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

факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

кафедра автоматика та управління в технічних системах
(повна назва кафедри)

«На правах рукопису»
УДК ___________

«До захисту допущено»
Завідувач кафедри
__________ О. І. РОЛІК
(підпис) (ініціали, прізвище)
“___”_____________2018 р.

Магістерська дисертація

зі спеціальності (спеціалізації) 126 «Інформаційні системи та технології»
(код і назва спеціальності)

на тему: Система управління обчислювальними ресурсами застосунків в умовах контейнерної віртуалізації

Виконав: студент 6 курсу, групи ІА–73мп

Загорулько Андрій Вікторович ____________
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник професор, д.т.н. Теленик С.Ф ____________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант доцент, к.т.н. Жаріков Е.В. ____________
(науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

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

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент ____________
((підпис)

Київ – 2018 року

Національний технічний університет України
"Київський політехнічний інститут
імені Ігоря Сікорського"

Факультет інформатики та обчислювальної техніки
(повна назва)

Кафедра автоматики та управління в технічних системах
(повна назва)

Ступінь вищої освіти – другий (магістерський)
(код, назва)

Спеціальність 126 «Інформаційні системи та технології»
(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри
__________ О. І. РОЛІК
(підпис) (ініціали, прізвище)

"___"_____________2018 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Загорулько Андрію Вікторовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Система управління обчислювальними ресурсами застосунків в умовах контейнерної віртуалізації

Науковий керівник дисертації Теленик Сергій Федорович, д. т. н. , професор

затверджено наказом по університету від " 29 " жовтня 2018р. №

2. Строк подання студентом дисертації " 4 " грудня 2018 р.

3. Об`єкт дослідження: система управління обчислювальниими ресурсами застосунків

4. Вихідні дані: а) загальна пам'ять серверів – 2 Tb; б) займана пам'ять контейнером – 2 Gb; в) навантаження на застосунок.

5. Зміст пояснювальної записки а) призначення та область застосування; б) вибір рішень для розгортання та управління віртуальними контейнерами; в) вибір програмного забезпечення для управління контейнерами; г) розробка алгоритму збалансованого розміщення контейнерів на фізичних машинах; д) створення алгоритмів з використанням арифметичної та геометричної прогресії

6. Перелік графічного (ілюстративного) матеріалу а) загальний алгоритм роботи системи; б) схема роботи алгоритму автоматичного розташування контейнерів на серверах; в) схема алгоритму розгортання контейнерів з

використанням арифметичної прогресії; г) схема алгоритму згортання контейнерів з використанням арифметичної прогресії; д) схема алгоритму розгортання контейнерів з використанням геометричної прогресії; е) схема алгоритму згортання контейнерів з використанням геометричної прогресії; ж) use case діаграма; и) структурна схема.

7. Консультанти розділів проекту:

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

8. Дата видачі завдання “ 29 ” жовтня 2018 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1	Огляд існуючих рішень та розробка структури системи	10.11.2018	
2	Аналіз технології контейнерної віртуалізації	15.11.2018	
3	Дослідження архітектури програмного забезпечення Kubernetes	20.11.2018	
4	Дослідження архітектури програмного забезпечення Docker	25.11.2018	
5	Побудова системи управління обчислювальними ресурсами застосунка в умовах контейнерної віртуалізації	29.11.2018	
6	Оформлення текстової та графічної документації	2.12.2018	
7	Представлення до захисту	4.12.2018	

Студент ____________ Загорулько А.В.
(підпис) (ініціали, прізвище)

Керівник проекту ____________ Теленик С.Ф
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Магістерська дисертація присвячена побудові системи управління обчислювальними ресурсами застосунків з використанням технології контейнерної віртуалізації. Для побудови системи було використано відкрите програмне забезпечення Kubernetes для автоматизації розгортання, масштабування і управління контейнеризованими додатками та програмного забезпечення Docker для автоматизованого запуску контейнерів в середовищі віртуалізації.

В роботі розглянуто архітектурні особливості двох типів технології віртуалізації та визначено основні відмінності здійснення віртуалізації з допомогою контейнерів та віртуальних машин.

Значну увагу в роботі приділено розгляду архітектури програмних забезпечень Kubernetes та Docker, їх алгоритмам роботи з розгортання, масштабування та видалення контейнерів з середовища.

Робота може бути корисною компаніям з розробки програмного забезпечення при побудові застосунків з необхідністю віртуалізації обчислювальних ресурсів, для центрів обробки даних або хостинг компаній, які надають свої обчислювальні потужності для користувачів, а також в навчальних дисциплінах при розгляді технологій віртуалізації.

ANNOTATION

The Master's degree paper is concentrated on the construction of a system for managing the computing resources of applications using the technology of container virtualization. To build the system, open source software Kubernetes was used to automate the deployment, scale, and management of containerized applications and Docker software for automated container run in the virtualization environment.

The paper considers the architectural peculiarities of two types of virtualization technology and identifies the main differences in the implementation of virtualization with the help of containers and virtual machines.

The prominent attention is paid considering the architecture of the software Kubernetes and Docker, their algorithms for deploying, zooming and removing containers from the environment.

The research can be useful to software development companies that build applications with the need for virtualization of computing resources, as well as in academic disciplines that consider virtualization technologies.

ЗМІСТ

ВСТУП.......... 9

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ.......... 11

2 ТЕХНІЧНА ХАРАКТЕРИСТИКА 12

3 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ РОЗГОРТАННЯ ТА УПРАВЛІННЯ ВІРТУАЛЬНИМИ КОТНЕЙНЕРАМИ 13

3.1 AWS ECS.......... 13

3.2 Nomad 14

3.3 Docker Swarm.......... 15

3.4 Kubernetes.......... 16

3.5 Rancher.......... 17

4 ВІРТУАЛІЗАЦІЯ ЯК СПОСІБ ОРГАНІЗАЦІЇ УПРАВЛІННЯ ОБЧИСЛЮВАЛЬНИМИ РЕСУРСАМИ ДОДАТКА 19

4.1 Поняття віртуалізації. Розвиток технології віртуалізації.......... 19

4.2 Підходи віртуалізації. Спільне та різне між контейнерами та віртуальними машинами 25

5 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗГОРТАННЯ ВІРТУАЛЬНИХ КОНТЕЙНЕРІВ ТА УПРАВЛІННЯ НИМИ 32

5.1 Програмне забезпечення Kubernetes. Архітектура платформи та її компоненти управління 32

5.2 Алгоритм роботи платформи Kybernetes та взаємозв’язок її компонентів.......... 38

5.3 Програмне забезпечення Docker. Архітектура платформи та її компоненти управління 54

5.4 Основні технології платформи Docker.......... 57

5.5 Моніторинг Docker-контейнерів.......... 60

6 ПРАКТИЧНА ЧАСТИНА 62

6.1 Обгрунтування системи 62
6.2 Алгоритм розміщення контейнерів 64
6.3 Алгоритм керування контейнерами з використанням арифметичної прогресії66
6.4 Алгоритм керування контейнерами з використанням геометричної прогресії 71
6.5 Аналіз роботи алгоритмів 76
6.5.1 Аналіз арифметичної прогресії 77
6.5.2 Аналіз геометричної прогресії 79
6.5.3 Порівняння ефективності алгоритмів, що застосовується 81
7 РОЗРОБКА СТАРТАП – ПРОЕКТУ 84
7.1 Опис ідеї проекту (товару, послуги, технології) 84
7.2 Технологічний аудит ідеї проекту 85
7.3. Аналіз ринкових можливостей запуску стартап-проекту 85
7.4. Розроблення ринкової стратегії проекту 92
7.5. Розроблення маркетингової програми стартап-проекту 94
ВИСНОВКИ 97
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 98
ДОДАТОК А 101
ДОДАТОК Б 102
ДОДАТОК В 103
ДОДАТОК Г 104
ДОДАТОК Д 105
ДОДАТОК Е 106
ДОДАТОК Ж 107
ДОДАТОК И 108

