.

Якщо підсумувати вищесказане про даний компонент, то можна сказати що StatisticPage надає адміністраторам можливість отримати комплексну інформацію про стан системи в одному місці, що значно спрощує процес моніторингу та управління. Цей компонент відображає важливі показники системи, такі як мережевий трафік, використання дискових ресурсів та оперативної пам'яті, забезпечуючи цілісну картину поточного стану системи.

Візуалізація даних у вигляді графіків допомагає швидко ідентифікувати аномалії та проблеми в роботі системи, що сприяє оперативному реагуванню та вирішенню проблем. Інтерактивність графіків дозволяє користувачам взаємодіяти з даними, отримуючи більш детальну інформацію про кожен параметр.

Це забезпечує можливість глибокого аналізу використання ресурсів, що дозволяє виявляти неефективності та оптимізувати продуктивність системи. Життєвий цикл даного компонента зображено на рисунку 3.7.

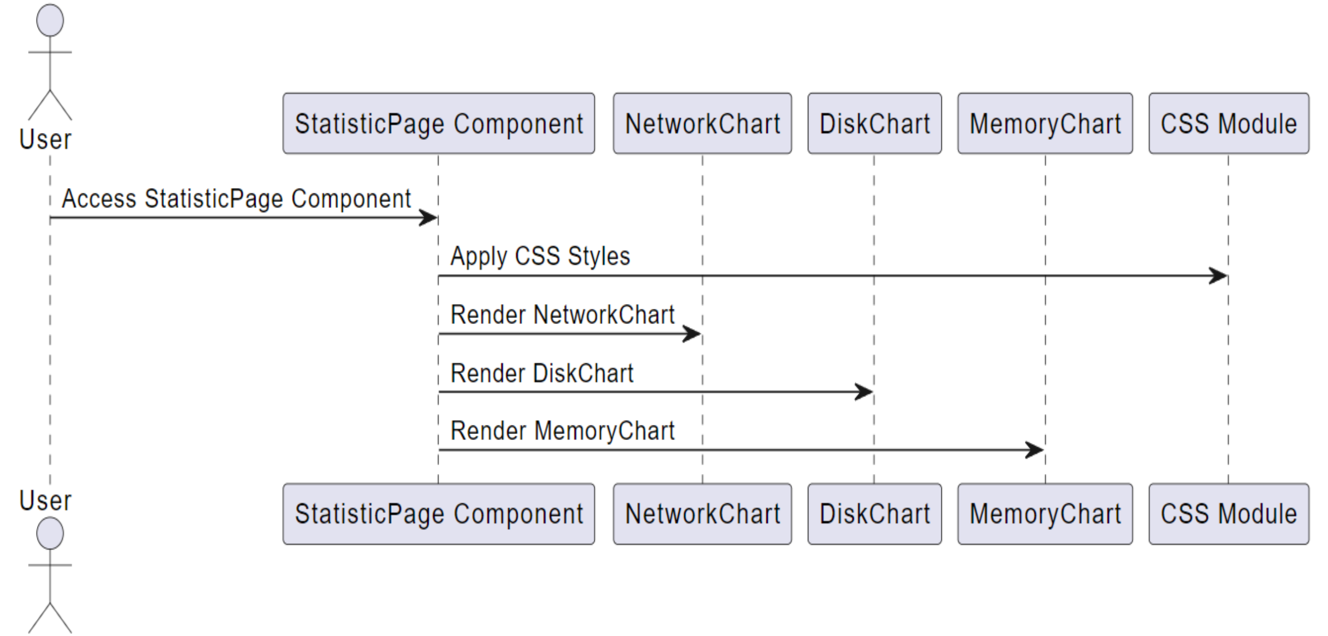


Рисунок 3.7 – схема життєвого циклу компонента StatisticPage

Завдяки модульному підходу, компонент легко адаптується до різних потреб користувачів.

Це означає, що його можна швидко розширювати для відображення додаткових метрик або інтегрувати з іншими системами моніторингу. Така

гнучкість забезпечує зручність використання та підтримки, а також дозволяє масштабувати рішення в міру зростання потреб бізнесу чи інфраструктури.

Висновки

Під час роботи над даним розділом у системі було впроваджено алгоритм Weighted Round Robin, який забезпечує рівномірний розподіл завдань між вузлами системи, враховуючи їхню продуктивність. Завдяки цьому вдалося значно підвищити загальну ефективність системи та знизити час обробки запитів. Алгоритм WRR дозволяє динамічно налаштовувати ваги для кожного вузла залежно від його поточної завантаженості, що забезпечує оптимальний розподіл ресурсів та мінімізує ризики перевантаження окремих вузлів.

Це рішення сприяє стабільній роботі системи навіть при високих навантаженнях та швидких змінах умов експлуатації.

Також було створено набір основних компонентів React, які разом складають ефективний та зручний додаток для моніторингу розподіленої системи. Це включає компоненти для управління з'єднаннями, логами та статистикою, що дозволяє адміністраторам ефективно стежити за станом системи та приймати обґрунтовані рішення. Дані компоненти були розроблені з урахуванням вимог до масштабованості та надійності, що забезпечує безперебійну та швидку роботу системи.

