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

Інститут телекомунікаційних систем

Кафедра Інформаційно-телекомунікаційних мереж

До захисту допущено:

В.о. завідувача кафедри

_____ Лариса ГЛОБА

«___»_____2021 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформаційно-комунікаційні технології»

спеціальності 172 «Телекомунікації та радіотехніка»

на тему: «Метод підвищення продуктивності сервера за рахунок залучення незадіяних ресурсів пристроїв в мережі»

Виконав:
студент IV курсу, групи ТІ-71
Сегеда Сергій Андрійович _____

Керівник:
Доцент кафедри ІТМ ІТС
Алєксєєв Микола Олександрович _____

Рецензент:
Доцент кафедри ТС ІТС
Осипчук Сергій Олександрович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент_____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут телекомунікаційних систем
Кафедра Інформаційно-телекомунікаційних мереж

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 172 «Телекомунікації та радіотехніка»
Освітньо-професійна програма «Інформаційно-комунікаційні технології»

ЗАТВЕРДЖУЮ
В.о.завідувача кафедри
_____ Лариса ГЛОБА
«___»_____2021 р.

ЗАВДАННЯ
на дипломну роботу студенту
Сегеді Сергію Андрійовичу

1. Тема роботи «Метод підвищення продуктивності сервера за рахунок залучення незадіяних ресурсів пристроїв в мережі», керівник роботи доцент кафедри інформаційно-телекомунікаційних мереж ІТС Алєксєєв Микола Олександрович, затверджені наказом по університету від «14» квітня 2021 р. № 1007-с

2. Термін подання студентом роботи 7 червня 2021 р.

3. Вихідні дані до роботи

1. Корпоративна мережа
2. Незадіянні ресурси пристроїв

4. Зміст роботи

1. Сутність та поняття технології. Огляд хмарних сервісів та їх основних моделей.
2. Аналіз існуючих рішень для оркестрації серверів.

3. Розробка методу для балансування навантаження між серверами в кластері.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

Плакат №1 (слайд) Тема роботи;
Плакат №2 (слайд) Актуальність дослідження;
Плакат №3 (слайд) Моделі хмарних сервісів;
Плакат №4 (слайд) Мета та задачі дослідження;
Плакат №5 (слайд) Наявність вільних ресурсів в системі;
Плакат №6 (слайд) Існуючі рішення та недоліки;
Плакат №7 (слайд) Постановка практичної задачі;
Плакат №8 (слайд) Архітектура рішення;
Плакат №9 (слайд) Діаграма станів;
Плакат №10 (слайд) Практичні результати;
Плакат №11 (слайд) Тест зі збільшеним навантаженням;
Плакат №12 (слайд) Висновки по роботі;

6. Дата видачі завдання 24.10.2020_____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Отримання завдання	24.10.2020	виконано
2.	Збір інформації	01.02.2021	виконано
3.	Аналіз сфери хмарних сервісів	20.02.2021	виконано
4.	Порівняння та вибір технологій для розробки	07.03.2021	виконано
5.	Розробка практичного методу	18.05.2021	виконано
6.	Аналіз результатів	25.05.2021	виконано
7.	Оформлення дипломної роботи	30.05.2021	виконано
8.	Отримання допуску до захисту	07.06.2021	виконано

Студент Сергій СЕГЕДА

Керівник Микола АЛЄКСЄЄВ

РЕФЕРАТ

Дипломна робота «Метод підвищення продуктивності сервера за рахунок залучення незадіяних ресурсів пристроїв в мережі» складається з переліку умовних скорочень, вступу, основної частини, що містить 3 розділи, висновків і списку використаних джерел. Загальний обсяг роботи – n сторінок. Робота містить 24 рисунків та 1 таблицю. Список використаних джерел включає 25 одиниць.

Актуальність: робота присвячена актуальній проблемі ефективного виконання веб-сервісів у кластерному середовищі. На даний момент існують рішення з оркестрації веб-сервісів у кластері, але вони потребують встановлення додаткових програмних засобів на робочих вузлах і призначені насамперед для кластерів зі стаціонарною архітектурою, у той час як використання ресурсів пристроїв, що підключені до корпоративної мережі, передбачає її досить часті зміни. У роботі пропонується підхід на основі використання балансувальника навантаження Nginx, який дозволяє задіяти вільні ресурси пристроїв під керуванням різних операційних систем ОС Windows, Linux, macOS, що підключені до мережі. Ефективність такого рішення оцінена за допомогою засобів для тестування пропускної здатності сервера Apache Bench.

Мета роботи: запропонувати підхід до оркестрації веб-сервісів у кластері, який дозволяє задіяти вільні ресурси пристроїв під керуванням різних операційних систем ОС Windows, Linux, macOS, що підключені до мережі та не потребує встановлення додаткового програмного забезпечення на робочі вузли.

Ключові слова: Кластерне середовище, балансувальник навантаження, Nginx, Apache Bench.

ABSTRACT

Thesis "A method for server performance increasing by engaging unused resources of network connected devices" consists of a list of abbreviations, introduction, main part, containing 3 sections, conclusions and a list of sources used. Total volume of work is n pages. The work contains 24 illustrations and 1 tables. The list of sources used includes 25 units.

Relevance: the relevance of the study is that the number of services on the Internet is growing every day. Sufficient computing resources must be provided to serve customers with the appropriate level of quality. Against the background of rising prices for cloud services, it is advantageous to try to attract devices on the network for maintenance.

Currently, there are solutions for orchestrating web services in a cluster, but they are designed primarily for clusters with a fixed architecture, while the use of resources of devices connected to the corporate network involves its fairly frequent changes.

Purpose: the purpose is to offer an approach based on the use of Nginx load balancer, which allows you to use available resources of devices connected to the network and running various operating systems such as Windows, Linux and macOS. The effectiveness of this solution is evaluated by using the tools to test the bandwidth of the server such as Apache Bench.

Keywords: Cluster environment, load balancer, Nginx, Apache Bench.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ПЗ	Програмне забезпечення;
AWS	Amazon Web Services, платформа для хмарних обчислень
IaaS	Infrastructure as a service модель обслуговування “інфраструктура як послуга
SaaS	Software as a service, модель обслуговування “програмне забезпечення як послуга”
PaaS	Platform as a service, модель обслуговування “платформа як послуга”
ПК	персональний комп’ютер
CPU	Central processing unit, функціональна частина комп'ютера, що призначена для інтерпретації команд
HTTP	Hypertext Transfer Protocol, протокол прикладного рівня для передачі гіпермедійних документів
API	Application programming interface, набір чітко визначених методів для взаємодії різних компонентів
JVM	Java Virtual Machine, механізм, що забезпечує середовище виконання для керування кодом Java або додатками
ООП	об’єктно-орієнтоване програмування
ОС	операційна система
REST	Representational state transfer, підхід до проектування мережевих протоколів
ECR	Elastic Container Registry, сховище для зображень Docker від AWS
CGI	Common Gateway Interface , стандарт зв’язку програми з веб сервером

ВСТУП

На сьогоднішній день, коли кількість сервісів в інтернеті постійно зростає, проблема продуктивності серверів, які їх обслуговують, стоїть дуже гостро. Проте, маючи необмежені фінансові ресурси, цю проблему досить просто вирішити на основі хмарних рішень. Так, наприклад, Amazon Web Services дозволяє побудувати кластер майже будь якої потужності, який буде автоматично масштабуватись, виходячи з потреб користувача. Але, коли необхідність у такому кластері епізодична, а у користувача є доступ до корпоративної мережі з підключеними пристроями, більш вигідним може бути залучення їх ресурсів, які не використовуються. Зазвичай, в компаніях та організаціях обчислювальні можливості серверного обладнання та комп'ютерів користувачів задіяні в робочі процеси не повному обсязі. Значні проміжки часу частина обладнання не використовується, тобто «простоює».

Отже, у роботі вирішується задача управління та задіяння ресурсів пристроїв корпоративної мережі, що утворюють собою кластер до якого є можливість їх динамічно додавати або видаляти. Вказане вирішується зміною конфігураційного файлу кластеру та налаштовуванням балансувальника. Автоматичне розміщення, координація та управління складними комп'ютерними системами та службами має назву оркестрація. Оркестрація описує, які сервіси повинні взаємодіяти між собою, використовуючи обмін повідомленнями, бізнес-логіку та послідовність дій.

Таким чином, ***об'єктом досліджень*** є оркестрування веб-сервісів у кластері.

Предмет досліджень – принцип роботи балансувальників навантаження в корпоративній мережі.

Мета досліджень – аналіз існуючих рішень та порівняння з власною розробкою.

Наукова новизна дослідження – розробка методу оркестрування незадіяних ресурсів пристроїв, підключених до корпоративної мережі в кластері.

РОЗДІЛ 1
СУТНІСТЬ ТА ПОНЯТТЯ ТЕХНОЛОГІЇ. ОГЛЯД ХМАРНИХ СЕРВІСІВ ТА ЇХ ОСНОВНИХ МОДЕЛЕЙ.

1.1. Сучасна ситуація в області хмарних сервісів

На сьогоднішній час потреба в обчислювальних ресурсах значно зросла, так як багато галузей визнали, що послуги надані через інтернет можуть надати значні переваги споживачам, підприємствам і державним установам. З огляду на стрімкий ріст кількості сервісів та забезпечення необхідності їх продуктивності, як ніколи вигідно залучити додаткові ресурси пристроїв, які «простоюють».

Одним із напрямків збільшення обчислювальних можливостей є використання хмарних сервісів. Згідно з дослідженням видання Facts and Factors, глобальний ринок хмарних обчислень оцінювався у 321 млрд. Доларів США у 2019 році, і очікується, що він досягне 1025,9 мільярда доларів США до кінця 2026 року. Очікується, що глобальний ринок хмарних обчислень буде рости зі складеними річними темпами зростання на 18 % з 2019 по 2027 рік [1].

Це показує, що плани компаній інвестувати в хмарні рішення тільки збільшуються та розробка не збирається зупинятись. Залучення фінансів таких масштабів і настільки велика мережа пристроїв стала можливою завдяки відносній простоті в налаштуванні і використанні, формуванню цін, програмному забезпеченню, та зручності таких систем. Адже завдяки цьому без жодних завад придбати у власне користування може будь яка особа.

Зазвичай хмарні ресурси розрізняють за їх моделлю (рисунок. 1.1). В даний час існує три найбільш поширені моделі [2]:

– IaaS. По суті, це апаратні засоби та ресурси (мережа, процесор, сховище), але розташовані в хмарі. Зазвичай це віртуальне оточення, створене з використанням технології віртуалізації. Поява IaaS відкрило нові

сфери для бізнесу, що дозволило компаніям скоротити витрати на IT-інфраструктуру.

– SaaS. Ці послуги надають програмне забезпечення, яке не потрібно завантажувати та встановлювати на ПК. Щоб використовувати необхідні функції, просто відкрийте Інтернет-ресурси. Приклади технологій SaaS: електронна пошта, зберігання файлів, онлайн-карти, онлайн-редактори тощо.

– PaaS. Це повноцінна онлайн-платформа з інструментарієм та середовищем для розробки. Більшість "звичайних" користувачів не цікавляться цим видом послуг. PaaS в основному використовується розробниками програмного забезпечення для прискорення та оптимізації розробки, запуску та управління додатками. По суті, PaaS є проміжною моделлю між ресурсами SaaS та IaaS. Прикладом PaaS платформи виступає AWS.