ПЕРЕЛІК СКОРОЧЕНЬ

ABAC - Attribute-Based Access Control

API - Application Programming Interface

CLI - Command Line Interface

CPU - Central Processing Unit

CRI - Container Runtime Interface

DNS - Domain Name System

HTTP - HyperText Transfer Protocol

ICMP - Internet Control Message Protocol

IP - Internet Protocol

iSCSI - Internet Small Computer System Interface

JSON - JavaScript Object Notation

LXC - Linux Containers

NFS - Network File System

RAM - Random Access Memory

RBAC - Role-Based Access Control

SDN - Software-defined networking

SSD - Solid-State Drive

TCP - Transmission Control Protocol

UID - User Identifier

VLAN - Virtual Local Area Network

ВМ - Віртуальна Машина

ОЗП - Оперативний Запам’ятовуючий Пристрій

ОС - Операційна Система

ПК - Персональний Комп’ютер

ЦП - Центральний Процесор

ВСТУП

Коли технологія віртуалізації ще не була так широко застосовувана як сьогодні, фізичні сервери використовувалися не дуже ефективно, тому що майже кожне окреме програмне забезпечення, встановлювалося на окремий фізичний сервер в основному для зручності адміністрування. Недоліки такого підходу полягають у наступному:

– низька утилізація фізичних серверів;

– висока вартість обслуговування;

– повільний введення в експлуатацію кожного нового додатка або сервісу;

– часто відсутність ізоляції між додатками.

Технологія віртуалізація обчислювальних ресурсів розвивається і існує вже досить тривалий час. Під віртуализацією розуміється технологія, яка надає можливість створення набору обчислювальних ресурсів, таких як сервери, пристрої зберігання даних або ж мережеві ресурси, абстрагованих від їх апаратної реалізації. Прикладами віртуалізації можуть бути:

– віртуальний сервер (віртуальна машина) або контейнер, що розпоряджаються частиною ресурсів процесора і пам'яті фізичного сервера;

– пристрої зберігання даних, які найчастіше реалізується за рахунок файлу спеціального формату, розташованого на розподіленій кластерній файловій системі (NAS), а віртуальна машина використовує цей файл, як локально підключений до неї жорсткий диск;

– мережеві ресурси, такі як мережевий протокол (VLAN, VxLAN) або SDN рішення (Nuage, NSX), що призначені для ізоляції мережевого трафіку в локальних мережах Ethernet, які використовуються для того, щоб ізолювати мережевий трафік між віртуальними машинами, а в разі використання SDN рішень ще й надати безліч додаткових віртуальних мережевих сервісів.

Контейнерна віртуалізація або віртуалізація на рівні операційної системи - це метод віртуалізації, при якому ядро операційної системи підтримує кілька ізольованих примірників простору користувача, замість одного. Це знижує накладні витрати і дозволяє використовувати віртуалізацію найбільш ефективно.

Контейнерна віртуалізація не використовує віртуальні машини, а створює віртуальне оточення з власним простором процесів і мережевим стеком. Примірники просторів користувача (часто звані контейнерами або зонами) з точки зору користувача повністю ідентичні реальному сервера, але вони в своїй роботі використовують один екземпляр ядра операційної системи. Ядро забезпечує повну ізольованість контейнерів, тому програми з різних контейнерів не можуть впливати один на одного.

Технології контейнерної віртуалізації на рівні додатків стають сьогодні дедалі популярнішими серед розробників і програмістів, однак їх повсюдне застосування в корпоративному середовищі не заввжи має перевагу над іншими технологіями. Причина цього - недостатня зрілість екосистеми і не вирішені до кінця питання безпеки. Але навіть при цьому контейнерна віртуалізація виявляється дуже привабливою для вирішення цілої низки завдань. Використання контейнерів допомагає вирішувати багато завдань швидше і дешевше. Швидше, бо контейнери можна використовувати на будь-якому обладнанні, моментально поширювати і модифікувати. Доступність контейнерних рішень забезпечується тим, що вони не несуть в собі нічого зайвого - тільки те, що потрібно для певного додатка або в робочому середовищі, через це вони займають менше серверних ресурсів, особливо - оперативної пам'яті. Віртуалізація на рівні операційної системи дає значно кращу продуктивність, масштабованість, щільність розміщення, динамічне управління ресурсами, а також легкість в адмініструванні, ніж в альтернативних рішеннях.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Система управління обчислювальними ресурсами застосунків в умовах контейнерної віртуалізації призначена для керування набором обчислювальних ресурсів, що мають логічну ізоляцію один від одного обчислювальних процесів та виконуються на одному фізичному ресурсі (сервері).

Головне завдання системи забезпечити оптимальне розгортання кількості контейнерів в залежності від навантаження на сервер.

2 ТЕХНІЧНА ХАРАКТЕРИСТИКА

Розроблювана система управління обчислювальними ресурсами розміщується на хості, який має наступні обчислювальні характеристики:

- SSD- 2048Gb (2 Tb);
- RAM-32Gb;
- Intel Xeon Silver 4110;
- швидкісний інтернет 1Gb/сек.

Надана потужність була розділена на 8 серверів, кожен з яких мав :

- SSD- 256Gb (2 Tb);
- RAM-4Gb.

Така конфігурація надає змогу більш ефективно масштабувати контейнери на кожному з серверів.

Вхідні параметри контейнера:

- займана пам’ять на сервері 4Gb.

3 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ РОЗГОРТАННЯ ТА УПРАВЛІННЯ ВІРТУАЛЬНИМИ КОТНЕЙНЕРАМИ

На сьогоднішній день існує величезна кількість різних інструментів і рішень по управлінню контейнерами: Kubernetes, Rancher, Docker Swarm, ECS, DC / OS, CoreOS, Tectonic, Nomad і т.д Кожен з перерахованих вище інструментів призначений для вирішення своєї конкретної задачі.

3.1 AWS ECS

Використання контейнерів замість віртуальних або поверх віртуальних машин дає можливість не тільки ще більш ефективно використовувати доступні обчислювальні ресурси, а й істотно збільшити швидкість розгортання поновлення контейнерного додатки. Однак використання Amazon ECS не дає можливість отримати ці переваги. Вся справа в тому, що у AWS дуже жорсткі вимоги, що пред'являються до можливості організувати безпеку інфраструктури кінцевого користувача. Як ви знаєте, всі контейнери, що запускаються на одному фізичному або віртуальному хості, використовують один і той же ядро хостової (де запускаються контейнери) операційної системи. Щоб надати своїм користувачам з одного боку використовувати інтерфейси управління контейнерами всередині своєї платформи, а з іншого боку забезпечити цим рішенням той же рівень безпеки, що і у EC2, AWS змушений запускати кожен окремий контейнер всередині окремої віртуальної машини, тобто запустити кілька контейнерів всередині однієї ВМ у вас не вийде. При цьому, якщо підключити CloudWatch або Newrelic і подивитися за реальною утилізацією обчислювальних потужностей, то ви легко побачите, що зазвичай 60-70% обчислювальних потужностей, відведених під контейнер, в ECS простоюють.

Щоб ще краще утилізувати орендовані у AWS обчислювальні потужності, найкраще використовувати власний кластер, побудований поверх наданої платформи віртуалізації, який вже буде керувати розміщеними всередині нього контейнерами. Найпримітивніші вимоги до такого рішення - це наявність

вбудованих засобів пошуку послуг, авто-масштабування, управління чутливими даними, поділ ресурсів, наявність зручних засобів управління і активність спільноти.