Реалізація сервера на базі Node.js з підтримкою HTTP та WebSockets дозволила створити гібридну комунікаційну архітектуру, що також забезпечує стабільну роботу системи в умовах динамічної зміни навантаження. Використання WebSockets дозволяє здійснювати двосторонню комунікацію в режимі реального часу, що є критичним для моніторингу стану системи.

Це рішення підвищило ефективність розподіленої обробки даних та моніторингу стану системи, забезпечуючи швидке реагування на зміни у навантаженні та оперативне оновлення даних.

ВИСНОВКИ

У цій дипломній роботі було розглянуто та реалізовано розподілену систему моніторингу та обробки даних з використанням сучасних веб-технологій. Основна увага приділялась створенню гібридної комунікаційної архітектури та реалізації ефективного алгоритму балансування навантаження.

Реалізація програми моніторингу системи включала використання Electron та React для створення зручного користувацького інтерфейсу, що забезпечує динамічне відображення даних у реальному часі. Було впроваджено різноманітні компоненти для управління з'єднаннями, логами та статистикою системи, що дозволяє адміністраторам ефективно стежити за станом системи та приймати обґрунтовані рішення. Ці компоненти були створені з урахуванням вимог до масштабованості та надійності, що забезпечує стабільну та оперативну роботу системи.

Створення гібридної комунікаційної архітектури передбачало реалізацію сервера на базі Node.js з підтримкою як HTTP, так і WebSockets. Використання бібліотеки socket.io забезпечило двосторонній зв'язок між сервером та клієнтами, що дозволяє ефективно передавати та отримувати дані в режимі реального часу. Це особливо важливо для додатків моніторингу, де оперативне оновлення інформації є критичним для прийняття своєчасних рішень. Крім того було впроваджено механізм шифрування даних за допомогою AES, що гарантує високий рівень безпеки при передачі даних між клієнтом та сервером.

Реалізація middlewares для обробки HTTP-запитів дозволила організувати структуровану та зручну для розширення обробку даних. Middlewares забезпечують можливість впровадження додаткових функцій, таких як автентифікація, логування, обробка помилок та інше, що робить архітектуру більш гнучкою та масштабованою. Кожен middleware виконує свою специфічну роль, забезпечуючи чистоту коду та полегшуючи підтримку і розвиток системи.

Застосування MongoDB для зберігання логів та статистичних даних дозволило ефективно керувати великими обсягами інформації. MongoDB, як документно-орієнтована база даних, забезпечує гнучкість у зберіганні та обробці

даних завдяки своїй структурі JSON-подібних документів. Це дозволяє легко масштабувати систему, додаючи нові вузли, коли це необхідно, без значних змін у кодовій базі чи архітектурі.

Реалізація алгоритму балансування навантаження включала створення спеціалізованого класу LoadBalancer, призначеного для управління підключеннями та розподілу завдань між різними компонентами системи. Цей клас був розроблений для забезпечення високої ефективності та надійності роботи системи, враховуючи всі необхідні параметри для оптимального розподілу навантаження.

Впроваджений алгоритм Weighted Round Robin (WRR) є одним з найпоширеніших методів балансування навантаження, який використовує ваги для визначення пріоритетності завдань. Кожному працівнику (вузлу системи) присвоюється вага, що відображає його продуктивність та потужність. Це дозволяє алгоритму динамічно розподіляти завдання таким чином, щоб максимально використовувати доступні ресурси. Алгоритм WRR працює за принципом циклічного обслуговування, де кожен працівник отримує завдання відповідно до своєї ваги. Працівники з вищими вагами отримують більше завдань, що дозволяє збалансувати навантаження і уникнути перевантаження менш потужних компонентів. Цей підхід забезпечує рівномірний розподіл обчислювальних ресурсів і підвищує загальну продуктивність системи.

Розроблена система демонструє високу ефективність та гнучкість у розподіленій обробці даних та моніторингу стану системи. Використання вищезгаданих технологій, таких як Electron, React, Node.js та MongoDB, дозволило створити інтуїтивно зрозумілий інтерфейс та забезпечити стабільну роботу системи. Система легко адаптується до змін та розширюється завдяки модульному підходу, що забезпечує її подальший розвиток і масштабування відповідно до потреб бізнесу чи інфраструктури. Запропоновані у цій роботі рішення та реалізовані механізми можуть бути використані у різних галузях для ефективного управління розподіленими системами та обробки великих обсягів даних у реальному часі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Pershin, I. M., T. V. Kukharova, and V. V. Tsapleva. Designing of distributed systems of hydrolithosphere processes parameters control for the efficient extraction of hydromineral raw materials. *Journal of Physics: Conference Series*. 2021.№1728.P. 1-6.
2. Aksenov, Vitaly, and Petr Kuznetsov. Review of the Third Summer School on the Practice and Theory of Distributed Computing SPTDC 2020. *ACM SIGACT News* 2021.№51.P. 75-84.
3. National Academies of Sciences, et al. Enhancing the resilience of the nation's electricity system. National Academies Press, 2017. 59 p.
4. Kassambara A. Practical guide to cluster analysis in R: Unsupervised machine learning. – Sthda, 2017. 147 p.
5. Sattler, Felix, Klaus-Robert Müller, and Wojciech Samek. Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints. *IEEE transactions on neural networks and learning systems* 2020.№32.P 3710-3722.
6. Li, Songhuan, Hong Jiang, and Mingkang Shi. Redis-based web server cluster session maintaining technology. *International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*. IEEE, 2017.№13.P. 34-74.
7. Yu, Anji, and Shimin Yang. Research on web server cluster load balancing algorithm in web education system. *The Journal of Supercomputing* 2020.№76.1.P. 3364-3373.
8. Raynal, Michel. Fault-tolerant message-passing distributed systems: an algorithmic approach. 2018. p 453.
9. Kraus, Klemens. Security management process in distributed, large scale high performance systems. 2014. 182 p.
10. Wang, Chen, et al. A general and fast distributed system for large-scale dynamic programming applications. *Parallel Computing* 2016.№60.P 1-21.
11. Hasan, Mohamad Kamrul, et al. Timing synchronization framework for wide area measurement system in smart grid computing. *Global Conference on Wireless and Optical Technologies (GCWOT)*. IEEE, 2020.№37.P 1-16.
12. Du, Wei, et al. Modeling of grid-forming and grid-following inverters for dynamic simulation of large-scale distribution systems. *IEEE Transactions on Power Delivery* 2020.№36.P 2035-2045.