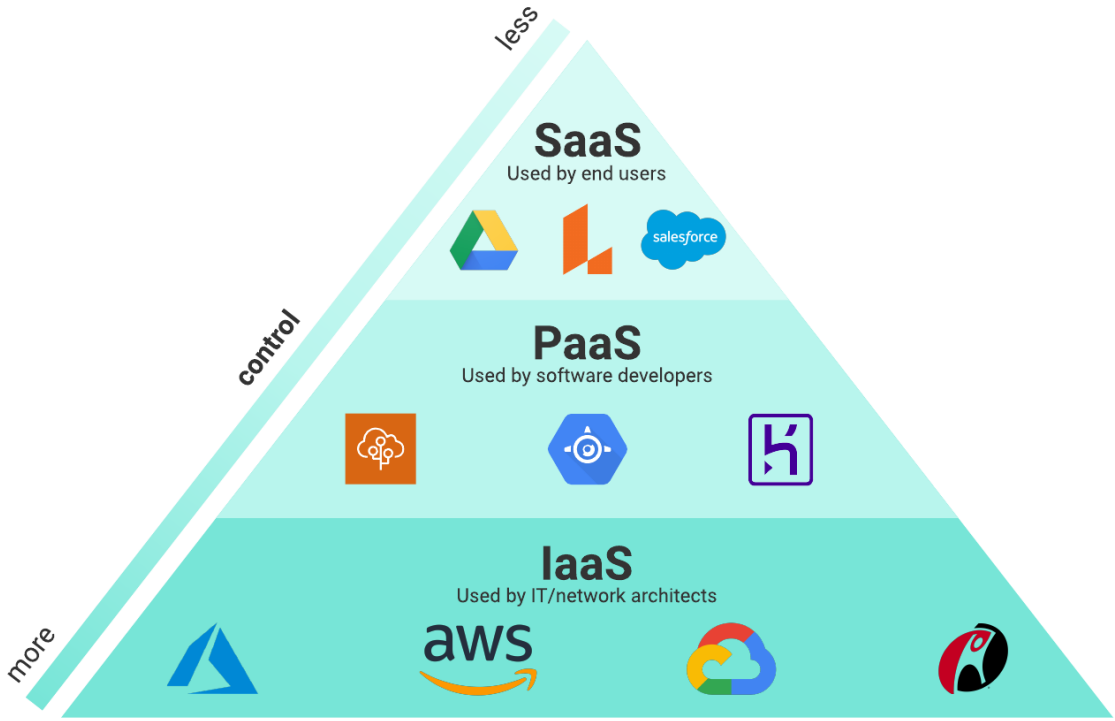

Рис.1.1 Види моделей хмарних сервісів та приклади

Кожна описана модель хмарного сервісу має свою аудиторію. Ресурси SaaS призначені для великої кількості користувачів ПК, вони зручні, зрозумілі та доступні кожному. Хмари PaaS по суті цікавлять IT-фахівців, розробників програмного забезпечення, розробників ігор, молодих стартапів та студентів. Користувачами рішень IaaS є переважно великі Інтернет-компанії, мережі та провайдери послуг [3].

1.2. Проблеми хмарних сервісів

Основною проблемою розглянутих вище хмарних сервісів є їх ціна [4]. Так, на найпопулярнішому сервісі з хмарних рішень AWS, оренда комп'ютера з типовою конфігурацією для сучасного ноутбука (4 ядра, 8-16гб оперативної пам'яті) коштуватиме від 0.102$ за годину [4], що дорівнює 0.102*24*30 = 73.44$ на місяць. Дивлячись на те, що зазвичай для обслуговування інфраструктури середнього проекту одного комп'ютера недостатньо, ціна за послуги хмари може вийти дуже значна. Та це ще все, в цю ціну не враховані витрати на інші ресурси AWS, пам'ять, передача даних, та інші сервіси які доведеться використовувати придбавши там комп'ютер.

Таблиця 1.1

Порівняльна таблиця цін комп'ютерів на AWS станом на 15.05.21

Назва	**Ціна/год**	**К-ть ядер**	**Пам'ять**	**Канал**
a1.xlarge	$0.102	4	8 GiB	Up to 10 Gigabit
t4g.xlarge	$0.1344	4	16 GiB	Up to 5 Gigabit
t3.xlarge	$0.1664	4	16 GiB	Up to 5 Gigabit
t3a.xlarge	$0.1504	4	16 GiB	Up to 5 Gigabit

Продовження таблиці 1.1

t2.xlarge	$0.1856	4	16 GiB	Moderate
m6g.xlarge	$0.154	4	16 GiB	Up to 10 Gigabit
m6gd.xlarge	$0.1808	4	16 GiB	Up to 10 Gigabit
m5.xlarge	$0.192	4	16 GiB	Up to 10 Gigabit
m5a.xlarge	$0.172	4	16 GiB	Up to 10 Gigabit
m5ad.xlarge	$0.206	4	16 GiB	Up to 10 Gigabit
m5d.xlarge	$0.226	4	16 GiB	Up to 10 Gigabit
m5dn.xlarge	$0.272	4	16 GiB	Up to 25 Gigabit
m5n.xlarge	$0.238	4	16 GiB	Up to 25 Gigabit
m5zn.xlarge	$0.3303	4	16 GiB	Up to 25 Gigabit
m4.xlarge	$0.20	4	16 GiB	High

Для порівняння в провідного українського постачальника хмарних послуг GigaCloud сервер на ОС Linux з конфігурацією в 8 Гб пам’яті, 3vCPU та 50Gb SSD коштуватиме 1560 грн на місяць [5].

Як було сказано вище, PaaS хмари найчастіше використовуються в компаніях, в яких, зазвичай, організовано корпоративну мережу. До цієї мережі під’єднано всі пристрої компанії, ноутбуки, стаціонарні комп’ютери, сервери, тощо, однак, за дуже рідким виключенням, ці пристрої використовують всі свої обчислювальні ресурси на протязі всього дня. Організувавши оркестр з таких пристроїв можна було б значно зменшити витрати на інфраструктуру та підвищити продуктивність сервера за рахунок збільшення кількості оброблених запитів за секунду.

Враховуючи викладене, задіяння обчислювальних ресурсів пристроїв корпоративної мережі, які не використовуються, «простоюють», є одним із перспективних напрямків збільшення продуктивності серверів. Проте

просто об’єднати ці пристрої в звичний кластер неможливо, пристрої постійно від’єднуються, або приєднуються до мережі.

Висновки

Зростання кількості сервісів в мережі інтернет потребує збільшення обчислювальних ресурсів для надання користувачам якісних послуг.

Хмарні технології дозволяють задовольнити фактично будь-які потреби користувачів, але є фінансово затратними. У цьому розділі були представлені основні моделі хмарних сервісів та наведена орієнтовна вартість витрат за віртуальний ПК на прикладі іноземного провайдера AWS та українського Gigacloud.

Зазвичай в умовах корпоративної мережі залишається багато невикористаних обчислювальних ресурсів, які можна задіяти на користь зменшення витрат на хмарні сервіси, або, навіть повністю їх замінити.

РОЗДІЛ 2
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ОРКЕСТРАЦІЇ СЕРВЕРІВ

У даному розділі розглядаються рішення за допомогою яких так чи інакше можна досягти результату, який задовільнить умови задачі. На даний час можна виділити сім основних.

2.1. Dynamic Configuration of Upstreams with the NGINX Plus API.

Це рішення використовується для балансування навантаження між робочими вузлами кластера с динамічною конфігурацією на основі програмного продукту Nginx Plus. Nginx Plus розширює функціонал звичайного Nginx з відкритим кодом. Саме на базі цього програмного продукту та технології віртуалізації буде запропоновано вирішити поставлену задачу.

До переваг цього програмного забезпечення слід віднести повноцінне рішення щодо доставки застосунків, а саме: балансування навантаження, кеш вміст, веб-сервер, засоби контролю, розширений моніторинг та управління програмами. у поєднанні з простотою використання. У разі придбання програмного продукту Nginx Plus користувач додатково отримує також доступ до Nginx API Gateway та Http API, що значно спрощує виконання поставленої задачі.

Приклад побудови інфраструктури API з використанням Nginx Plus [6] наведено на рисунку 2.1. Завдяки використанню Nginx Plus як шлюзу для API, контролер забезпечує автентифікацію користувачів, що мають доступ до вказаного сервісу.

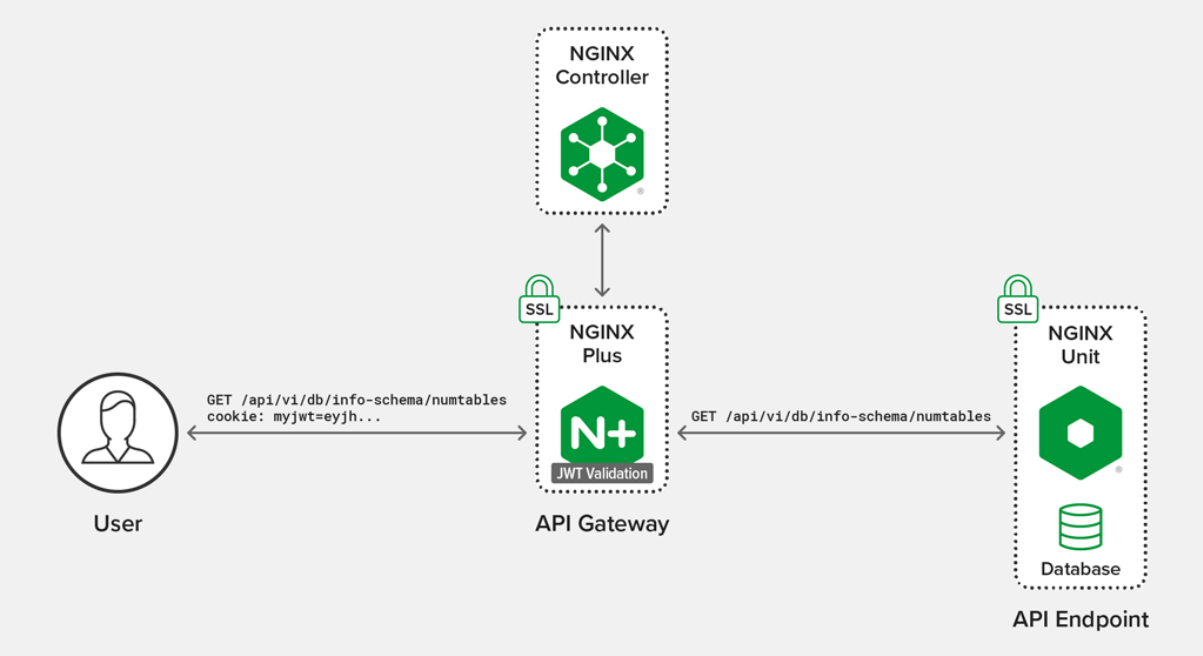

Рис. 2.1 Приклад інфраструктури API побудованого з використанням Nginx Plus

Однією з відмінних особливостей NGINX Plus є надання сервісу збору та зберігання додаткових показників з застосунків, таких як їх стан та конфігурація. Таким чином, за допомогою NGINX Plus з'являються можливості збирати докладні розширені дані про стан сервісі, здійснювати ефективні їх перевірки та забезпечувати API динамічної конфігурації, що в цілому забезпечує вирішення поставленої задачі. Слід зазначити, що використовуючи API NGINX Plus можливо оновлювати конфігурацію балансувальника «нальоту», з нульовим простоєм. Цей сервіс пропонує дуже широкий функціонал, що дозволить задовольнити вимоги у динамічному конфігуруванні кластера [7].

Серед недоліків цього сервісу є його ціна, яка стартує від декількох тисяч доларів на рік, що зводить нанівець його використання для вирішення поставленого завдання.

2.2. Kubernetes.

Kubernetes, також відомий як K8s, - це система з відкритим кодом для автоматизації масштабування, управління, та розгортання контейнерними

програмами. Kubernetes не є традиційною системою PaaS (платформа як послуга), що включає все. Він також надає деякі функції, загальні для PaaS, такі як розгортання, масштабування, балансування навантаження та дозволяє користувачам інтегрувати свої рішення для ведення журналу та моніторингу [8].