Отже, має сенс використовувати ECS тільки в тому випадку, якщо користувач збираєтеся розміщувати в кластері невеликі і досить прості Docker додатки. Як тільки додаток розростається до 10 або більше взаємодіюючих один з одним мікросервісов, стає складно організувати роботу так, щоб все працювало без збоїв.

3.2 Nomad

Nomad (від Hashicorp) являє собою всього один бінарний файл, який може працювати одночасно і як сервер, і як клієнт. Це рішення є планувальником ресурсів, кластеру якого не потрібні додаткові зовнішні служби для зберігання свого стану, а в разі відмови одного з майстрів в Nomad новий майстер вибирається автоматично. Всередині себе Nomad має всі необхідні абстракції управління, які дозволяють логічно розділити кластер на центри обробки даних, регіони і зони доступності. Більш того, планувальник в Nomad працює дуже швидко.

У Nomad є деякі недоліки:

– у складі рішення не вистачає служби пошуку послуг (service discovery), тому вам дійсно знадобиться Consul або etcd;

– Consul повинен бути встановлений на кожному вузлі кластера Nomad, etcd не обов'язково;

– немає рішення по управлінню чутливими секретними даними всередині кластера, тому потрібно встановлювати додаткове програмне забезпечення на зразок Vault-а;

– налаштування та підтримка кожної з вищеописаних окремих систем (Consul, etcd, Vault) - це окрема складна задача;

– у відкритій версії Nomad відсутня офіційний графічний користувальницький інтерфейс і панель моніторингу, які, наприклад, пропонують у своєму складі Kubernetes, Rancher і DC / OS. Даний факт буде однозначно обмежувати здатність швидко поглянути на стан хостів і сервісів всередині

кластера. Більш того, при використанні Nomad управляти всім доведеться тільки власними скриптами з командного рядка.

Використовувати Nomad має сенс тільки тоді, коли в інфраструктурі вже активно використовується Consul і Vault, і додавання Nomad не принесе в проект ще кілька додаткових незалежних рішень, які доведеться підтримувати.

3.3 Docker Swarm

Docker Swarm має вбудовані можливості по кластеризації, що дозволяють перетворити групу розрізнених Docker Engine в єдиний віртуальний Docker Engine. Структура типового кластера Swarm схожа на структуру кластера Nomad: кілька кластеризованих майстер-серверів, що забезпечують отказоустойчивую роботу кластера в цілому і підключення до них обчислювальні вузли, на яких запускаються контейнери. Істотний плюс використання Swarm в тому, що є можливість використовувати Docker Compose для запуску всього набору контейнерів і сервісів цілком.

Починаючи з версії Docker 1.12, Swarm вбудований в сам Docker Engine, і це "рідний" спосіб запуску кластера Docker для додатків. Більш того, Swarm забезпечує автоматичне TLS-шифрування між усіма вузлами кластера, і користувач "з коробки" отримує автоматичне ротирование всіх TLS-сертифікатів кластера кожні 90 днів.

Swarm також підтримує просте створення оверлейних мереж в порівнянні з Nomad і Kubernetes. Для останніх користувачеві буде потрібно на кожному вузлі встановити додаткові мережеві розширення на зразок Weave або Calico. Swarm дозволяє планувати запуск певних контейнерів на конкретних обчислювальних вузлах вашого кластера так само, як це робить за допомогою міток Kubernetes. Однак, в ббесплатной версії Swarm немає призначеного для користувача інтерфейсу. Проте, користувач може використовувати легкий і функціональний Portainer.io для управління всім кластером.

3.4 Kubernetes

Kubernetes - це рішення, призначене насамперед не тільки для обслуговування контейнерних додатків, але ще і для логічного поділу команд, що відповідають за інфраструктуру (адміністратори і DevOps інженери) і розробку (програмісти). Відмінною рисою Kubernetes в порівнянні з Swarm і Nomad є те, що при роботі з Kubernetes користувач оперує не самими контейнерами, а так званими подами (Pods) і іншими абстракціями на кшталт набору реплікації (RaplicaSet), набору демонів (DaemonSet), сервісів (Services) і т.д. Розуміння цих абстракцій на початковому етапі досить складне. Однак, як тільки призначення і взаємозв'язки сутностей Kubernetes стануть зрозумілі, буде очевидно, наскільки потужний і інструмент виявляється в руах користувача. Більш того, є можливість налаштувати Мультімастер конфігурацію керуючих вузлів, а також організувати федерацію (централізоване управління) з декількох кластерів.

У Kubernetes є дуже потужний Web-інтерфейс, що дозволяє:

– Запускати нові контейнери;

– Створювати служби (балансувальник навантаження, які будуть направляти трафік до відповідних контейнери);

– Управляти розгортання (запускати оновлення, замінювати поточні образи Docker додатки один за іншим або відразу групами);

– Масштабувати ваші програми вгору і вниз;

– Управляти секретнимічувствітельнимі даними і картами налаштувань;

– Управляти постійними дисками, використовуваними контейнерами;

– Переглядати пакетні завдання і набори демонів;

– Переглядати логи кожного контейнера;

– Легко переглядати, редагувати і оновлювати YAML опису ваших сервісів і контейнерів;

– Масштабувати кількість контейнерів;

– Встановлювати мінімальні вимоги до CPU і RAM.

У доповненні до Kubernetes легко встановити такі рішення призначені для аггрегации і моніторингу логів кластера як: Elasticsearch, Grafana і Kibana. Якщо ж

користувач запускає Kubernetes в GCE, то все логи можуть бути передані безпосередньо в Google Cloud Monitoring.

Одна з найбільш важливих особливостей, на яку заздалегідь необхідно звертати увагу - це мережева ізоляція між контейнерами. У Kubernetes для поділу контейнерних інфраструктур один від одного з одного боку існує концепція просторів імен (Namespaces), які логічно ділять додатки, служби і все інше. З іншого боку, щоб розмежувати мережу між просторами імен, буде необхідно встановити один з полігонів для оверлейной мережі, наприклад Calico, Weave або Flannel.

3.5 Rancher

Платформа мають відмінний web-інтерфейс, безліч плагінів і функцій, активно розробляються і підтримуються спільнотою. Rancher може полегшити управління складними середовищами, розміщеними в декількох хмарах. Також платформа може бути підключена іншими планувальникам контейнерів, тому що Docker Swarm, Kubernetes або Apache Mesos, якщо користувачеві потрібні будь-які додаткові функції крім тих, які реалізовані вбудованими планировщиками.

Починаючи з версія Rancher 1.3 початку експериментальну підтримку контейнерів в Windows оточенні. Додатки, які можуть бути встановлені з каталогу в Rancher, залежать від обраної платформи оркестрації. На кнопці може відображатися мітка «не сумісно», якщо додаток не може працювати на поточній платформі. Однак, завжди можна створити додаткове «оточення» і вибрати в ньому іншу платформу оркестрації. Цей факт дає велику перевагу Rancher при виборі його в якості основи для інфраструктури.

Rancher також забезпечує легке обслуговування кластерів: досить просто перемикатися між налаштуваннями Cattle, Kubernetes, Swarm або Mesos в меню, а потім додавати робочі вузли в конкретний кластер і запускати необхідні додатки в найкоротші терміни.

На додаток до свого привабливому призначеному для користувача інтерфейсу Rancher також пропонує важливі функції Enterprise рівня, такі як журнал аудиту, сховище сертифікатів і авторизацію користувачів на основі ролей. Користувач

також може налаштувати контроль доступу за допомогою Active Directory, Azure AD, GitHub і інших провайдерів аутентифікації, щоб дозволити розробникам або членам DevOps команди отримати обмежений доступ до певних кластерів і ресурсів.