13. Burns, Brendan. Designing distributed systems: patterns and paradigms for scalable, reliable services. O'Reilly Media, Inc., 2018. 145 p.

14. Deng, Shuiguang, et al. Burst load evacuation based on dispatching and scheduling in distributed edge networks. *IEEE Transactions on Parallel and Distributed Systems* 2021.№32.P 1918-1932.

15. Capitanescu, Florin, et al. Assessing the potential of network reconfiguration to improve distributed generation hosting capacity in active distribution systems. *IEEE Transactions on Power Systems* 2014.№30.P 346-356.

16. Dehghanpour, Kaveh, et al. A survey on state estimation techniques and challenges in smart distribution systems. *IEEE Transactions on Smart Grid* 2018.№10.P 2312-2322.

17. Zhao, Zhuoran, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2018.№37.P 2348-2359.

18. El-Sayed, Hesham, et al. Edge of things: The big picture on the integration of edge, IoT and the cloud in a distributed computing environment. *IEEE access* 6 2017.№6.P 1706-1717.

19. Teerapittayanon, Surat, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. *IEEE 37th international conference on distributed computing systems (ICDCS)*. IEEE, 2017.№37.P 235-274.

20. Neghabi, Ali Akbar, et al. Load balancing mechanisms in the software defined networks: *a systematic and comprehensive review of the literature*. *IEEE access* 6 2018.№6.P 14159-14178.

21. Mishra, Sambit Kumar, Bibhudatta Sahoo, and Priti Paramita Parida. Load balancing in cloud computing: a big picture. *Journal of King Saud University-Computer and Information Sciences* 2020.№32.P 149-158.

22. Milani, Alireza Sadeghi, and Nima Jafari Navimipour. Load balancing mechanisms and techniques in the cloud environments: Systematic literature review and future trends. *Journal of Network and Computer Applications* 2016.№71.P 86-98.

23. Ramezani, Fahimeh, Jie Lu, and Farookh Khadeer Hussain. Task-based system load balancing in cloud computing using particle swarm optimization. *International journal of parallel programming* 2014.№42.P 739-754.

24. Van Steen, Maarten, and Andrew S. Tanenbaum. Distributed systems. Leiden, The Netherlands: Maarten van Steen, 2017. 596 p.

25. Lin, Qingwei, et al. Log clustering based problem identification for online service systems. *Proceedings of the 38th International Conference on Software Engineering Companion*. 2016.№38.P 102-111.

26. Alaa, F. I. A. D., Mekkakia Maaza Zoulikha, and B. E. N. D. O. U. K. H. A. Hayat. Improved round robin scheduling algorithm with varying time quantum. *2020 second international conference on embedded & distributed systems (EDiS)*. IEEE, 2020.№1.P 43-49

27. Tamri, Rajae, Jilali Antari, and Radouane Iqdour. A New WRR Algorithm for an Efficient Load Balancing System in IoT Networks under SDN. *International Journal of Interactive Mobile Technologies* 2024.№18.P 1-45.

28. Tychalas, Dimitrios, and Helen Karatza. An advanced weighted round robin scheduling algorithm.*Proceedings of the 24th Pan-Hellenic Conference on Informatics*. 2020.№24.P 188-191.

29. Cantelon, Mike, et al. *Node. js in Action*. Greenwich: Manning, 2014. 392 p.

30. Zhu, Jiapeng, et al. Node. js scalability investigation in the cloud. *CASCON*. ACM, 2018.№28.P 201-212.

31. Davis, James, Gregor Kildow, and Dongyoon Lee. The case of the poisoned event handler: Weaknesses in the Node. js event-driven architecture. *Proceedings of the 10th European Workshop on Systems Security*. 2017.№6.P 1-6.

32. Sun, Haiyang, et al. Efficient dynamic analysis for Node. js. *Proceedings of the 27th International Conference on Compiler Construction*. 2018.№27.P 23-41.