Однак Kubernetes не є монолітним, він надає будівельні блоки для побудови платформ для розробників, але зберігає вибір користувача та гнучкість там, де це важливо. Kubernetes працює з різноманітними інструментами контейнерів і запускає контейнери в кластері, часто із образами, створеними за допомогою Docker. При розгортанні Kubernetes, ви отримуєте кластер. Кластер Kubernetes складається з так званих вузлів - набору робочих машин, які запускають контейнерні програми. Кожен кластер має принаймні один робочий вузол. Робочі вузли розміщують поди, які є компонентами робочого навантаження програми. Поди - це найменші одиниці обчислювальної техніки, які можна створити та керувати ними в Kubernetes. Група з одного або декількох контейнерів із загальним сховищем та мережевими ресурсами та специфікацією запуску контейнерів називається подом. Под моделює специфічний для програми «логічний хост»: він містить один або кілька контейнерів програм, які відносно тісно пов'язані. У нехмарних контекстах програми, що виконуються на одній віртуальній або фізичній машині, є аналогами хмарних програм, що виконуються на одному логічному хості. На рисунку 2.2 наведена схема кластера Kubernetes з усіма компонентами, зв'язаними між собою [9]. Ядром кластеру є Control plane, який виконує функції керування системою та масштабування подів.

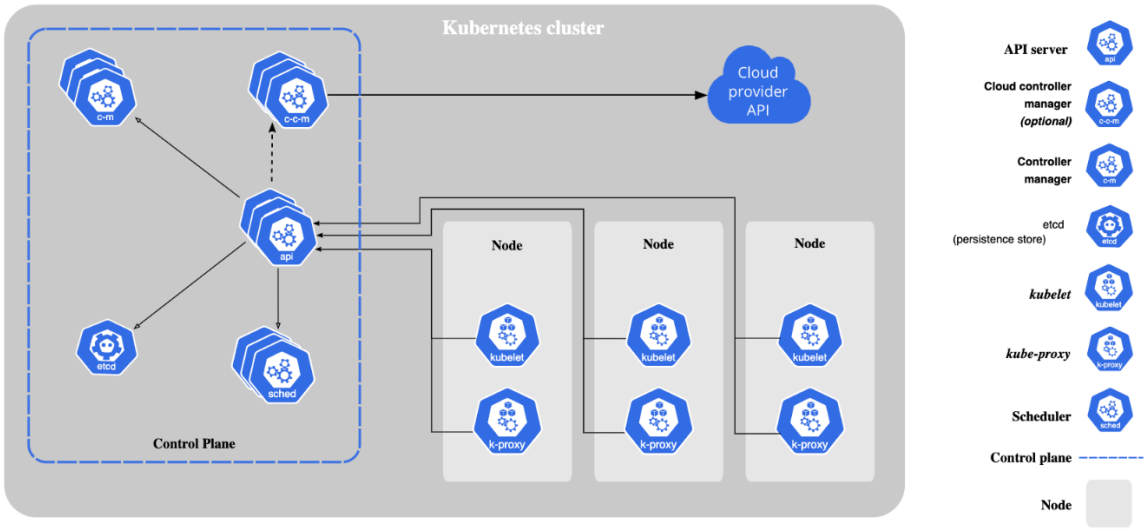

Рис. 2.2 Схема кластера Kubernetes

У виробничих середовищах Control plane, як правило, працює на декількох комп'ютерах, а кластер зазвичай працює на декількох вузлах, забезпечуючи відмовостійкість і високу доступність. Крім того, Control plane приймає глобальні рішення щодо кластера (наприклад, планування), а також виявляє і реагує на події кластера. Наприклад, запускає новий под, коли кількість бажаних реплік розгортання не задовольняється [10]. Вузол в залежності від кластера може бути віртуальною або фізичною машиною. Всі вузли містять сервіси, необхідні для запуску подів і управляються Control plane. Зазвичай має бути кілька вузлів у кластері, однак у навчальному або обмеженому ресурсами середовищі може бути лише один вузол.

Загалом кажучи засоби Kubernetes для управління застосунками в контейнерах можна ефективно використовувати для оркестрації. Але Kubernetes зміни у конфігурації кластера сприймає перш за все як виключну ситуацію, бо це є наслідком автоматичного масштабування кластеру. І, що більш важливе, програмно-апаратні характеристики обчислювальних вузлів такого кластеру мають бути однаковими для групи вузлів, що є досить складним для предметної області даної задачі. Тобто, кожен

пристрій що планується бути застосованим для оркестрації, в більшості випадків, повинен мати однакові апаратні характеристики (оперативна пам'ять, процесор, тощо), що в умовах поставленої задачі фактично неможливо. Крім того, на нодах крім програмного забезпечення для підтримки контейнеризації повинні бути встановлені додаткові програмні компоненти, такі як kubelet і kube-proxy.

2.3. Docker Swarm

Docker Swarm - це група фізичних або віртуальних машин, на яких запущена програма Docker і які налаштовані на об'єднання в кластер, який називається роєм [11]. Після того, як група машин згрупована, з'являється можливість запускати звичні команди Docker, які виконуються всіма машинами у кластері. Діяльність кластера контролюється менеджером роїв, а машини, що приєдналися до кластера, називаються вузлами. Принцип роботи Docker Swarm наведений на рисунку 2.3.

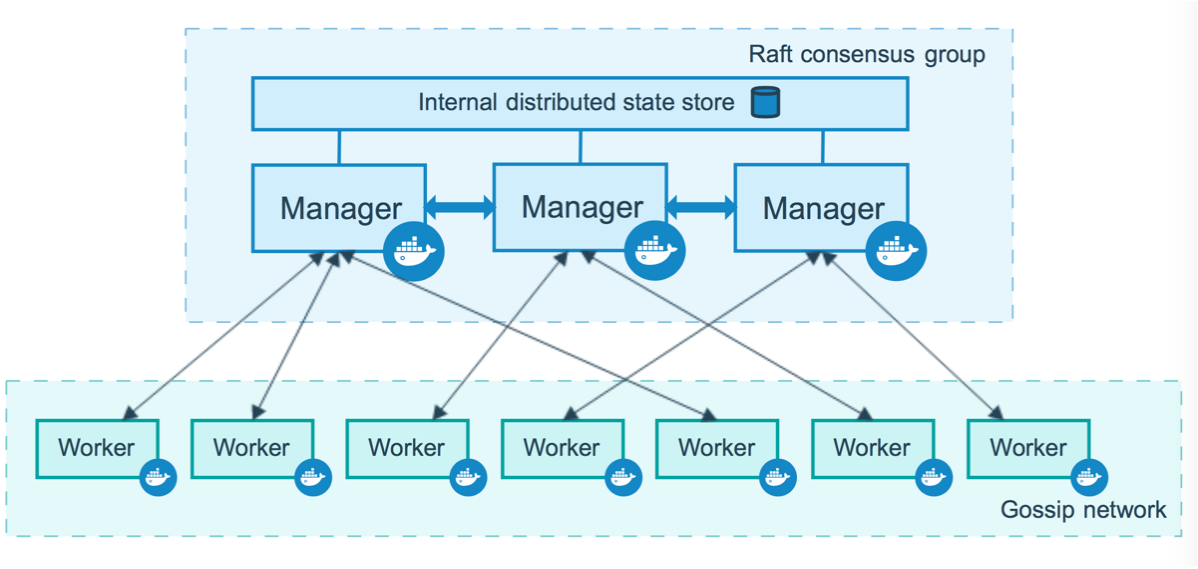

Рис. 2.3 Принцип роботи Docker Swarm

Однією з ключових переваг, пов'язаних з роботою Docker Swarm є високий рівень доступності програм. У Docker Swarm зазвичай є кілька робочих вузлів і принаймні один менеджер-вузол, який відповідає за

ефективну обробку ресурсів робочих вузлів та забезпечення ефективної роботи кластера. Додавання робочих вузлів збільшує пропускну здатність.

При розгортанні сервісу в Docker Swarm, головний елемент системи Docker планує завдання на доступних вузлах, будь то робочі вузли або вузли менеджера, а при додаванні вузлів до рою, збільшується масштаб рою для обробки завдань.

Вузли менеджера підвищують відмовостійкість, а також виконують функції оркестрації та управління кластером для рою. Серед менеджерських вузлів один керівний вузол виконує завдання оркестровки. Якщо вузол лідера вимикається, інші вузли менеджера обирають нового лідера та відновлюють оркестровку та підтримку стану рою. За замовчуванням вузли менеджера також виконують завдання. Docker Swarm – це досить зручний інструмент для оркестровки, проте великим недоліком являється поведінка в сценарії, коли один вузол видаляється з рою Docker знову переходить в звичайний режим. Після цього оркестратор більше не може приймати завдання для вузлів, що в умовах поставленої задачі буде виникати постійно.

2.4. Mesos

Mesos призначений для використання в розподіленому обчислювальному середовищі задля забезпечення ізоляції ресурсів та зручного управління підлеглим кластером, який являє собою підлеглий вузол. Тобто цей програмний продукт - це централізована відмовостійка система управління кластерами. У певному сенсі це відносно новий та ефективний спосіб ефективного управління серверною інфраструктурою. Характер його роботи протилежний традиційній віртуалізації - Mesos може об’єднати купу віртуальних машин в одну, замість поділу фізичної машини на кілька віртуальних [12]. Структурна схема такого кластера наведена на рисунку 2.4.

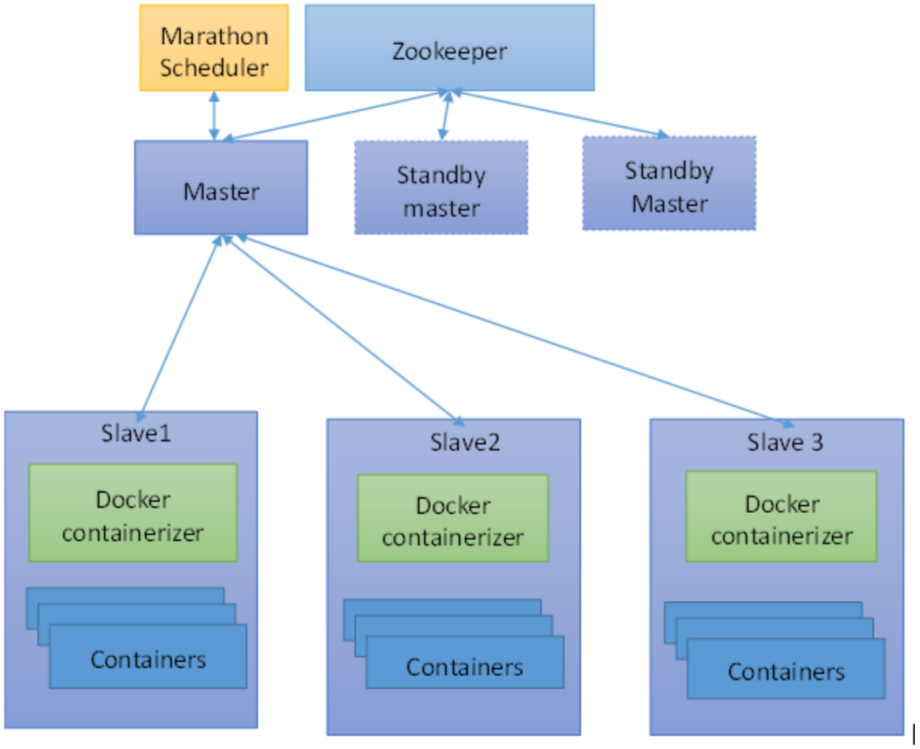

Рис. 2.4 Архітектура кластера Mesos

Mesos розподіляє ресурси ЦП та пам'ять для завдань у кластері способом, подібним до того, як ядро Linux розподіляє системні ресурси серед локальних процесів. Фреймворки беруть доступні ресурси від головного вузла і запускають завдання на підлеглих. Marathon - одна з таких платформ, яка запускає контейнерні програми на кластері Mesos. Разом Mesos і Marathon стають двигуном для організації контейнерів, таким як Docker Swarm або Kubernetes. Проте в Mesos також не можна динамічно додавати або видаляти підлеглі вузли (slave) з конфігурації, що є критичним недоліком для вирішення поставленої задачі.

2.5. Nomad

Nomad - простий у використанні, гнучкий та продуктивний оркестратор, який поєднує мікросервісні, пакетні, контейнерні та неконтейнерні додатки. Nomad легко масштабується, а також має вбудовану інтеграцію з іншими продуктами компанії Hashicorp [13]. Він

запускається як єдиний файл, в якому у єдину систему поєднується функції управління ресурсами та планування.