4 ВІРТУАЛІЗАЦІЯ ЯК СПОСІБ ОРГАНІЗАЦІЇ УПРАВЛІННЯ ОБЧИСЛЮВАЛЬНИМИ РЕСУРСАМИ ДОДАТКА

4.1 Поняття віртуалізації. Розвиток технології віртуалізації

Згідно зі статистикою [1], середній рівень завантаження процесорних потужностей у серверів під управлінням Windows не перевищує 10%, у Unixсистем цей показник краще, але, тим не менше, в середньому не перевищує 20%. Низька ефективність використання серверів пояснюється широко застосовуваним з початку 90-х років підходом "один додаток - один сервер", тобто кожного разу для розгортання нової програми компанія купує новий сервер. Очевидно, що на практиці це означає швидке збільшення серверного парку і, як наслідок - зростання витрат на його адміністрування, енергоспоживання та охолодження, а також потреби в додаткових приміщеннях для установки все нових серверів і придбання ліцензій на серверну ОС.

Віртуалізація ресурсів фізичного сервера дозволяє гнучко розподіляти їх між додатками, кожен з яких при цьому "бачить" тільки призначені йому ресурси і "вважає", що йому виділено окремий сервер, тобто в даному випадку реалізується підхід "один сервер - кілька додатків", але без зниження продуктивності, доступності та безпеки серверних додатків. Крім того, рішення віртуалізації дають можливість запускати в розділах різні ОС за допомогою емуляції їх системних викликів до апаратних ресурсів сервера. В основі віртуалізації лежить можливість одного комп'ютера виконувати роботу декількох комп'ютерів завдяки розподілу його ресурсів за кількома середовищами. За допомогою віртуальних серверів і віртуальних настільних комп'ютерів можна розмістити кілька ОС і кілька додатків в єдиному місці розташування. Таким чином, фізичні і географічні обмеження перестають мати будь-яке значення. Крім енергозбереження та скорочення витрат завдяки більш ефективному використанню апаратних ресурсів, віртуальна інфраструктура забезпечує високий рівень доступності ресурсів, більш ефективну систему управління, підвищену безпеку і вдосконалену систему відновлення в критичних ситуаціях.

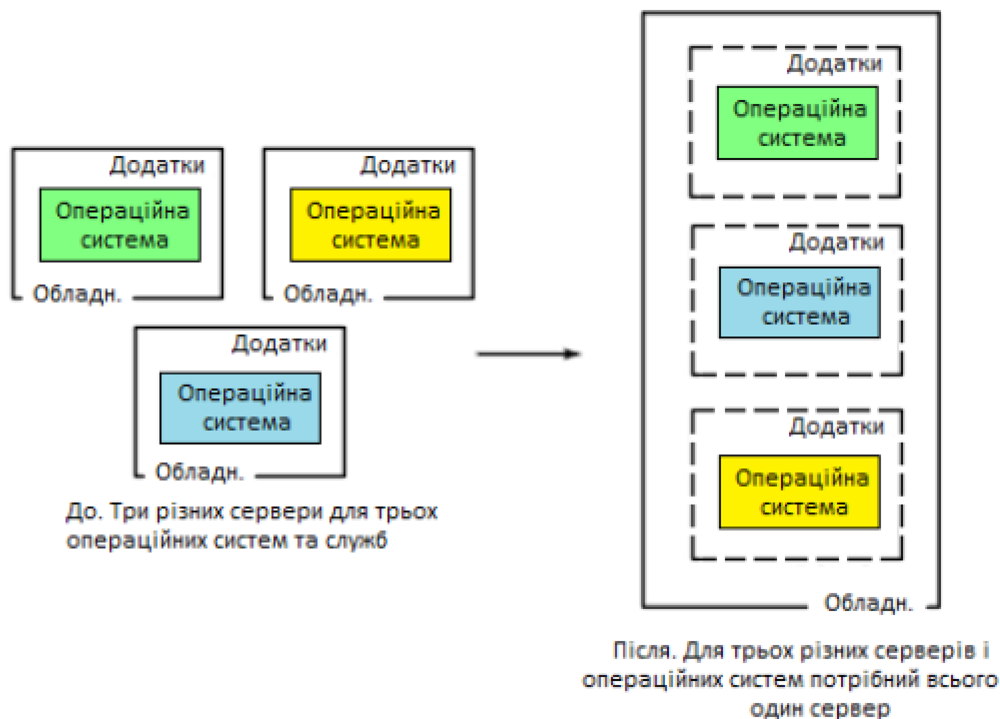


Рисунок 4.1 – Віртуалізація - перехід від фізичного відокремлення серверів до логічного

У широкому сенсі, поняття віртуалізації являє собою приховування справжньої реалізації будь-якого процесу або об'єкта від істинного його представлення для того, хто ним користується. Продуктом віртуалізації є щось зручне для використання і яке насправді, має більш складну або зовсім іншу структуру, відмінну від тієї, яка сприймається при роботі з об'єктом. Іншими словами, відбувається відділення представлення від реалізації чого-небудь. Віртуалізація покликана абстрагувати програмне забезпечення від апаратної частини.

У комп'ютерних технологіях під терміном "віртуалізація" зазвичай розуміється абстракція обчислювальних ресурсів і надання користувачеві системи, яка "інкапсулює" (приховує в собі) власну реалізацію. Простіше кажучи, користувач працює із зручним для себе поданням об'єкта, і для нього не має значення, як об'єкт влаштований в дійсності. Зараз можливість запуску декількох віртуальних машин на одній фізичній викликає великий інтерес серед комп'ютерних фахівців, не тільки

тому, що це підвищує гнучкість IT-інфраструктури, а й тому, що віртуалізація, насправді, дозволяє економити гроші.

Історія розвитку технологій віртуалізації налічує понад сорок років. Компанія IBM була першою, хто задумався про створення віртуальних середовищ для різних призначених для користувача завдань, тоді ще в мейнфреймах. У 60-х роках минулого століття віртуалізація представляла чисто науковий інтерес і була оригінальним рішенням для ізоляції комп'ютерних систем в рамках одного фізичного комп'ютера. Після появи персональних комп'ютерів інтерес до віртуалізації дещо послабився через бурхливого розвитку операційних систем, які пред'являли адекватні вимоги до апаратного забезпечення того часу. Однак бурхливе зростання апаратних потужностей комп'ютерів в кінці дев'яностих років минулого століття змусив IT-спільнота знову згадати про технології віртуалізації програмних платформ. У 1999 р компанія VMware представила технологію віртуалізації систем на базі x86 в якості ефективного засобу, здатного перетворити системи на базі x86 в єдину апаратну інфраструктуру загального користування та призначення, що забезпечує повну ізоляцію, мобільність і широкий вибір ОС для прикладних середовищ. Компанія VMware була однією з перших, хто зробив серйозну ставку виключно на віртуалізацію. Як показав час, це виявилося абсолютно виправданим. Сьогодні VMware пропонує комплексну віртуалізаційну платформу четвертого покоління VMware vSphere 4, яка включає засоби як для окремого ПК, так і для центру обробки даних. Ключовим компонентом цього програмного комплексу є гіпервізор VMware ESX Server. Пізніше в "битву" за місце в цьому модному напрямку розвитку інформаційних технологій включилися такі компанії як Parallels (раніше SWsoft), Oracle (Sun Microsystems), Citrix Systems (XenSourse).

Корпорація Microsoft вийшла на ринок засобів віртуалізації в 2003 році з придбанням компанії Connectix, випустивши свій перший продукт Virtual PC для настільних ПК. З тих пір вона послідовно нарощувала спектр пропозицій в цій галузі і на сьогодні майже завершила формування віртуалізаційної платформи, до складу якої входять такі рішення як Windows 2008 Server R2 з компонентом Hyper-

V, Microsoft Application Virtualization (App-v), Microsoft Virtual Desktop Infrastructure (VDI), Remote Desktop Services, System Center Virtual Machine Manager. Підвищений інтерес до технологій віртуалізації в даний час не випадковий. Обчислювальна потужність нинішніх процесорів швидко зростає, і питання навіть не в тому, на що цю міць витрачати, а в тому, що сучасна "мода" на двоядерні і багатоядерні системи, що проникла вже і в персональні комп'ютери (ноутбуки та десктопи), як не можна краще дозволяє реалізувати багатющий потенціал ідей віртуалізації операційних систем і додатків, виводячи зручність користування комп'ютером на новий якісний рівень. Технології віртуалізації стають одним з ключових компонентів (в тому числі, і маркетингових) в найновіших і майбутніх процесорах Intel і AMD, в операційних системах від Microsoft і ряду інших компаній.