33. Chaniotis, Ioannis K., Kyriakos-Ioannis D. Kyriakou, and Nikolaos D. Tselikas. Is Node. js a viable option for building modern web applications? *A performance evaluation study*. 2015.№97.P 1023-1044.

34. Mukhiya, Suresh Kumar, and Hoang Khac Hung. An Architectural Style for Single Page Scalable Modern Web Application. *International Journal of Recent Research Aspects* 2018.№5.P 21-42.

35. McFarlane, Timo. Managing State in React Applications with Redux. 2019. 41 p.

36. Murley, Paul, et al. Websocket adoption and the landscape of the real-time web. *Proceedings of the Web Conference*. 2021.№1.P 1-18

37. Seufert, Michael, et al. A survey on quality of experience of HTTP adaptive streaming. *IEEE Communications Surveys & Tutorials* 2014.№17.P 469-492.

38. Hong, Kiwon, et al. SDN-assisted slow HTTP DDoS attack defense method. *IEEE Communications Letters* 2017.№22.P 688-691.

39. Ogundeyi, K. E., and C. Yinka-Banjo. WebSocket in real time application. *Nigerian Journal of Technology* 2019.№38.P 1010-1020.

40. Liu, Wen Tao. Research on the development of websocket server. *Advanced Materials Research* 2014.№886.P 694-697.

41. Łasocha, Wojciech, and Marcin Badurowicz. Comparison of WebSocket and HTTP protocol performance. *Journal of Computer Sciences Institute* 2021.№19.P 67-74.

42. Buluş, Ayşin, and Ercan Buluş. Cipher with AES. *International Conference on Computer Science and Engineering (UBMK)*. IEEE, 2018.№3.P 1-36

43. Noorbasha, Fazal, et al. Design of AES based chiper and decipher cryptography system using Verilog HDL. *Journal of Physics: Conference Series*, 2021.№1804.P 1-6.

44. Devi, Loly Puspa. "Implementasi Algoritma Algoritma Vigenere Chiper Dan Advanced Encryption Standart (Aes) Untuk Keamanan Data Teks." *Bulletin Information Systems* 2023.№1.P 25-29.

45. Bradshaw, Shannon, Eoin Brazil, and Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, 2019. 475 p.

46. Abbes, Hanen, and Faiez Gargouri. Big data integration: A MongoDB database and modular ontologies based approach. *Procedia Computer Science* 2016.№96.P 446-455.

47. Effendy, Faried, and Bramantyo Adhilaksono. Performance comparison of web backend and database: A case study of node. js, Golang and MySQL, Mongo DB. *Recent Advances in Computer Science and* 2021.№14.P 1955-1961.

48. Patil, Mayur M., et al. A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing—Sharding in MongoDB and its advantages. *International Conference on I-SMAC*. IEEE, 2017.№1.P 132-142.

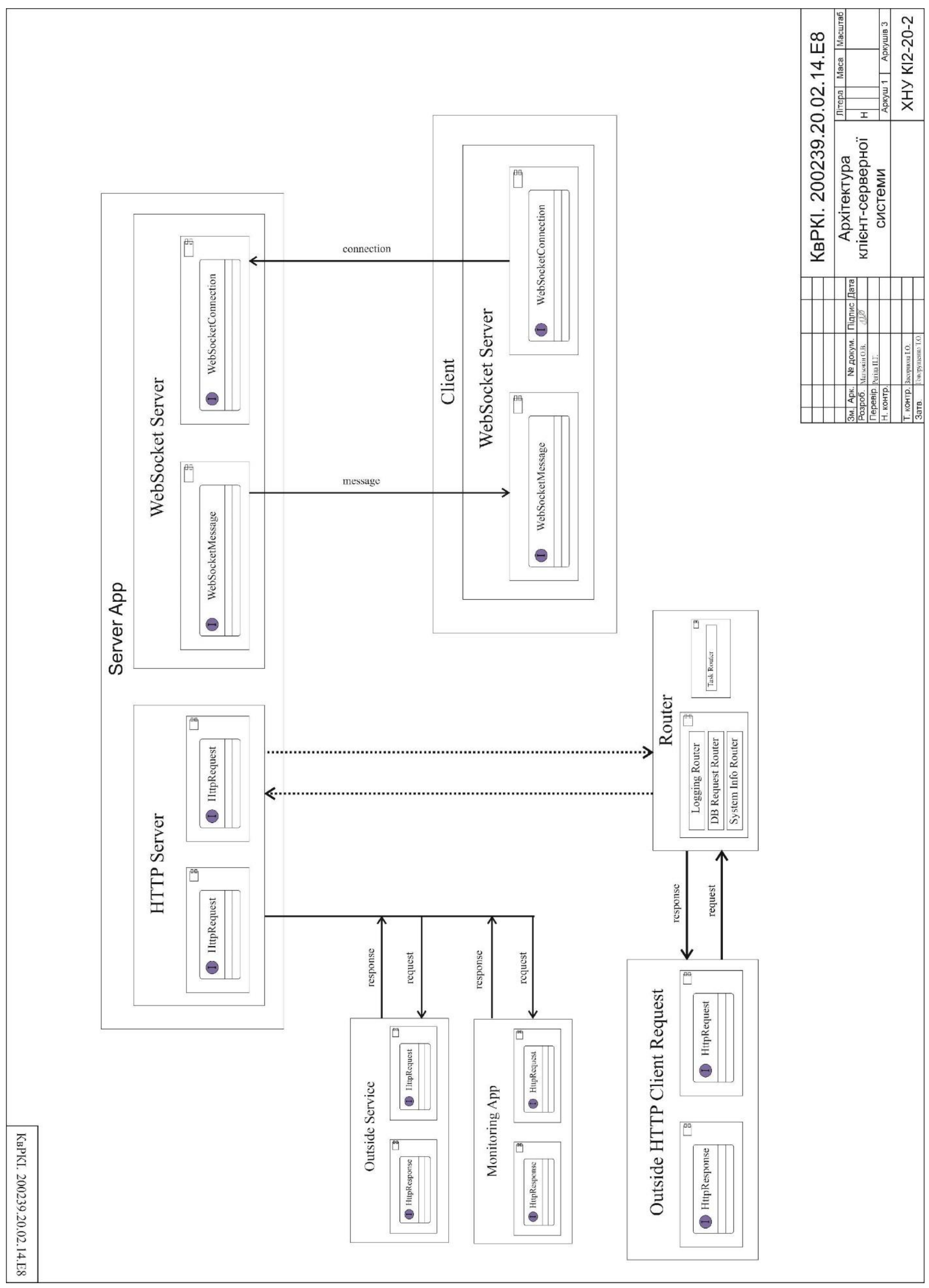
49. Karpathiotakis, Manos, c Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proceedings of the VLDB Endowment* 2016.№9.P 972-983.