До ключових переваг слід віднести те, що Nomad не вимагає жодних зовнішніх послуг для зберігання або координації. Цей оркестратор автоматично обробляє помилки програм, вузлів та драйверів. В ньому використовується низка технологій щоб забезпечити високу готовність у разі невдач. Nomad розроблений як альтернатива Kubernetes [14]. Розробники програмного забезпечення обирають Nomad як альтернативу Kubernetes завдяки двом основним сильним сторонам:

– Простота використання та ремонтопридатність

– Гнучкість розгортання та управління контейнерними та неконтейнерними програмами [15].

Структурну схему кластера Nomad наведено на рисунку 2.5

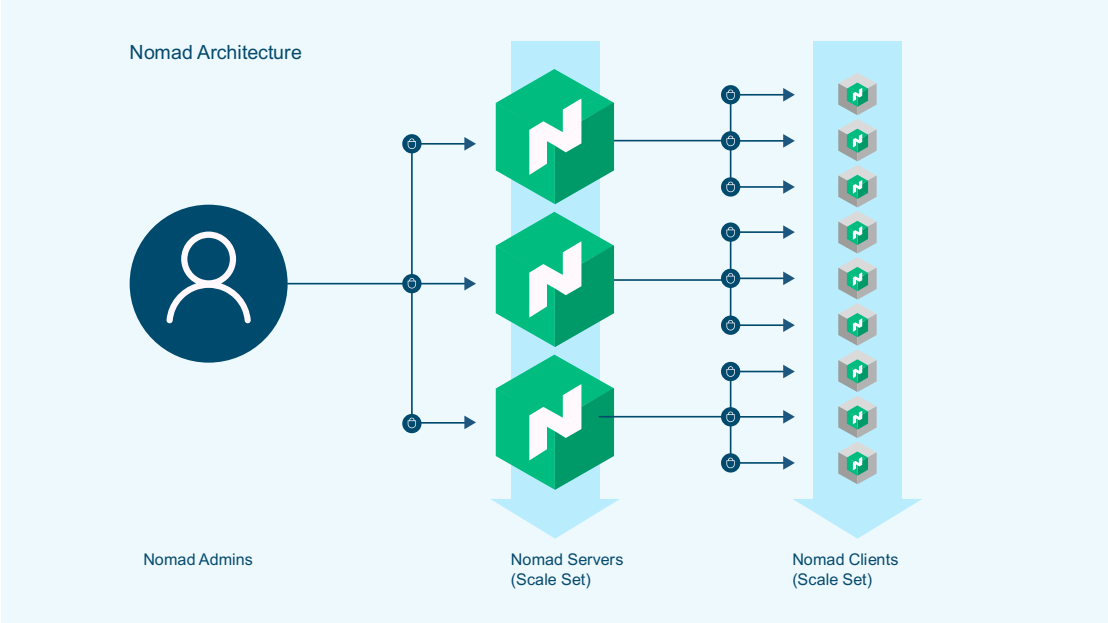

Рис. 2.5 Архітектура кластера Nomad

Крім того, разом із більшою простотою у порівнянні з Kubernetes, Nomad пропонує менше можливостей, та й загалом, технічно дане рішення знаходиться позаду інших розробок.

2.6. Flocker

Flocker - це сервіс оркестрації даних контейнерів з відкритим кодом для докеризованих програм [16]. Flocker надає всі інструменти, необхідні для запуску контейнерних сервісних служб, наприклад таких, як бази даних з високим навантаженням. На відміну від Docker, який прив'язаний до одного сервера, Flocker є портативним і може використовуватися з будь-яким контейнером, незалежно від того, де цей контейнер запущений [17]. Структурну схему кластеру з використанням Flocker наведено на рисунку 2.6.

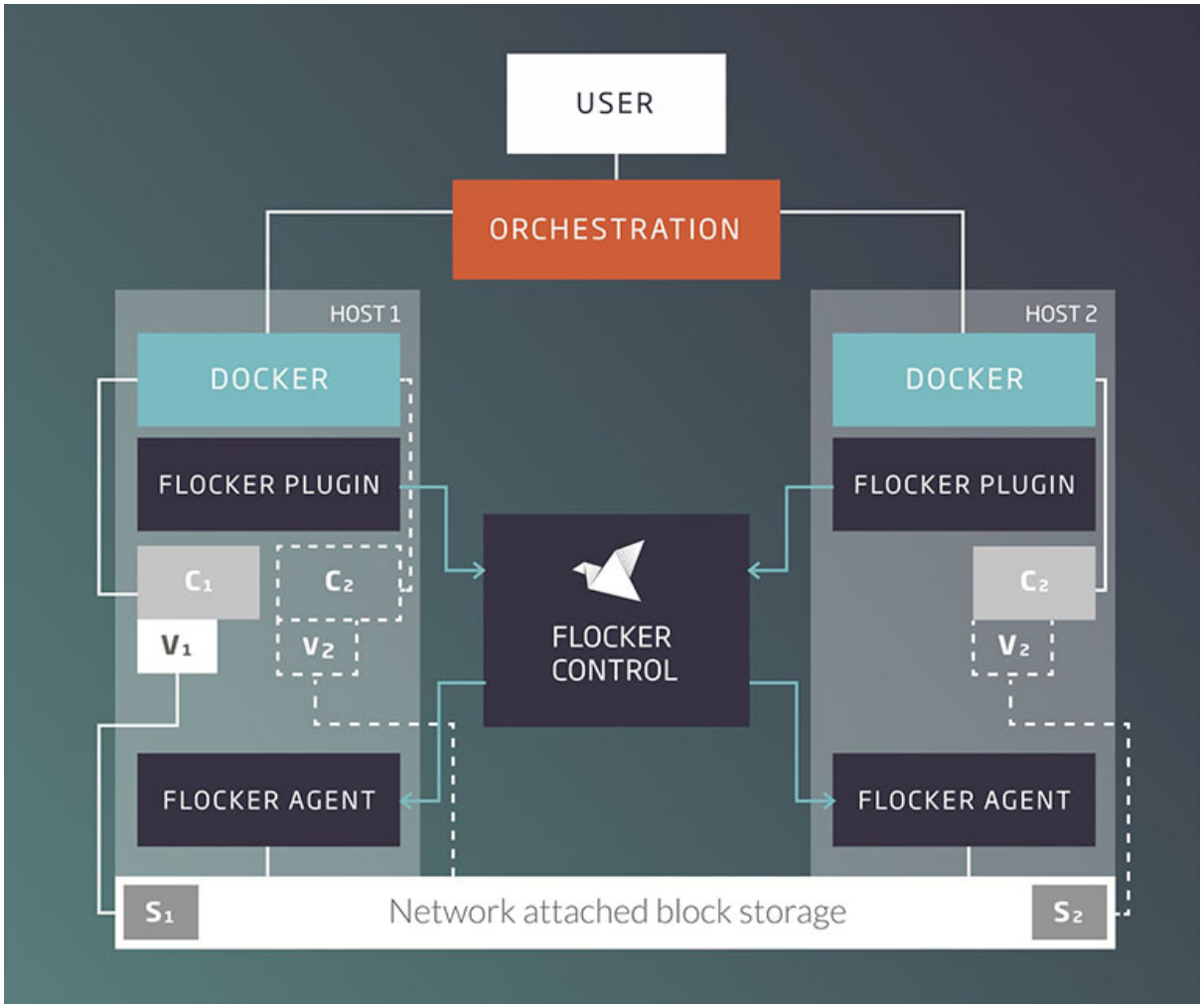

Рис. 2.6 Структура кластера з використанням Flocker

При використанні Flocker для управління мікросервісом, Flocker буде стежити за вашими контейнерами, коли вони переміщаються між різними

хостами у кластері, що було би досить зручно в умовах поставленої задачі, однак підтримку даного програмного пакету завершено.

2.7. Helios

Helios - це платформа оркестрації для розгортання та управління контейнерами одразу на великій кількості серверів. Helios пропонує HTTP API, а також клієнт командного рядка для взаємодії із серверами, на яких запущені контейнери. Helios також зберігає історію подій у кластері, включаючи таку інформацію, як розгортання, перезавантаження та зміни версії. Платформа націлена вирішити всі проблеми, які можуть виникнути при розгортанні сервісів. З популярних фреймворків оркестрування Docker Helios - єдиний, в якого не має залежностей, що є значною перевагою, оскільки гарантує більшу надійність. Тобто, він не потребує наявності хмарних сервісів, певної топології мережі або запуску певної операційної системи. Єдина вимога - мати розгорнутий кластер та JVM на машинах, на яких працює Helios [18]. Цей проект створений, коли не було інших фреймворків оркестрації контейнерів з відкритим кодом. З моменту появи Kubernetes та інших інструментів його підтримку завершено. Розробники перестали додавати нові функції до Helios і перейшли до інших інструментів, таких як Kubernetes. Цей проект більше не отримуватиме нових функцій і не прийматиме запити на зміну коду. Однак розробники будуть продовжувати виправляти знайдені помилки.

Висновки

Результати аналізу існуючих рішень показують, що для вирішення поставленої задачі необхідно або адаптувати існуючі технології, або ж спробувати створити власний оркестратор.

Найбільш перспективним з огляду на можливості модифікації та виглядає Kubernetes. Однак, він має низку недоліків, тому у даній роботі не застосовується.

Враховуючи відкритий програмний код Nginx пропонується вирішити поставлену задачу за допомогою цього програмного пакету.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ ДЛЯ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ МІЖ СЕРВЕРАМИ В КЛАСТЕРІ

Типовою архітектурою сучасного серверу може розглядатись наступна [19]. Існує кластер (сервер), на якому працює бізнес логіка, що обслуговує додаток. Для виконання запитів сервером необхідний балансувальник навантаження, наприклад Nginx або Apache HTTP Server [20]. Основне завдання балансувальника у більшості випадків, це перенаправлення запитів у вказане місце. У нашому випадку на сервер з бізнес логікою. Сервер може бути не один і балансувальник допомогає розподілити запити між серверами із заданим алгоритмом.

У роботі пропонується використовувати у якості балансувальника Nginx. Вибір на користь Nginx обумовлено тим, що Apache HTTP Server використовує застарілі технології, надає менше гнучкості для розробки та гірше справляється з навантаженням. Apache використовує власне рішення для обробки запитів, під назвою “Multi-Processing Modules”, розробка якого датується виходом першої версії веб серверу. І, хоча, це рішення було модернізовано, воно все ще технічно відстає від технологій, використаних в Nginx. Так, станом на січень 2021 року Netcraft підрахував, що Apache обслуговував 24,63% від мільйона найбільш зайнятих веб-сайтів, тоді як Nginx обслуговував 23,21%, а Microsoft на третьому місці - 6,85% (за деякими іншими показниками Netcraft Nginx випереджає Apache), за даними W3Techs, Apache посідає перше місце - 35,0%, Nginx - друге - 33,0%, а Cloudflare Server - третє - 17,3%. [21]

Nginx має декілька доступних правил балансування, одне з них використано в роботі та має назву алгоритм “Round Robin”. При цьому алгоритмі кожен сервер має свою “вагу” і запити розподіляються відповідно до неї [22]. Розглянемо запропоноване рішення, структурна схема якого наведена на рисунку 3.1.