Основними перевагами використання технології віртуалізації є:

– Ефективне використання обчислювальних ресурсів: замість 3х, чи більше серверів, завантажених на 5 - 20 % можна використовувати один, використовуваний на 50 - 70 %. Крім іншого, це економія електроенергії, а також значне скорочення фінансових вкладень: придбавається один високотехнологічний сервер, що виконує функції 5 - 10 серверів. За допомогою віртуалізації можна досягти значно більш ефективного використання ресурсів, оскільки вона забезпечує об'єднання стандартних ресурсів інфраструктури в єдиний пул і долає обмеження застарілої моделі «один додатків на один сервер».

– Скорочення витрат на інфраструктуру: віртуалізація дозволяє скоротити кількість серверів і пов'язаного з ними IT -обладнання в інформаційному центрі. У результаті цього потреби в обслуговуванні, електроживленні й охолодженні матеріальних ресурсів скорочуються, і на IT витрачається значно менше коштів.

– Зниження витрат на програмне забезпечення: деякі виробники програмного забезпечення ввели окремі схеми ліцензування спеціально для віртуальних середовищ. Так, наприклад, купуючи одну ліцензію на Microsoft Windows Server 2008 Enterprise, ви отримуєте право одночасно її використовувати на 1 фізичному сервері і 4 віртуальних (в межах одного сервера), а Windows Server 2008 Datacenter

ліцензується тільки на кількість процесорів і може використовуватися одночасно на необмеженій кількості віртуальних серверів.

– Підвищення гнучкості і швидкості реагування системи: віртуалізація пропонує новий метод управління IT – інфраструктурою і допомагає IT – адміністраторам затрачати менше часу на виконання повторюваних завдань – наприклад, на ініціацію, налаштування, відстеження і технічне обслуговування. Багатосистемні адміністратори відчували неприємності, коли «валиться» сервер. І не можна, витягнувши жорсткий диск, переставивши його в інший сервер, запустити все як раніше. При використанні віртуального сервера – можливий моментальний запуск на будь-якому “залізі“, а якщо немає подібного сервера, то можна скачати готову віртуальну машину зі встановленим та настроєним сервером, з бібліотек, підтримуваних компаніями розробниками гіпервізорів (програм для віртуалізації).

– Несумісні додатки можуть працювати на одному комп'ютері: при використанні віртуалізації на одному сервері можлива установка linux і windows серверів, шлюзів, баз даних і інших абсолютно несумісних в рамках однієї не віртуалізованої системи додатків.

– Підвищення доступності додатків і забезпечення безперервності роботи підприємства: завдяки надійній системі резервного копіювання та міграції віртуальних середовищ цілком без перерв в обслуговуванні ви зможете скоротити періоди планового простою і забезпечити швидке відновлення системи в критичних ситуаціях. "Падіння» одного віртуального сервера не веде до втрати інших віртуальних серверів. Крім того, у разі відмови одного фізичного сервера можливо зробити автоматичну заміну на резервний сервер. Причому це відбувається не помітно для користувачів без перезагрузки. Тим самим забезпечується безперервність бізнесу.

– Можливості легкої архівації: оскільки жорсткий диск віртуальної машини зазвичай представляється у вигляді файлу певного формату, розташований на якому -небудь фізичному носії, віртуалізація дає можливість простого копіювання цього файлу на резервний носій як засіб архівування і резервного копіювання всієї віртуальної машини цілком. Можливість підняти з архіву сервер повністю – ще одна

чудова особливість. Можливо підняти сервер з архіву, не знищуючи поточний сервер і подивитися стан справ за минулий період.

– Підвищення керованості інфраструктури: використання централізованого управління віртуальною інфраструктурою дозволяє скоротити час на адміністрування серверів, забезпечує балансування навантаження і "живу" міграцію віртуальних машин.

Однак, застосування засобів віртуалізації, як і будь-якого складного продукту, пов'язане з низкою проблем:

По-перше, в процесі реалізації деяких проектів доводиться стикатися з ситуаціями, коли засоби віртуалізації не дозволяють прикладному ПЗ або інформаційним системам клієнта працювати з периферійними пристроями. Інакше кажучи, проміжний рівень представлення даних вносить певні труднощі в роботу тих чи інших сервісів. Це означає, що використання віртуалізації вимагає підготовки і ретельного тестування пілотного проекту. Тобто необхідно зібрати пул необхідного периферійного обладнання, пакет прикладного ПЗ, реалізувати заплановану архітектуру і домогтися повноцінної роботи і функціональності тестованого рішення.

По-друге, необхідно враховувати витрати обчислювальної потужності сервера на віртуалізацію. Щоб віртуалізація була ефективною, потрібно дуже уважно стежити за завантаженням віртуальних серверів і своєчасним виділенням достатніх для оптимальної роботи ресурсів. Справа в тому, що при пікових навантаженнях обробка конкурентного доступу до ресурсів викликає зростання накладних витрат на віртуалізацію. Тому все віртуальні сервери необхідно ретельно налаштовувати, для чого зазвичай використовуються засоби моніторингу навантаження, якими оснащуються всі віртуальні сервери. Крім того, існує маса програмних засобів, що дозволяють перерозподілити навантаження між серверами, встановленими на різних комп'ютерах.

По-третє, основна мета віртуалізації - оптимізувати використання апаратного забезпечення. Для цього адміністратор системи повинен стежити за завантаженістю віртуальних серверів і при необхідності перерозподіляти обчислювальні ресурси

між ними або самі сервери переміщати між апаратними платформами. Саме тому застосування засобів віртуалізації, як правило, передбачає використання як мінімум двох комплектів апаратних засобів. Подібне рішення корисно ще й тим, що позбавляє нас від збоїв апаратури. Таким чином, під час налаштування віртуальних серверів необхідно застосовувати спеціальні засоби визначення завантаженості, що допомагають встановлювати оптимальний баланс навантаження і, крім того, рекомендується використовувати такі архітектурні рішення засобів віртуалізації, які встановлюються на порожню апаратну платформу.

4.2 Підходи віртуалізації. Спільне та різне між контейнерами та віртуальними машинами

Існують два підходи до створення незалежних ізольованих обчислювальних просторів на одному фізичному сервері: віртуальні машини, яким потрібен гипервизор, і віртуальні контейнери.

Віртуальна машина - це повністю ізольований програмний контейнер, який працює з власною ОС і додатками, подібно фізичному комп'ютеру. Віртуальна машина діє так само, як фізичний комп'ютер, і містить власні віртуальні (тобто програмні) ОЗП, жорсткий диск і мережевий адаптер. ОС не може розрізнити віртуальну і фізичну машини. Те ж саме можна сказати про додатки та інших комп'ютерах в мережі. Навіть сама віртуальна машина вважає себе "справжнім" комп'ютером. Але незважаючи на це віртуальні машини складаються виключно з програмних компонентів і не включають обладнання.

На відміну від віртуальної машини, що забезпечує апаратну віртуалізацію, контейнер забезпечує віртуалізацію на рівні операційної системи за допомогою абстрагування «користувацького простору». В цілому, контейнери виглядають як віртуальні машини. Наприклад, у них є ізольований простір для запуску додатків, вони дозволяють виконувати команди з правами суперкористувача (root), мають приватний мережевий інтерфейс і IP-адреса, призначені для користувача маршрути і правила брандмауера і т. д.