50. Eyada, Mahmoud Moustafa, et al. Performance evaluation of IoT data management using MongoDB versus MySQL databases in different cloud environments. *IEEE access,* 2020.№8.P 110656-110668.

ДОДАТОК А

(обов’язково)

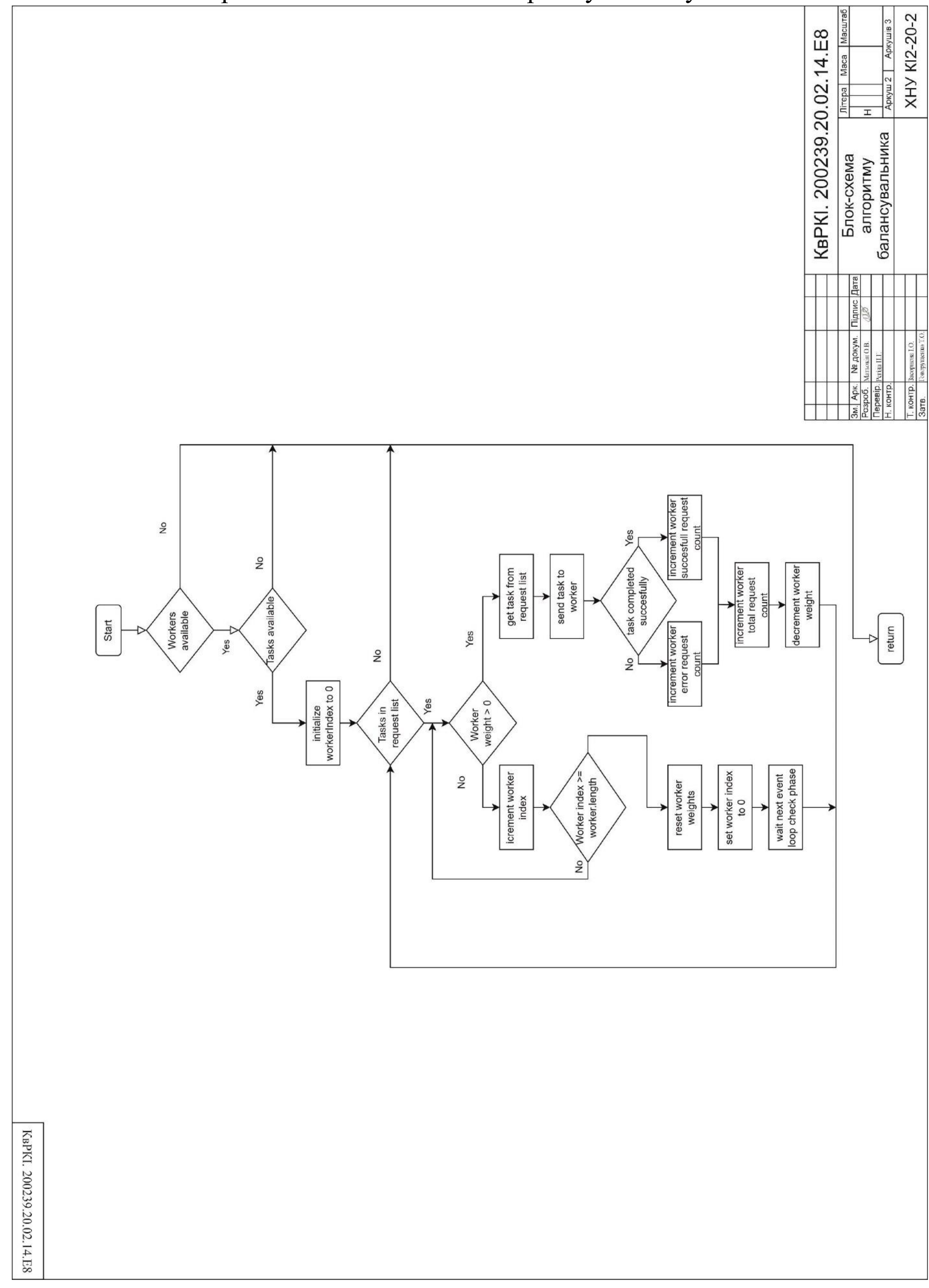
Копія креслення архітектури клієнт-серверної системи