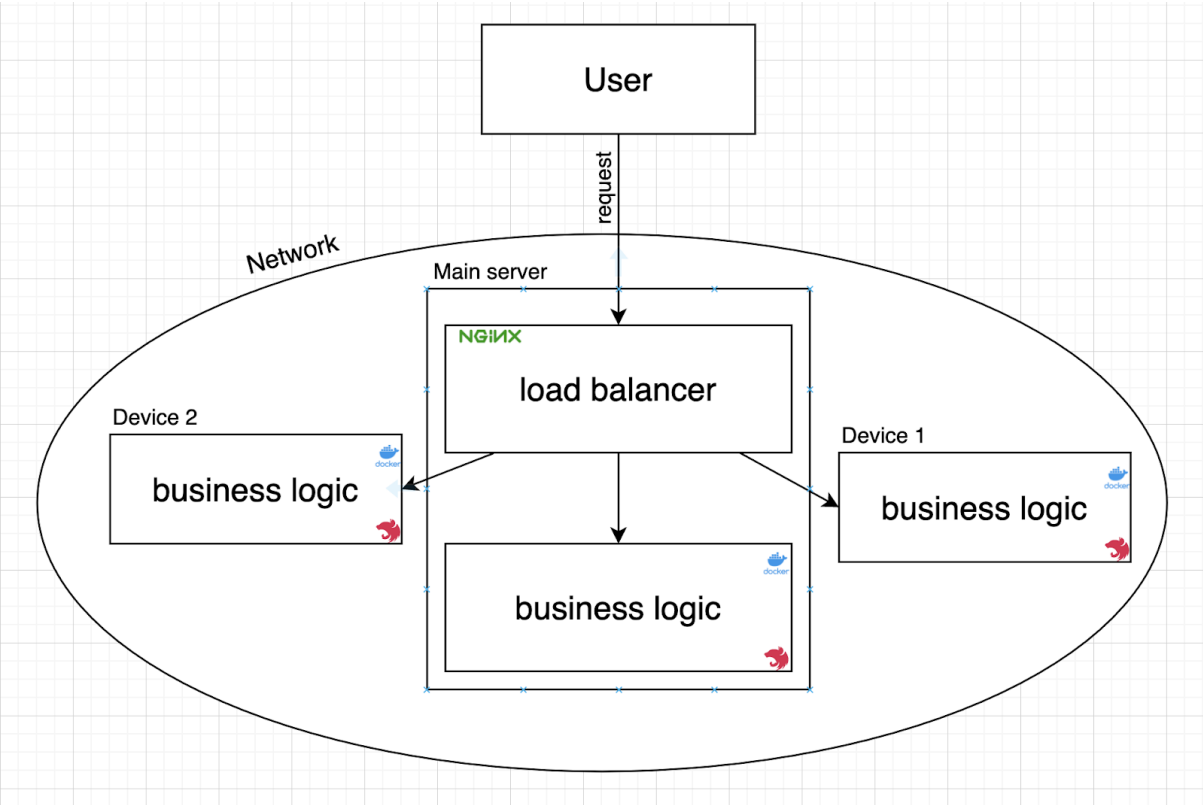

Рис. 3.1. Структурна схема рішення

Стислий опис рішення полягає в наступному. В наявності є мережа, в якій знаходиться головний сервер з балансувальником Nginx. Як було описано раніше у роботі Nginx - це безкоштовний веб-сервер з відкритим кодом, який також може використовуватися як зворотний проксі-сервер, балансувальник навантаження, поштовий проксі-сервер та кеш HTTP. Компанія Nginx також є розробником платного програмного забезпечення Nginx Plus який було розлянуто як конкурента запропонованої технології і ідеї якого використано в розробці. Бізнес логіка застосунку упакована в Docker контейнер. Docker - це програмний продукт, який формує платформу PaaS, проте він працює не в хмарі. Він може бути розгорнутий на майже будь-якому комп'ютері, який на рівні операційної системи підтримує віртуалізацію для створення контейнерів, тобто пакетів програмного забезпечення. Контейнери об'єднують власне програмне забезпечення, бібліотеки та конфігураційні файли і ізольовані один від одного. Контейнери спілкуються між собою за чітко визначеними

каналами, які конфігурує користувач. Оскільки всі контейнери використовують служби одного ядра операційної системи, вони використовують менше ресурсів, ніж віртуальні машини. Docker може пакувати програми та їх залежності у віртуальний контейнер. Контейнери можуть працювати на будь-якому комп'ютері під управлінням ОС Linux, Windows або macOS [23]. Це дозволяє запускати програми в різних місцях, таких як локальні, загальнодоступні та/або приватні хмари.

При запуску в Linux Docker ізолює ресурси ядра Linux та використовує власну функціональну файлову систему, що дозволяє контейнерам запускатися в межах одного екземпляра Linux. Це дозволяє уникнути додаткових витрат на запуск та обслуговування віртуальних машин. На macOS для запуску контейнерів Docker використовується віртуальна машину Linux [24]. Інтерфейси, які використовує Docker для доступу до засобів віртуалізації наведено на рисунку 3.2.

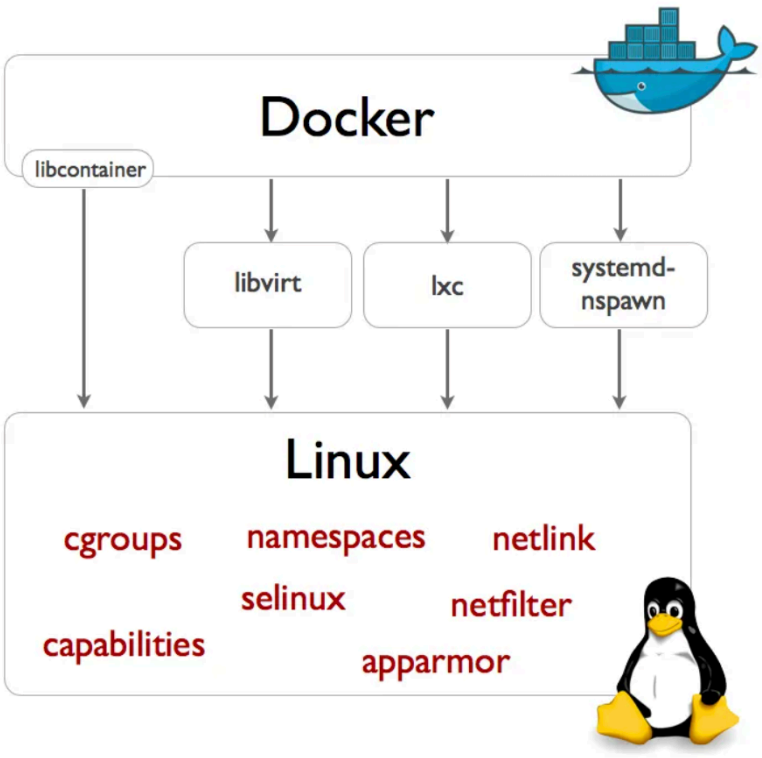

Рис. 3.2. Інтерфейси, які використовує Docker для доступу до засобів віртуалізації

Docker зручно використовувати для контейнеризації застосунку, тому, що це дозволяє абстрагуватися від операційної системи, на якій буде запущено додаток. Також є можливисть запакувати все в образі, який потім може бути скачаний з репозиторія для їх зберігання, такого як Docker Hub або AWS ECR. Всі комп'ютери, що використовуються для задачі також мають встановлений Docker і знаходяться в одній мережі з головним сервером, хоча останнє і не є обов'язковим.

Для того щоб виконати початкове налаштування балансувальника та для зміни конфігурації в ході роботи використаємо CGI. CGI має певні переваги над іншими методами при зміні конфігурації під час роботи. CGI не використовує багато системних ресурсів, не потребує сторонніх технологій, функціоналу яких буде забагато для цих цілей. Також CGI має багато варіантів реалізації на різних мовах програмування та не потребує багато зусиль в розробці [25]. У запропонованому рішенні розглянутий варіант CGI з використанням NodeJS.

Отримуючи запит на редагування конфігурації від нового пристрою, сервер отримує ip адресу комп'ютера з якого цей запит було надіслано і додає його до списку серверів в конфігурації Nginx. Надалі запити будуть транслюватися на ці сервери. Для оновлення конфігурації необхідно перезавантажити Nginx, при цьому запити будуть розподілятись між серверами. Таким чином діаграма станів проекту зображена на рисунку 3.3.

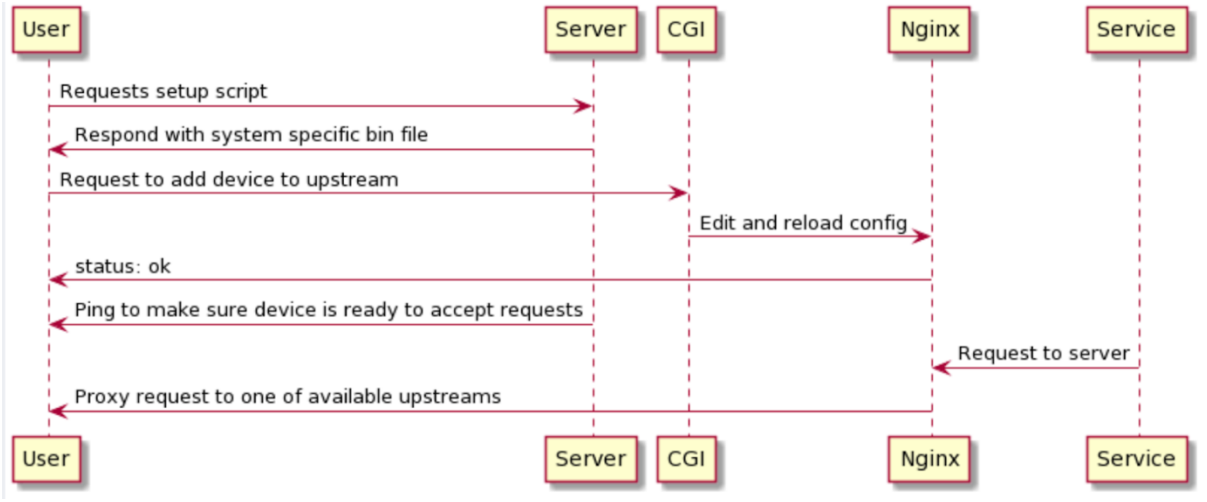

Рис. 3.3 Діаграма станів проекту

При запуску головний сервер публікує посилання на певній адресі. Користувач здійснює завантаження скрипта з цієї адреси. Скрипт після виконання скачує образ зі схожою бізнес логікою, встановить його та запустить. Код скрипта наведено нижче.

```
#!/bin/sh

echo starting
docker run -d -p 3000:3000 --network host --name
scaler sealch/scaler-client
```

Після виконання цих команд додаток буде автоматично запущено. У якості бізнес логіки додатку виступає контейнер з node.js всередині, та додаток під управлінням фреймворку NestJS. Вибір NestJS обумовлено тим, що це найдійний та перевірений часом фреймворк для NodeJS, який підходить для створення ефективних, надійних та масштабованих серверних застосунків. Крім того NestJS надає багато гнучкості, дозволяючи використовувати будь-які інші бібліотеки сторонніх розробників завдяки модульній архітектурі. Цей фреймворк також має адаптивну екосистему, яка є повноцінною основою для всіх видів серверних програм, а також використовує переваги останніх функцій JavaScript та TypeScript. NestJS поєднує елементи ООП, функціонального програмування та функціонального реактивного програмування. NestJS

використовує інші надійні фреймворки, такі як Express (за замовчуванням), або Fastify, який може збільшити швидкодію. Структурна схема файлів проекту наведена на рисунку 3.4.

Рис. 3.4. Структура файлів в проекті

Dockerfile – файл конфігурації Docker, де описані правила та послідовність дій щодо збирання та пакування застосунку у контейнер. Конфігурація Dockerfile наведена на рисунку 3.5.

```
FROM node:lts-alpine as builder

WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci
COPY . .

RUN npm run build

FROM node:lts-alpine
WORKDIR /usr/src/app

COPY --from=builder /usr/src/app/package*.json ./
COPY --from=builder /usr/src/app/dist ./dist

RUN npm ci --only=production
EXPOSE 3000
CMD [ "npm", "run", "start:prod" ]
```

Рис. 3.5. Файл конфігурації зображення Docker

Будування зображення додатку проводиться у два етапи задля полегшення першого запуску. В конфігурації використано полегшену alpine версію зображення nodejs та інстальовано всі необхідні залежності за допомогою команди npm ci. Файли застосунку побудовано та перенесено як готовий результат. Після цього залежності для побудови не потрібні, здійснюються запуск команди інсталяції з параметром --only=production. Останнім кроком буде запуск додатку.

Зображення Docker будується за допомогою наступної команди:

```
docker build -t scaler-client .
```

Додатково необхідно завантажити зображення на один з хостингів, наприклад Docker Hub. Попередньо необхідно створити на ньому власний аккаунт та репозиторій.

```
docker tag scaler-client sealch/scaler-client
docker push sealch/scaler-client:latest
```

Після цього пристрої, які будуть під'єднуватись до мережі, зможуть завантажувати та запускати зображення.