Відповідно, віртуалізація з допомогою віртуальних машин дозволяє створювати неоднорідні обчислювальні середовища на одному комп'ютері, віртуалізація з використанням віртуальних контейнерів - тільки однорідні.

Однак, оскільки віртуальні машини включають операційну систему, їх розмір може складати кілька гігабайт. Проте недоліком є те, що для завантаження ОС і ініціалізації програми, яке в них розміщено, потрібно відносно більше часу. Контейнери легші і, в основному, їх розмір образу вимірюється в мегабайтах.

Порівнюючи продуктивність контейнерів з віртуальними машинами, то контейнери можуть запускатися майже миттєво. При виборі між контейнерами та віртуальними машинами слід враховувати цілі, які потрібно досягти.

На рисунку 4.2 видно, що контейнери упаковують тільки простір для користувача, а не ядро або віртуальну апаратуру, як це роблять віртуальні машини. Кожен контейнер отримує своє власне ізольоване призначене для користувача простір для забезпечення можливості запуску кількох контейнерів на одному хості. Архітектура рівня операційної системи розділяється між контейнерами, саме тому контейнери настільки легковажні.

Спільні риси між контейнерами та віртуальними машинами наведені нижче:

– Ізольоване оточення: як і віртуальні машини, контейнери гарантують ізоляцію файлової системи, змінних оточення, реєстру і процесів між додатками. Це означає, що, як і віртуальна машина, кожен контейнер створює ізольоване оточення для всіх додатків всередині себе. При міграції та контейнери, і віртуальні машини зберігають не тільки додатки всередині, але також і контекст цих додатків.

– Міграція між хостами: велика перевага роботи з віртуальними машинами в тому, що можна зстосувати живу міграцію зліпків віртуальних машин між гіпервізорами, при цьому не потрібно змінювати їх вміст, що дає змогу не зупиняти машину. Це справедливо і для контейнерів. Там, де віртуальні машини можна "мігрувати" між різними гіпервізорами, контейнери можна "мігрувати" між різними хостами контейнерів. При «мігруванні" обох видів артефактів між різними хостами вміст віртуальної машини/контейнера залишається таким же, як і на попередніх хостах.

– Управління ресурсами: інша спільна риса - це те, що доступні ресурси (ЦП, ОЗУ, пропускна здатність мережі) як контейнерів, так і віртуальних машин можуть бути обмежені до заданих значень. В обох випадках це управління ресурсами може здійснюватися тільки на стороні хоста контейнера або гипервизора. Управління ресурсами гарантує, що контейнер отримує обмежені ресурси, щоб звести до мінімуму ризик того, що він вплине на продуктивність інших контейнерів, які виконуються на тому ж самому хості. Наприклад, контейнеру можна задати обмеження, що він не може використовувати більше 10% ЦП.

Різниця між контейнерами та віртуальними машинами полягає у наступному:

– Рівень віртуалізації: контейнери - це новий рівень віртуалізації. Якщо поглянути на історію віртуалізації, то вона почалася з таких понять, як віртуальна пам'ять і віртуальні машини. Контейнери - це новий рівень цієї тенденції віртуалізації. Там, де віртуальні машини забезпечують віртуалізацію апаратного забезпечення, контейнери забезпечують віртуалізацію платформ. Це означає, що якщо віртуалізація апаратного забезпечення дозволяє віртуальній машині вірити, що її апаратні ресурси належать тільки їй, віртуалізація платформ дозволяє контейнеру вірити, що вона належить тільки йому. У контейнерів, наприклад, немає власного режиму ядра. З цієї причини контейнери не видно як віртуальні машини і вони також не розпізнаються, як віртуальні машини всередині операційної системи.

– Взаємодія з ОС: інше важлива відмінність між контейнерами та віртуальними машинами полягає в способі, яким вони взаємодіють з режимом ядра. Тоді, як у віртуальних машин є повноцінна ОС (і виділений режим ядра), контейнери поділяють ядро з іншими контейнерами і з хостом контейнерів. В результаті, контейнери повинні орієнтуватися на ядро хоста контейнерів, в той час, як віртуальна машина може вибрати будь-яку ОС (версію і тип), яку забажає. Там, де віртуальні машини можуть запускати ОС Linux на гіпервізора Windows, з технологією контейнерів неможливо запустити контейнер Linux на хості контейнерів Windows, і навпаки. (В Windows 10 можна запускати контейнери Linux, але вони запускаються всередині віртуальної машини Linux)

– Модель зростання: контейнери поділяють ресурси хоста контейнера, і створюються на основі образу, який містить рівно те, що потрібно для запуску застосування. Віртуальні машини створюються в зворотному порядку. Найчастіше починають з повною операційної системи і, в залежності від програми, прибирають речі, які не потрібні.

–

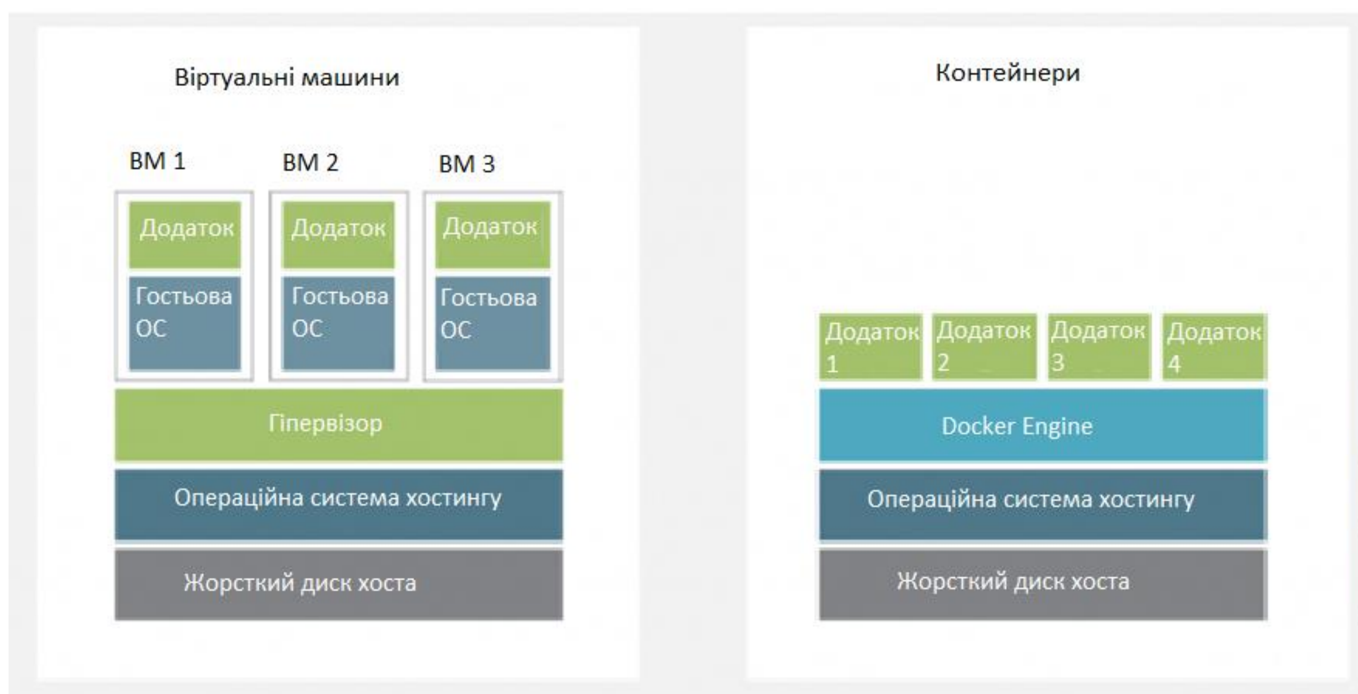


Рисунок 4.2 – Підходи до створення ізольованих обчислювальних машин на одному фізичному сервері

4.3 Поняття контейнерної віртуалізації. Переваги та недоліки використання технології контейнерної віртуалізації

Віртуалізація на рівні операційної системи - це метод віртуалізації [2], при якому ядро операційної системи допускає існування декількох ізольованих примірників робочих просторів, а не тільки одного. Такі екземпляри можуть виглядати як справжні комп'ютери з точки зору програм, запущених в них. Такий підхід корисний, коли необхідно налаштувати парк ОС з ідентичними конфігураціями. Різні програми можуть бути встановлені, налаштовані і можуть

виконуватися так само, як якщо запускається додаток на ОС хості. Ресурси, призначені контейнеру, видні тільки йому.