ДОДАТОК Б

(обов’язково)

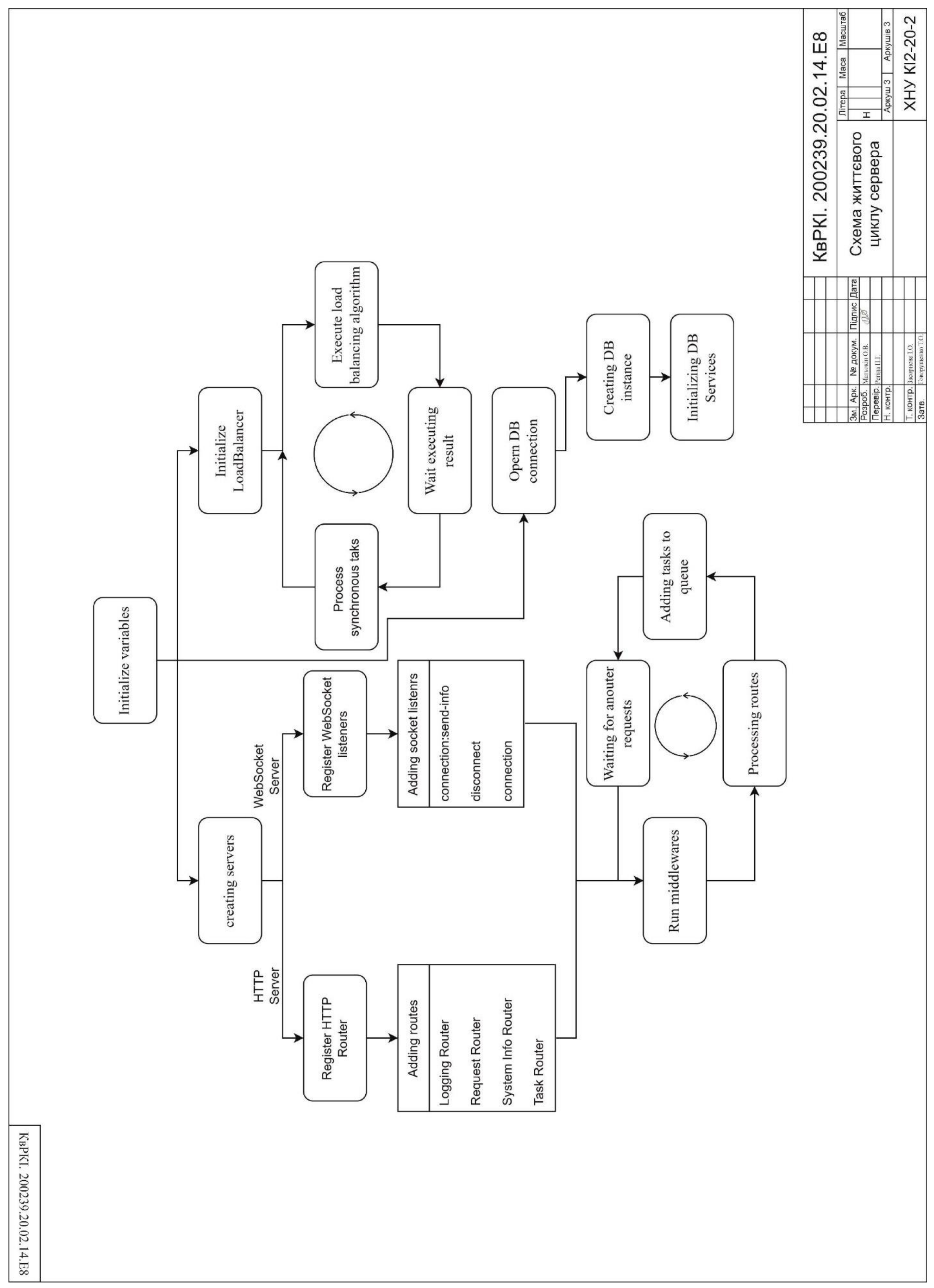
Копія креслення блок-схеми алгоритму балансувальника



ДОДАТОК В

(обов’язково)

Копія креслення схеми життєвого циклу сервера



ДОДАТОК Г

(обов’язково)