Файли з розширенням .controller.ts - це файли, які обробляють запити, так звані контролери. Бізнес логіка застосунку зазвичай розташовується в файлах .service.ts. Розглянемо файл app.module.ts

```
export class AppModule implements OnModuleInit {
  constructor(
    private readonly httpService: HttpService,
  ) {}

  async onModuleInit(): Promise<void> {
    console.log("notifying main server...")
    await
this.httpService.put("http://192.168.1.117/upstream"
      .toPromise()
      .then(r => r.data)
      .then(console.log)
  }
}
```

Цей файл описує залежності застосунку, імпортує необхідні сервіси та контролери. В класі модуля описана поведінка, яку необхідно провести на самому старті модуля, тобто при підключенні його в застосунок. Запити відправляються на головний сервер, де розташований CGI, який здатен їх обробити та оновити конфігурацію балансувальника Nginx.

```
@Controller()
export class AppController {
  constructor(private readonly appService:
AppService) {}

  @Put('upstream')
  async addUpstream(
    @Ip() ip: string,
  ): Promise<string> {
    const realIp = ip?.split(":").pop()
    this.appService.updateConf(realIp)
```

```
    return realIp
  }
}
```

Цей контролер має кінцеву точку, яка називається upstream, і описаний вбудованим в NestJS декоратором Put. Після надходження запиту до контролера за допомогою декоратора Ip буде виявлено ip адресу хоста, з якого надійшов запит.

Задля коректної обробки формату ip у локальній мережі виконується низка перетворень, кінцевий результат яких передається в сервіс для оновлення конфігурації.

```
export class AppService {
  constructor(
    private nginx: NginxService
  ) { }

  upstreams = new Set([
    'server host.docker.internal:3000 weight=1;'
  ])

  async updateConf(ip: string): Promise<any> {
    this.upstreams.add(`server ${ip}:3000 weight=1;`)
    this.nginx.update(Array.from(this.upstreams))
  }
}
```

Клас сервісу - це місце, де розробляється більшість бізнес логіки проекту. Тут також підключається інший сервіс NginxService, який безпосередньо контролює поведінку Nginx. Такі включення можливі завдяки системі ін'єкцій залежності аналогічній шаблонам проектування ООП, яка є в NestJS. Далі є сет серверів збережених в зручному форматі в пам’яті застосунку. Додатково реалізована асинхронна функція updateConf, яка в свою чергу приймає ip адресу в якості параметра, додає в сет серверів ще один запис, з отриманим ip та викликає функцію update сервісу nginx з переданим набором серверів.

```
@Injectable()
export class NginxService {
  public async update(upstreams: string[]):
Promise<void> {
    const config = `upstream scale {
  ${upstreams.join('\n  ')}
}

server {
  listen 80;
  server_name localhost;
  location / {
    proxy_pass http://scale;
  }
}`

    await
fs.writeFile('/etc/nginx/conf.d/default.conf',
config)

    exec('sudo systemctl reload nginx', (e, data) =>
{
      console.log(e, data)
    })
  }
}
```

Надалі переходимо в NginxService. Цей сервіс бере на себе відповідальність в контролюванні самого балансувальника Nginx. Маємо асинхронну функцію update, в якій змінна config репрезентує частину налаштувань Nginx. У функцію підставлено змінну, яка була передана параметром, тобто сет серверів для розподілення навантаження. Після цього за допомогою вбудованого в node.js пакету fs файл конфігурації зберігається та виконується команда перезавантаження.

Для перевірки працездатності і оновленої конфігурації Nginx розглянемо вміст контейнера.

```
docker exec -it my_nginx /bin/sh
```

Після цього треба зайти в папку nginx шляхом введення команди cd та вивести файл конфігурації за допомогою команди cat

```
cd /etc/nginx/
cat nginx.conf
```

Вивід команди cat наведено нижче.

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
    worker_connections 768;
    # multi_accept on;
}

http {

    upstream scale {
        server 192.168.1.117:3000;
        server 192.168.1.234:3000;
   }

  server {
    listen 80;
    server_name scaler;
    location / {
      proxy_pass http://scale;
    }
  }
}
```

Конфігурація оновлена, а отже кластер зконфігуровано і обидва підключених пристроїв готові обробляти запити.

Для оцінки результатів необхідно створити кінцеву точку, при запиті на яку буде створено навантаження на сервер, яке дозволить об’єктивно оцінити продуктивність системи. Для цього обрано знаходження чисел Фібоначчі за обмежену кількість ітерацій.

```
@Controller()
export class AppController {
  constructor(private readonly appService:
AppService) {}
```

```
  @Get('benchmark')
  getBenchmark(): string {
    return this.appService.benchmark();
  }
}
```

Контролер обробляє запит та робить виклик сервісної функції benchmark.

```
import { Injectable } from '@nestjs/common';
import * as fibonacci from 'fibonacci';

@Injectable()
export class AppService {
  benchmark(): string {
    return fibonacci.iterate(3000);
  }
}
```

Сервісна функція benchmark повертає результат пошуку за 3000 ітерацій. Для перевірки ретрансляції запитів на зконфігуровані сервери виконується команда curl, яка зробить запит на сервер та поверне результати.

```
curl http://192.168.1.150/benchmark
{"number":"4106158863079712603335683787192671052201251086373692524088854309269055842741134037313304916608500445608300368357069422745885693621454765026743730454468521604866062924973605034697734537331968874058472552900820490869075126220590545421958897580311092226708492747938595391333183712447955431476110732762400667379340851917318109932017067768389347667647787395021744702686278209185538422258583064083016618629003582668572382102358025043519514729979196765240047842363764533472683641526483462458405732142414199379172429186026398100978669423920154046201538186714257398350748513964211
```

```
9982713640679581178458198658692285968043243656709796000","length":627,"iterations":"3000","ms":803}
```

Отримані результати показує, що запропонований алгоритм працездатний та можна приступати до порівняння продуктивності сервера. Задля порівняння результатів використано командний застосунок Apache Bench, який дозволяє симулювати навантаження на сервер шляхом створення трафіку за вказаними параметрами. В першому випадку будемо використовувати налаштування в 100 паралельно відправлених запитів з 10 клієнтів. Тестування проводиться лише на головному сервері, конфігурація intel core i7-4720hq та 8гб оперативної пам’яті. Під Docker виділено 1 гб пам’яті та 1 vCPU.

```
ab -n 100 -c 10 http://192.168.1.150/benchmark
```

Виконання цієї команди запускає тестування, яке може продовжуватись достатньо довго. Результати виконання команди наведені на рисунку 3.6.

```
Server Software:        nginx/1.19.10
Server Hostname:        192.168.1.150
Server Port:            80

Document Path:          /benchmark
Document Length:        683 bytes

Concurrency Level:      10
Time taken for tests:   102.717 seconds
Complete requests:      100
Failed requests:        0
   (Connect: 0, Receive: 0, Length: 19, Exceptions: 0)
Total transferred:      89181 bytes
HTML transferred:       68281 bytes
Requests per second:    0.97 [#/sec] (mean)
Time per request:       10271.706 [ms] (mean)
Time per request:       1027.171 [ms] (mean, across all concurrent requests)
Transfer rate:          0.85 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        5    8   1.9      7      12
Processing:  1154 9881 2095.6  10131   13544
Waiting:     1064 7429 2409.2   8054   10848
Total:       1163 9889 2095.7  10138   13550

Percentage of the requests served within a certain time (ms)
  50%  10138
  66%  10240
  75%  10428
  80%  10629
  90%  11235
  95%  13211
  98%  13548
  99%  13550
 100%  13550 (longest request)
```

Рис. 3.6 Результати виконання тесту на 100 запитів з 10 клієнтів при відсутності додатково підключених серверів

Всі запити успішно оброблені, час виконання тестів склав 102.717 секунд. При цьому середня кількість запитів які сервер може обробити за секунду склала 0.97 запитів/сек. Таким чином, середній час обробки запиту дорівнює приблизно 1027 мілісекунд. Тепер активуємо новий пристрій, ноутбук з конфігурацією i7-9750H та 16 Гб оперативної пам’яті, під Docker

виділено 6 vCPU та 2 Гб оперативної пам’яті. Запускаємо тест на тих самих налаштуваннях. Результати виконання команди наведені на рисунку 3.7.

```
Server Software:        nginx/1.19.10
Server Hostname:        192.168.1.150
Server Port:            80

Document Path:          /benchmark
Document Length:        682 bytes

Concurrency Level:      10
Time taken for tests:   19.753 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      91400 bytes
HTML transferred:       68200 bytes
Requests per second:    5.06 [#/sec] (mean)
Time per request:       1975.267 [ms] (mean)
Time per request:       197.527 [ms] (mean, across all concurrent requests)
Transfer rate:          4.52 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        5    7  12.2      5     128
Processing:   230 1848 349.2   1943    2009
Waiting:      229 1848 349.2   1943    2009
Total:        238 1855 343.1   1949    2017