Для створення контейнерів ОС можна використовувати такі контейнерні технології, як LXC, OpenVZ, Linux VServer, BSD Jails і Solaris.

Віртуалізація додатків - це програмна технологія, яка інкапсулює комп'ютерні програми з базової операційної системи, на якій вона виконана. Повністю віртуалізований додаток не встановлено в традиційному сенсі, хоча воно все одно виконується так, як якщо б воно було встановлено. Дії додатку мають вид під час виконання, так як ніби воно безпосередньо взаємодіє з вихідної операційною системою і всіма ресурсами, якими вона управляє, але може бути ізольованим в різному ступені. Контейнери додатків призначені для упаковки і запуску служб, як одного процесу, тоді як в контейнерах ОС можуть виконуватися кілька сервісів і процесів.

Контейнерні технології, такі як Docker і Rocket, є прикладами контейнерів для додатків. На рисунку 4.3 наведено схематичне зображення архітектури двох типів контейнерної віртуалізації.

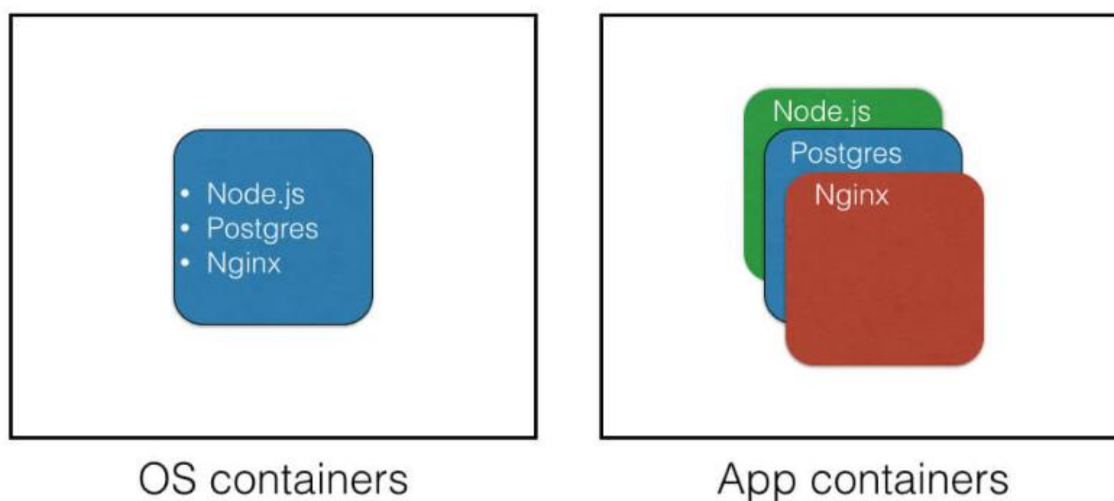


Рисунок 4.3 – Приклади контейнерної віртуалізації

Переваги використання контейнерів, в якості засобу віртуалізації полягають у наступному:

– Гнучке середовище: найбільша перевага у використанні технологій контейнерів полягає в тому, що їх можна створювати набагато швидше, ніж екземпляри віртуальних машин. Їх легка вага забезпечує менші накладні витрати з точки зору продуктивності і розміру;

– Підвищена продуктивність: контейнери підвищують продуктивність розробників за рахунок усунення міжмережевих залежностей і конфліктів. Кожен контейнер може розглядатися як окремий мікросервіс і, отже, може бути незалежно оновлений з мінімальними проблемами з синхронізацією;

– Управління версіями дозволяє відстежувати версії контейнера, стежити за відмінностями між ними і т.д;

– Переносимість середовища обчислень: контейнери інкапсулюють всі відповідні деталі, такі як залежності додатків і операційні системи, необхідні для запуску програми. Це полегшує переносимість способу контейнера з одного середовища в іншу. Наприклад, один і той же образ можна використовувати для роботи в середовищі Windows / Linux або dev / test / stage;

– Стандартизація: більшість контейнерів засновані на відкритих стандартах і можуть працювати в усіх основних дистрибутивах Linux, Microsoft і т. д.;

– Безпека: контейнери ізолюють процеси одного контейнера від одного і від базової інфраструктури. Таким чином, будь-яке оновлення або зміна в одному контейнері не впливає на інший контейнер.

Недоліками при використанні контейнерів є:

– Підвищена складність: при n числі контейнерів, які працюють з додатком, також збільшується коефіцієнт складності[3]. Управління безліччю контейнерів може бути складним завданням у виробничому середовищі. Такі інструменти, як Kubernetes і Mesos, можуть полегшити управління великою кількістю контейнерів;

– Розмір контейнера. Складність полягає в тому, що зазвичай в контейнер завантажуються набагато більше ресурсів, ніж потрібно, а це призводить до розростання образу і великому розміру контейнера;

– Підтримка Native Linux: більшість контейнерних технологій, таких як Docker, засновані на Linux-контейнерах (LXC). Тому запуск цих контейнерів в

середовищі Microsoft - є моторошною дією, а їх щоденне використання може викликати складності в порівнянні з початковим запуском цих примірників на Linux.

5 ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗГОРТАННЯ ВІРТУАЛЬНИХ КОНТЕЙНЕРІВ ТА УПРАВЛІННЯ НИМИ

5.1 Програмне забезпечення Kubernetes. Архітектура платформи та її компоненти управління

Kubernetes є проектом з відкритим вихідним кодом, призначеним для управління кластером контейнерів Linux як єдиною системою. Kubernetes управляє і запускає контейнери Docker на великій кількості хостів, а також забезпечує спільне розміщення та реплікацію великої кількості контейнерів. Розгортання контейнерів відбувається автоматично, що покращує утилізацію ресурсів і економить гроші. Потужні примітиви розгортання (Deployment Primitives) дозволяють поступово вивантажувати новий код, а контексти безпеки (Security Contexts) і мережеві політики (Network Policies) - безпечно запускати різні проекти в одному кластері. Проект був розпочатий Google і тепер підтримується багатьма компаніями, серед яких Microsoft, RedHat, IBM і Docker.

Зараз Kubernetes займає лідируючі позиції і пропонує безліч цікавих рішень. Одне з основних його переваг - те, що інженерам не потрібно знати, на яких машинах працюють їх застосування. Розподілені системи по-справжньому складні, і управління їх службами - одна з найбільших проблем, з якими стикаються операційні групи.

У Kubernetes реалізовані всі функції, необхідні для запуску додатків на основі Docker в конфігурації з високою доступністю (кластери більше 1000 вузлів, з multi-availability і multi-region зонами): управління кластером, планування, виявлення сервісів, моніторинг, управління обліковими даними та ін. Для реалізації цього використовується більше десятка сторонніх взаємодіючих послуг, які разом забезпечують необхідну функціональність. Наприклад, за координацію і зберігання налаштувань відповідає etcd, створення мереж між контейнерами - flannel. Це дещо ускладнює початкові установки, але дозволяє при необхідності просто замінити будь-який компонент. Для взаємодії служб використовуються різні CLI, API, які вже спільно реалізують API більш високого рівня для сервісних функцій, таких як

планування ресурсів. Потрібна функціональність повинна бути спеціально адаптована для Kubernetes. Наприклад, звернутися безпосередньо до API Docker не можна, слід використовувати Docker Compose.