Код балансувальна

```
const path = require('path');
const { decrypt, encrypt } = require('./functions/cryptoUtils');
class LoadBlancerWRR {
  workers = [];
  adminNamespace = undefined;
  sockets = new Map();
  requestList = [];
  io = undefined;
  sendingRequestInterval = undefined;

  setAdminNamespace(namespace) {
    this.adminNamespace = namespace;
  }
  removeWorker(socketId) {
    let workerIndex = undefined;
    this.workers.find((worker, index) => {
      if (worker.socketId == socketId) {
        workerIndex = index;
        return true;
      }
    });
    this.workers.splice(workerIndex, 1);
  }
  addSocket(socket) {
    this.sockets.set(socket.id, socket);
  }
```

```
addRequests(requests) {
  this.requestList.push(...requests);
}
addSingleRequest(request) {
  this.requestList.push(request);
}
addWorker(worker) {
  this.workers.push(worker);
}
setSocektIoInstance(io) {
  this.io = io;
}
startSendingRequestInfo() {
  this.sendingRequestInterval = setInterval(() => {
    for (const worker of this.workers) {
      this.adminNamespace.emit('admin:workerRequest', {
        workerId: worker.socketId,
        requests: worker.requests,
      });
    }
  }, 2000);
}

stopSending() {
  clearInterval(this.sendingRequestInterval);
}
async sendTaskToWorker(index, task) {
  this.sockets
    .get(this.workers[index].socketId)
```

```
    .emit('request:task', task, (responce) => {
      if (responce.status == 'ok') {
        this.workers[index].requests.totalSuccessfulRequests += 1;
        console.log(responce.payload);
      }
      if (responce.status == 'error') {
        this.workers[index].requests.totalErrorRequests += 1;
      }
    });
}

async WRR() {
  if (!this.workers.length) {
    return;
  }
  if (!this.requestList.length) {
    return;
  }

  let workerIndex = 0;
  while (this.requestList.length) {
    while (this.workers[workerIndex].weight <= 0) {
      console.dir(this.workers);
      workerIndex++;
      if (workerIndex >= this.workers.length) {
        for (let index = 0; index < this.workers.length; index++) {
          this.workers[index].weight = this.workers[index].originalWeight;
        }
        workerIndex = 0;
```

```
          await new Promise((resolve) => setImmediate(resolve));
          break;
        }
      }
      const task = this.requestList.shift();
      if (task == undefined) {
        console.error('Task undefiner error');
        return;
      }

      this.sendTaskToWorker(workerIndex, task);

      this.workers[workerIndex].requests.totalRequestCount += 1;

      this.workers[workerIndex].weight--;
    }
  }
}
module.exports = new LoadBlancerWRR();
```

Код маршрутизаторів

```
const { parseToDateTimeFormat } = require('../functions/dateFormtter');
const store = require('../store');
module.exports = function (db = undefined) {
  return async function (request, response) {
    const dbName = 'dest-server';
    const collectionName = 'logs';
    if (
```

```
  request.params.pathname == '/logs/date-provided' &&
  request.method == 'GET'
) {
  if (db) {
    const logs = await db.getItemsByDate(
      dbName,
      collectionName,
      request.params.searchParams.get('date'),
    );
    response.send([...logs]);
  }
}
if (request.params.pathname == '/logs' && request.method == 'GET') {
  if (db) {
    const logs = await db.getAllItems(dbName, collectionName);
    response.send([...logs]);
  }
}
if (request.method === 'OPTIONS') {
  response.writeHead(200, {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE',
    'Access-Control-Allow-Headers': 'Content-Type, Authorization',
  });
  response.end();
  return;
}
if (
  request.params.pathname == '/logs/remove' &&
```

```
    request.method == 'DELETE'
  ) {
    const deleteType = request.params.searchParams.get('type');
    console.log(deleteType);
    if (deleteType == 'single') {
      const result = await db.removeSingleItemByDate(
        dbName,
        collectionName,
        request.params.searchParams.get('date'),
      );
      response.send(result);
    } else if (deleteType == 'range') {
      const result = await db.removeManyItemsByDate(
        dbName,
        collectionName,
        request.params.searchParams.get('date'),
        request.params.searchParams.get('endDate'),
      );
      response.send(result);
    }
  }
  if (request.params.pathname == '/logs/test' && request.method == 'POST') {
    if (db) {
      store.addLog('Test Log');

      response.send({ message: 'Test log added' });
    }
  }
};
```

```
};
const si = require('systeminformation');
const {
  gatherSystemMetrics,
  gatherNetworkTrafficData,
  gatherDisksData,
  gatherMemoryData,
  gatherCpuLoadData,
} = require('../functions/systemInfoUtils');
function getSystemInfoConfig(key, confParams) {
  return { usageConfig: { [key]: confParams } };
}

module.exports = function (io) {
  return async function (request, response) {
    if (
      request.params.pathname == '/system/cpu-usage' &&
      request.method == 'GET'
    ) {
      const socketId = request.params.searchParams.get('socketId');
      const worker = io.sockets.sockets.get(socketId);
      worker.emit(
        'system:cpu-usage',
        getSystemInfoConfig('currentLoad', 'currentLoad'),
        (callbackResponse) => {
          response.send(callbackResponse);
        },
      );
    }
```

```
if (
  request.params.pathname == '/system/server' &&
  request.method == 'GET'
) {
  try {
    const data = await gatherSystemMetrics();
    response.send(data);
  } catch (error) {
    console.error('Failed to gather system information:', error);
  }
}
if (
  request.params.pathname == '/system/server-network' &&
  request.method == 'GET'
) {
  try {
    const data = await gatherNetworkTrafficData();
    response.send(data);
  } catch (error) {
    console.error('Failed to gather system information:', error);
  }
}
if (
  request.params.pathname == '/system/server-disk' &&
  request.method == 'GET'
) {
  try {
    const data = await gatherDisksData();
    response.send(data);
```

```
      } catch (error) {
        console.error('Failed to gather system information:', error);
      }
    }
    if (
      request.params.pathname == '/system/server-memory' &&
      request.method == 'GET'
    ) {
      try {
        const data = await gatherMemoryData();
        response.send(data);
      } catch (error) {
        console.error('Failed to gather system information:', error);
      }
    }
  };
};

const loadBalancer = require('../LoadBalancerWRR');
const {
  generateSlar,
  getRandomInt,
  generateDiagonalSlar,
} = require('../functions/slarUtils');
module.exports = function (db = undefined) {
  return async function (request, response) {
    if (
      request.params.pathname == '/task/generate' &&
      request.method == 'GET'
```

```
    ) {
      const systems = [];
      const isRandom = request.params.searchParams.get('random') === 'true';
      const equationAmount =
        request.params.searchParams.get('equationAmount') ||
        (isRandom ? getRandomInt(10) : 3);
      const systemAmount =
        request.params.searchParams.get('systemAmount') ||
        (isRandom ? getRandomInt(10) : 1);
      const diagonal = request.params.searchParams.get('diagonal') === 'true';
      for (let index = 0; index < systemAmount; index++) {
        if (diagonal) {
          systems.push(generateDiagonalSlar(equationAmount));
        } else {
          systems.push(generateSlar(equationAmount));
        }
      }
      loadBalancer.addRequests(systems);
      return response.send(systems);
    }
  };
};
```