Percentage of the requests served within a certain time (ms)
  50%   1949
  66%   1957
  75%   1960
  80%   1961
  90%   1972
  95%   1974
  98%   1978
  99%   2017
 100%   2017 (longest request)
```

Рис. 3.7 Результати виконання тесту на 100 запитів з 10 клієнтів при одному підключеному серверу

Кількість часу, протягом якого виконувався тест, радикально зменшилась та відповідно кількість оброблених кластером запитів за секунду збільшилась до 5.06 запитів/сек. Середній час відповіді також

зменшився до 197.527 мілісекунд. Відповідно до цих результатів можна побудувати наступні графіки.

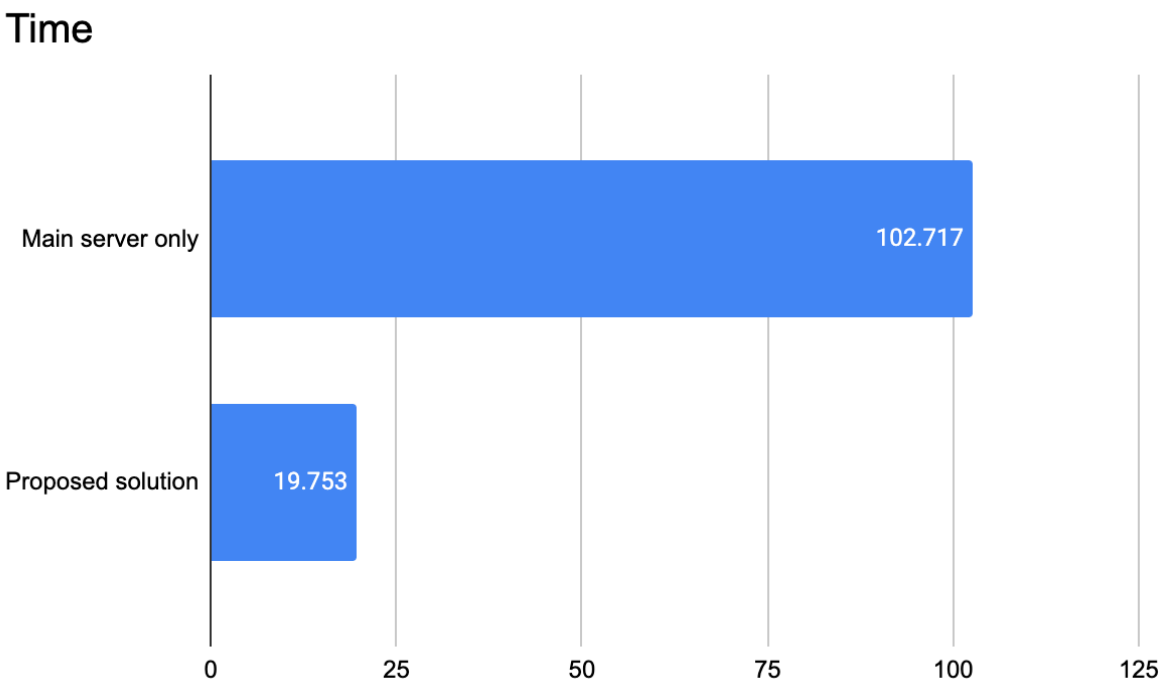

Рис. 3.8 Час витрачений на тест, менше - краще

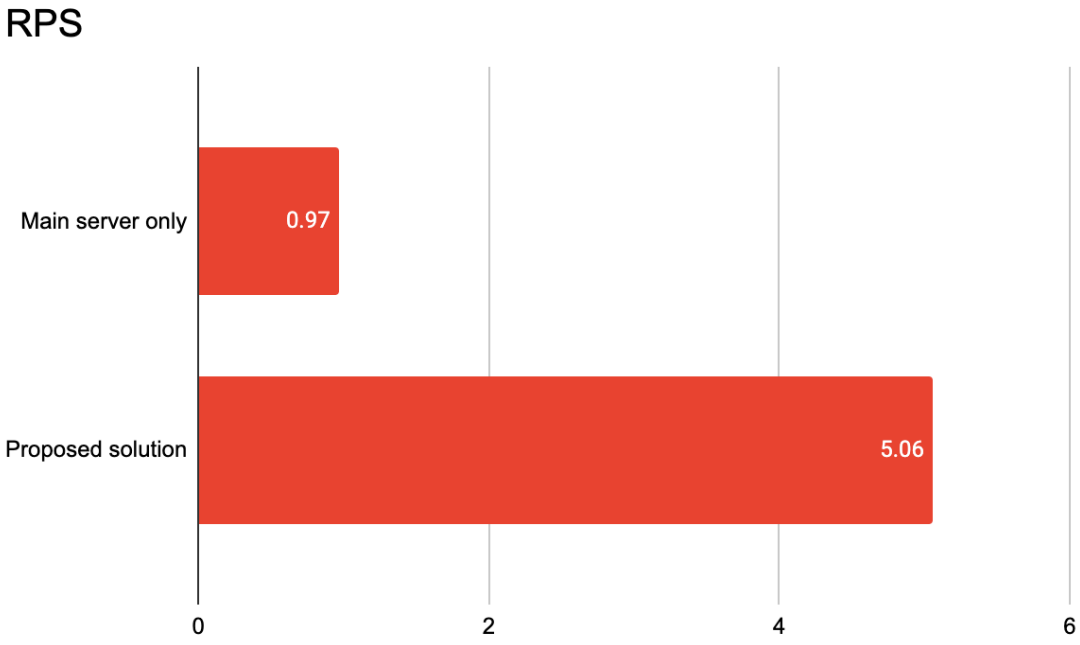

Рис. 3.9 Кількість оброблених запитів за секунду, більше - краще

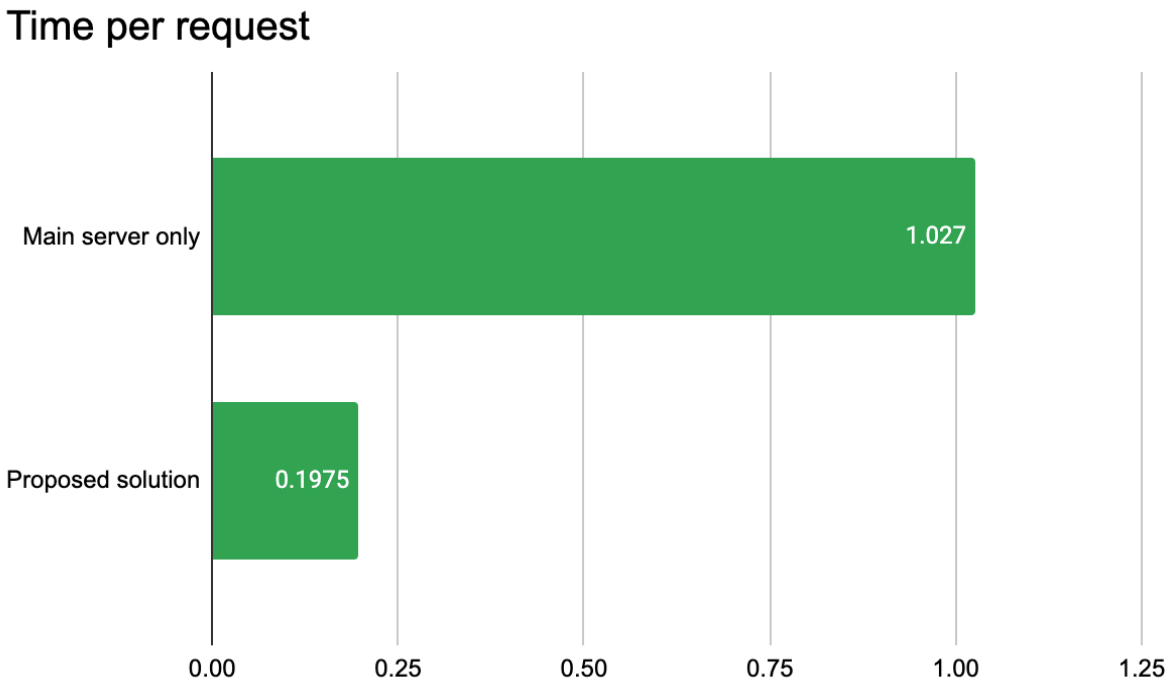

Рис. 3.10 Середній час відповіді сервера в секундах, менше - краще

Рис. 3.11 Кількість успішно оброблених запитів при 10 клієнтах, більше - краще

Тепер відключимо другий комп’ютер від мережі, а кількість одночасно відправляючих запити клієнтів збільшимо до 100, що більш реалістично в умовах сучасного невеликого веб застосунку та запустимо тест.

```
ab -n 100 -c 100 http://192.168.1.150/benchmark
```

Результати виконання тесту наведено на рисунку 3.12.

```
Server Software:        nginx/1.19.10
Server Hostname:        192.168.1.150
Server Port:            80

Document Path:          /benchmark
Document Length:        683 bytes

Concurrency Level:      10
Time taken for tests:   102.717 seconds
Complete requests:      100
Failed requests:        0
   (Connect: 0, Receive: 0, Length: 19, Exceptions: 0)
Total transferred:      89181 bytes
HTML transferred:       68281 bytes
Requests per second:    0.97 [#/sec] (mean)
Time per request:       10271.706 [ms] (mean)
Time per request:       1027.171 [ms] (mean, across all concurrent requests)
Transfer rate:          0.85 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        5    8   1.9      7      12
Processing:  1154 9881 2095.6  10131   13544
Waiting:     1064 7429 2409.2   8054   10848
Total:       1163 9889 2095.7  10138   13550

Percentage of the requests served within a certain time (ms)
  50%  10138
  66%  10240
  75%  10428
  80%  10629
  90%  11235
  95%  13211
  98%  13548
  99%  13550
 100%  13550 (longest request)
```

Рис. 3.12 Вивід тесту при відключених додаткових серверах

Підключаємо другий комп’ютер до оркестратора та знову запускаємо тест. Результати тестування наведено на рисунку 3.13.

```
Server Software:        nginx/1.19.10
Server Hostname:        192.168.1.150
Server Port:            80

Document Path:          /benchmark
Document Length:        682 bytes

Concurrency Level:      100
Time taken for tests:   20.229 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      91400 bytes
HTML transferred:       68200 bytes
Requests per second:    4.94 [#/sec] (mean)
Time per request:       20228.681 [ms] (mean)
Time per request:       202.287 [ms] (mean, across all concurrent requests)
Transfer rate:          4.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       29   97  36.0    113     191
Processing:   237 9959 5772.5  10044   19775
Waiting:      233 9959 5772.6  10044   19775
Total:        368 10056 5747.4  10159   19804

Percentage of the requests served within a certain time (ms)
  50%  10159
  66%  13382
  75%  15132
  80%  16103
  90%  18050
  95%  19036
  98%  19611
  99%  19804
 100%  19804 (longest request)
```

Рис. 3.13 Вивід тесту при додатково підключеному комп’ютері

Відповідно для цього тесту графіки матимуть наступний вигляд:

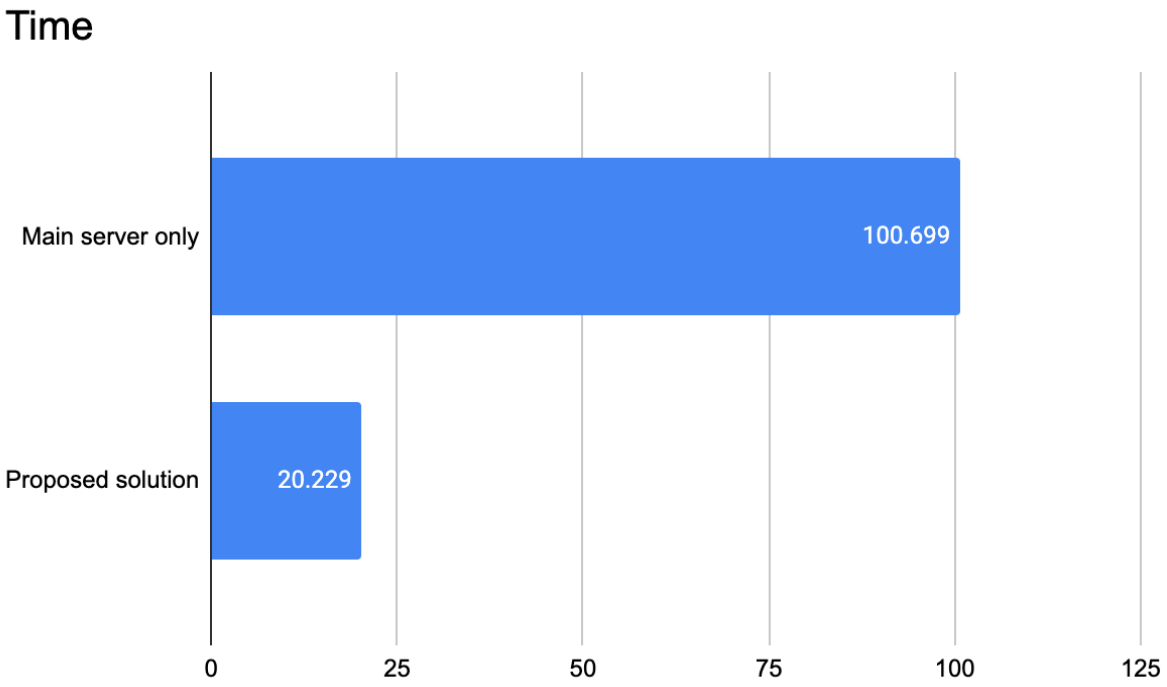

Рис. 3.14 Час витрачений на тест, менше - краще

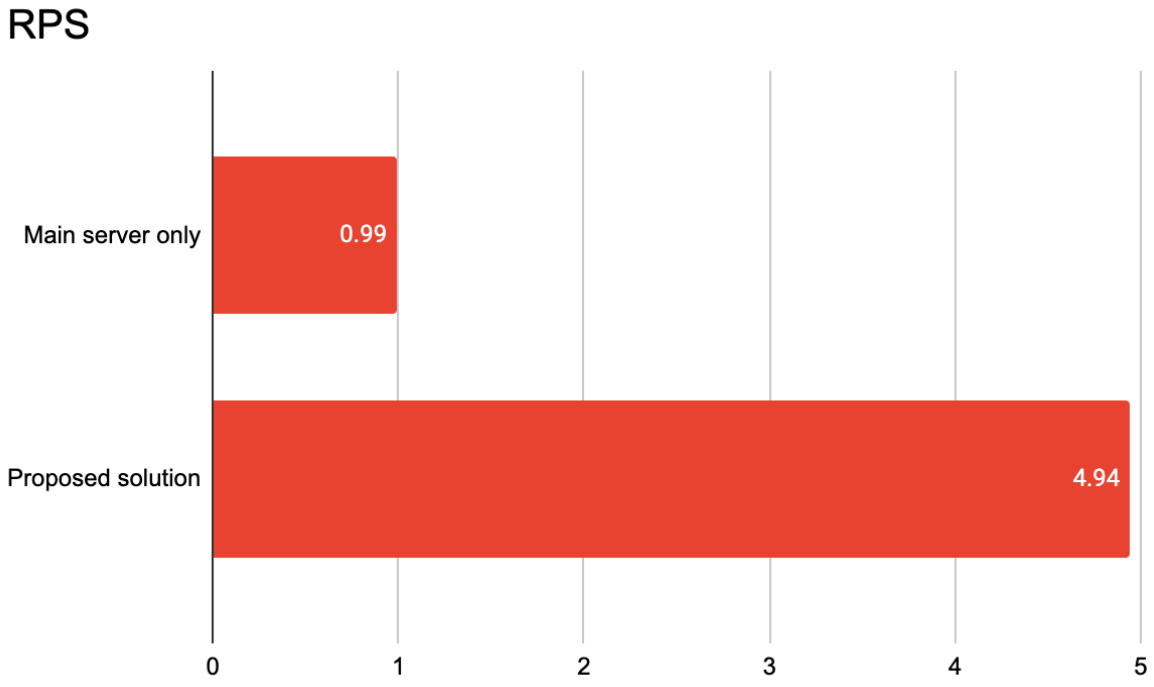

Рис. 3.15 Кількість оброблених запитів за секунду, більше - краще

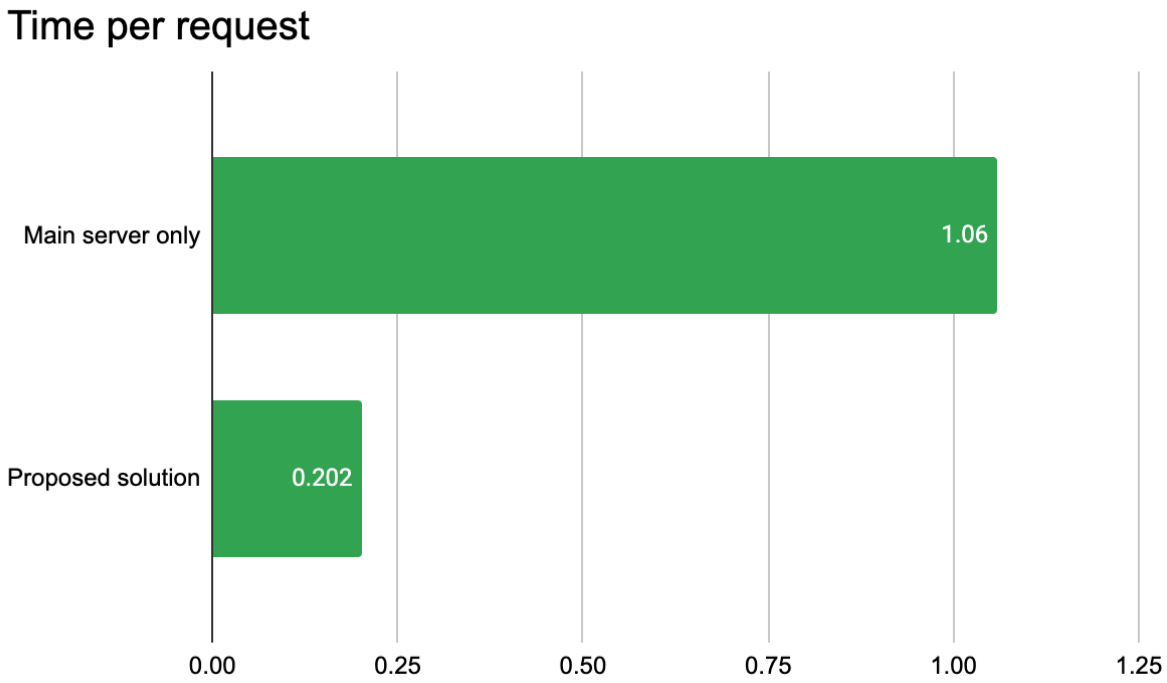

Рис. 3.16 Середній час відповіді сервера в секундах, менше - краще

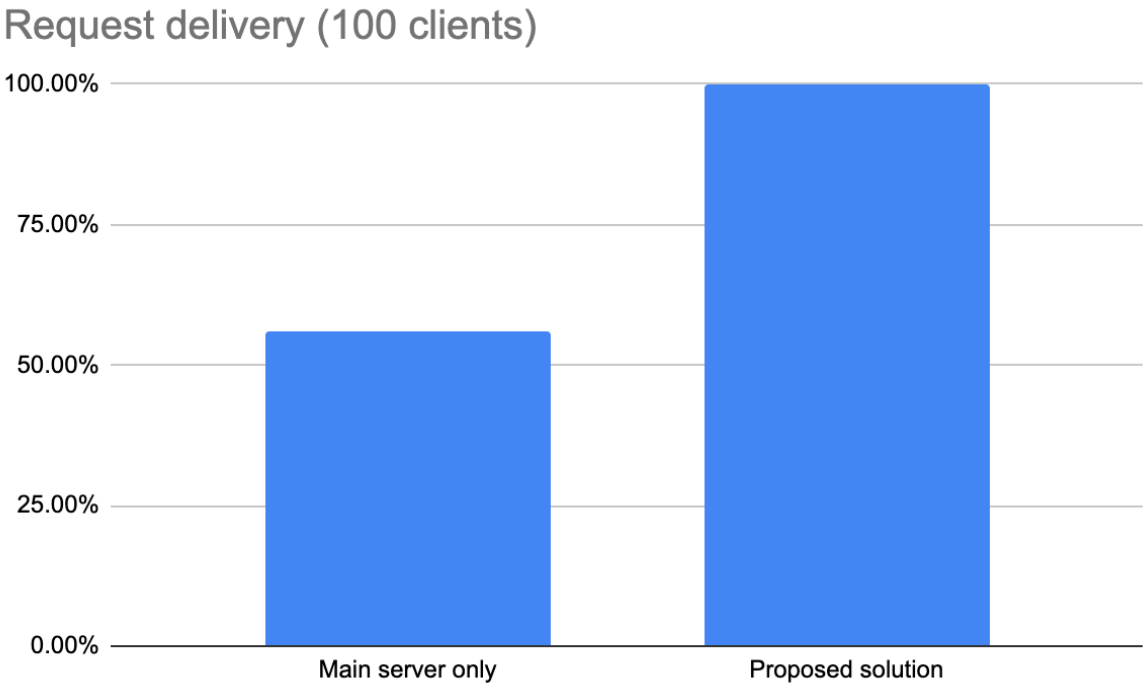

Рис. 3.17 Кількість успішно оброблених запитів при 10 клієнтах, більше - краще

В разі такого навантаження сервер власноруч не може обробити таку кількість запитів та витримати навантаження і 44% запитів були невдалими, що недопустимо для сервісу. В свою чергу в випадку з підключеним до мережі другим комп’ютером вдалося не тільки витримати навантаження, але й підвищити продуктивність сервера за рахунок збільшення успішно оброблених запитів за секунду.

Висновки

Залучення ресурсів пристроїв для побудови кластера може значно підвищити продуктивність сервера. Навіть з одним додатково підключеним комп’ютером можна уникнути необроблених запитів, зменшити середній час відповіді та підвищити кількість паралельно оброблених запитів за секунду.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

1. Проведений аналіз хмарних рішень з порівнянням їх функцій та цінових політик дозволив виявити основні можливості щодо скорочення витрат на інформаційну інфраструктуру організацій з існуючим парком комп'ютерного обладнання за рахунок залучення його не використаних обчислювальних ресурсів, а також дозволив сформулювати вимоги до системи, яка б дозволила це зробити.

2. Проведений порівняльний аналіз існуючих рішень, які у певній мірі можуть відповідати поставленим вимогам до системи, дозволив виділити найбільш перспективні для адаптування, але показав, що їхні недоліки роблять доцільним задачу створення власного рішення.

3. Експериментальне дослідження створеного у рамках роботи власного рішення на основі балансувальника Nginx показали, що навіть при додатковому залученні лише одного комп’ютера с існуючого парку обладнання, можна зменшити середній час відповіді та підвищити кількість паралельно оброблених запитів за секунду більш ніж на 40% що відповідає рівню альтернативних рішень, які потребують спеціально виділених для того обчислювальних потужностей, а також встановлення та конфігурування додаткових програмних компонентів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. New York, NY, Jan. 22, 2021. Facts and Factors. “Global Cloud Computing Market Size & Share Will Reach USD 1025.9 Billion by 2026” [Електронний ресурс] – Режим доступу до ресурсу: https://www.globenewswire.com/news-release/2021/01/22/2162789/0/en/Global-Cloud-Computing-Market-Size-Share-Will-Reach-USD-1025-9-Billion-by-2026-Facts-Factors.html

2. Найбільш поширені моделі хмарних сервісів [Електронний ресурс] – Режим доступу до ресурсу: https://www.stackscale.com/blog/cloud-service-models/

3. Аудиторії хмарних сервісів [Електронний ресурс] – Режим доступу до ресурсу: https://nachasi.com/tech/2017/10/02/cloud-servis/

4. Таблиця актуальних цін на віртуальні комп’ютери AWS EC2 [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/ru/ec2/pricing/on-demand/

5. Тарифи на хмарні віртуальні сервери GigaCloud [Електронний ресурс] – Режим доступу до ресурсу: https://gigacloud.ua/services/s-cloud

6. Розробка API на платформі застосунків NGINX [Електронний ресурс] – Режим доступу до ресурсу: https://www.nginx.com/blog/building-and-securely-delivering-apis-with-the-nginx-application-platform/

7. Документація API для динамічного конфігурування Nginx [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nginx.com/nginx/admin-guide/load-balancer/dynamic-configuration-api/

8. Чим Kubernetes відрізняється від PaaS платформ [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#what-kubernetes-is-not

9. Компоненти Kubernetes та їх призначення [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/components/

10. Функції Control plane в Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/components/#control-plane-components

11. Принцип роботи вузлів в Docker Swarm [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

12. Структура та компоненти кластеру Mesos [Електронний ресурс] – Режим доступу до ресурсу: https://subscription.packtpub.com/book/virtualization_and_cloud/9781788394383/2/ch02lvl1sec18/apache-mesos

13. Екосистема інтеграцій Nomad [Електронний ресурс] – Режим доступу до ресурсу: https://www.nomadproject.io/docs/ecosystem

14. Порівняння Nomad і Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://www.nomadproject.io/docs/nomad-vs-kubernetes

15. Jacob Lafors. Sep. 05, 2020, “Nomad: The workload orchestrator you may have missed” [Електронний ресурс] – Режим доступу до ресурсу: https://verifa.io/insights/nomad-the-workload-orchestrator-you-may-have-missed/

16. Принцип роботи та особливості Flocker [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ClusterHQ/flocker#flocker

17. Структурна схема кластеру Flocker https://ru.bmstu.wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Diagram-4.jpg

18. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/spotify/helios#why-helios

19. Chris Richardson, “Pattern: Microservice Architecture”, 2020, [Електронний ресурс] – Режим доступу до ресурсу: https://microservices.io/patterns/microservices.html
20. Nginx documentation, “What Is Load Balancing?” [Електронний ресурс] – Режим доступу до ресурсу: https://www.nginx.com/resources/glossary/load-balancing/#:~:text=A%20load%20balancer%20acts%20as,overworked%2C%20which%20could%20degrade%20performance.
21. Опитування Netcraft щодо використання веб серверів [Електронний ресурс] – Режим доступу до ресурсу: https://news.netcraft.com/archives/2020/12/22/december-2020-web-server-survey.html
22. Tony Mauro of F5, “Choosing an NGINX Plus Load-Balancing Technique”, October 29, 2015 [Електронний ресурс] – Режим доступу до ресурсу: https://www.nginx.com/blog/choosing-nginx-plus-load-balancing-techniques/
23. Noyes, Katherine (August 1, 2013). "Docker: A 'Shipping Container' for Linux Code" [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20130808043357/http:/www.linux.com/news/enterprise/cloud-computing/731454-docker-a-shipping-container-for-linux-code/
24. Опис роботи Docker на різних операційних системах [Електронний ресурс] – Режим доступу до ресурсу: https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/
25. Приклад CGI з документації Nginx, “FastCGI Example” [Електронний ресурс] – Режим доступу до ресурсу: https://www.nginx.com/resources/wiki/start/topics/examples/fastcgiexample/