Ім'я користувача:
Кафедра КІ

Дата перевірки:
04.06.2024 23:19:29 EEST

Дата звіту:
05.06.2024 07:50:10 EEST

ID перевірки:
1016321105

Тип перевірки:
Doc vs Internet + Library

ID користувача:
100005591

Назва документа: Матьокін_Балансувальник задач для динамічної розподіленої системи обчислень

Кількість сторінок: 71 Кількість слів: 16757 Кількість символів: 126033 Розмір файлу: 1.13 MB ID файлу: 1016119468

2.19% Схожість

Найбільша схожість: 0.86% з Інтернет-джерелом (https://dou.ua/forums/topic/31698/?from=similar_posts_tech)

2.01% Джерела з Інтернету | 79 .... Сторінка 73

0.6% Джерела з Бібліотеки | 41 .... Сторінка 74

0% Цитат

Не знайдено жодних цитат

Посилання | 1 .... Сторінка 74

0% Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 3

Wed Jun 05 06:56:19 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: 128362 Назва: БКР Балансувальник задач для динамічної розподіленої системи обчислень Додано в БД: 2024-06-05 Автора: О. В. Матьокін Керівники: П.Г Регіда Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	116929	906	210 (0%)	4 (0%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Матьокін Олег Вячеславович

Тема: Балансувальник задач для динамічної розподіленої системи обчислень

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 кваліфікаційної роботи було розроблено серверний додаток, який виконує балансування завдань для елементів розподіленої системи обчислень. В якості платформи програмування було використано NodeJS, а для комунікації між клієнтом та сервером використовувалися протоколи HTTP та WebSockets. Також було розроблено додаток для моніторингу стану системи.

2. Висновок про відповідність роботи 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, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому, теоретичному, розділі дипломного проекту якісно та в повній мірі розглянуті методи вирішення задач балансування навантажень у розподілених системах обчислень. У наступному розділі було здійснено обґрунтування вибору програмних засобів для реалізації алгоритму балансування. У основній проектній частині диплому було представлено реалізацію ефективний алгоритм балансування завдань для розподіленої системи обчислень. Було реалізовано програмний серверний додаток для організації гібридної системи, а також реалізовано програму для моніторингу стану системи. В загальному усі розділи відповідають завданню та містять сучасні методи вирішення поставлених завдань.

4. Позитивні сторони роботи: Дипломний проект відповідає вимогам кваліфікаційної роботи з розподілених системи обчислень. Дана робота демонструє

реалізований алгоритм балансування розподілених обчислень завдяки використанню сучасних технологій, таких як NodeJS для серверної частини та React, Zustand, Electron у додатку для моніторингу. Система має високу адаптивність та масштабованість. Проект вирізняється простотою інтеграції та використання, що дозволяє легко впроваджувати його у різні обчислювальні середовища та інфраструктури.

5. Негативні сторони роботи: Частково реалізована обробка ситуацій непередбачуваного закриття мережевого сокету, що може призвести до втрати даних.

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 (A).

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

доцент кафедри АКІТта Р Кофецька Л.О.

"05" 06 2024 р. (підпис)

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

Матьокіна Олега 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: Балансувальник задач для динамічної розподіленої системи

Автор: Матьокін Олег 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 допрацьована 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 оформленні посилання;
3) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 1-41 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 2.19% і адресується до 120 першоджерел, 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 КІС ______ Т. О. Говорущенко