Kubernetes має такі особливості:

– самостійно надає контейнерам свої власні IP-адреси і єдине ім'я DNS для набору контейнерів, і може розподіляти навантаження між ними;

– автоматично розміщує контейнери на основі їх вимог до ресурсів та інших обмежень;

– автоматично монтує систему зберігання на вибір користувача, незалежно від локального сховища, постачальника публічної хмари, такого як GCP або AWS , або мережеву систему зберігання даних, таку як NFS, iSCSI, Gluster, Ceph, Cinder або Flocker;

– самостійно перезавантажує контейнери, які зазнають невдачі, замінюють та перерозподіляють контейнери, коли вузли гинуть, виводить контейнери, які не відповідають запитуванню вами контрольної перевірки користувача, і не відображає їх клієнтам, поки вони не готові працювати;

– поступово викладає зміни до програми або її конфігурації, одночасно стеживши за збереженням роботоздатності програм, щоб гарантувати, що вони не видалять всі ваші екземпляри одночасно. Якщо щось не так, Kubernetes відкине зміну;

– секретне та конфігураційне управління дає можливість розгортати та оновлювати секрети та конфігурування додатків, не відновлюючи зображення та не викриваючи секрети у конфігурації стека;

– самостійно керує завантаженнями користувацьких пакетів, замінюючи контейнери, що не працюють, за бажанням;

– можливість масштабувати вгору і вниз за допомогою простої команди, з інтерфейсом користувача або автоматично на основі використання процессора;

Архітектура Kubernetes, що зображена на рисунку 5.1, створена таким чином, що дозволяє розширювати цю платформу та складається з двох вузлів: головного

сервера (master server) і великої кількості підлеглих серверів, які називаються міньйонами або нодами (minions / nodes).

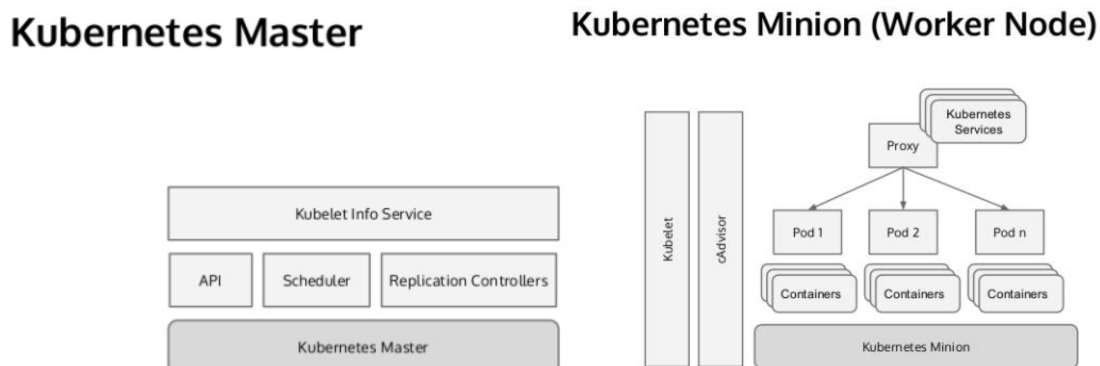


Рисунок 5.1 – Архітектура головного (Kubernetes Master) та підлеглого (Kuberntes Minion) серверів

Головний сервер – це сервер з сервісом Kubernetes API. Майстер сервер містить компоненти, які забезпечують рівень управління кластера. Наприклад, майстер-компоненти несуть відповідальність за глобальні рішення (наприклад, планування), виявлення та реагування на події всередині кластера (наприклад, запуск нового pod'a). Майстер-компоненти сервера можуть запускатися на будь-якому хості кластера, і працювати одночасно з компонентами користувача, проте, якщо хост обраний в ролі майстра, на Майстрі вимикаються всі компоненти користувача і в робочих залишаються тільки компоненти рівня управління. До майстер-компонентів відносяться: kube-apiserver, etcd, kube-controller-manager, kube-scheduler, різного роду додатки (провайдери DNS, kube-ui, fluentd-elasticsearch і система моніторингу cluster-monitoring). Конфігурація майстра може відбуватися через командний рядок, для цього розгортається сервіс kubecfg

Архітектуру системи можна розбити на сервіси, які працюють на кожній ноді і сервіси рівня управління кластера. На кожній Node Kubernetes запускаються сервіси, необхідні для управління нодою з боку майстра і для запуску додатків. Також, на кожній Node Kubernetes запускається Docker, який забезпечує регулярність образів і запуск контейнерів.

Кожен вузол має свій статус, який визначається такими параметри:

– Адреса вузла: хост-ім'я, зовнішній IP, внутрішній IP;

– Фаза вузла: поточний етап життєвого циклу вузла: виконується, знищений, в очікуванні;

– Стан вузла: описує стан виконання вузла. Існує один стан - Ready і його три властивості: True (готовий до прийому pod'a), False (є неполадки, не готовий до прийому pod'a), або Unknown (контролер вузла не має інформацію про вузол);

– Потужність вузла: описує CPU, пам'ять і максимальне число процесорів під які можуть бути запущені;

– Загальна інформація про вузол: версія ядра, версія Kubernetes (версія kubelet, версія kube-proxy), версія docker, назва ОС;

– На кожній ноді запускаються такі послуги (компоненти):

– Kubelet, який управляє pod'ами, їх контейнерами, образами, розділами;

– Kube-Proxy виконує найпростіше перенаправлення потоків TCP і UDP (round robin) між набором backend і налаштовується в Kubernetes API;

– YAML-файл, який містить в собі налаштування Node Kubernetes.

Система управління Kubernetes розділена на кілька компонентів. В даний момент всі вони запускаються master Node, але незабаром це буде змінено для можливості створення відмов кластеру. Ці компоненти працюють разом, щоб забезпечити подання стану кластера:

– etcd зберігає стан майстра в екземплярі, що забезпечує надійне зберігання конфігураційних даних і своєчасне оповіщення інших компонентів про зміну стану.

– Kubernetes API Server забезпечує роботу API-сервера та призначений для того, щоб бути CRUD сервером з вбудованою бізнес-логікою, реалізованої в окремих компонентах або в плагінах. Він обробляє REST операції, перевіряючи їх і оновлюючи відповідні об'єкти в etcd.

– Scheduler прив'язує pod'и, які не запустились до Node через виклик/binding API.

– Kubernetes Controller Manager Server керує Node Kubernetes та відповідає за відмовостійкість. Якщо одна Node Kubernetes вийшла з ладу, то контейнери з неї автоматично будуть перенесені на інші Node Kubernetes.

– ReplicationController механізм, який визначає, скільки повинно бути запущено оболонок pod або контейнерів. Контейнери розміщуються на серверах. У разі, якщо кількість контейнерів відхиляється від встановленого значення, то проводяться роботи по запуску або деактивації вузлів. У пакеті Kubernetes розглядаються стани вузлів, а не процеси. Коли визначається оболонка (pod), то Kubernetes намагається зробити так, щоб вона завжди працювала. Через це, якщо контейнер буде знищений, то буде зроблена спроба запустити новий контейнер. Якщо контролер реплікації визначає, що повинно бути три репліки, то Kubernetes намагатиметься завжди запустити саме таку їх кількість, запускаючи і зупиняючи контейнери в міру їх необхідності.

Вся робота з контейнеризованими додатками всередині Kubernetes описується декількома абстракціями (або сутностями), дані про яких зберігаються в etcd.

Абстракція Deployment описує для Kubernetes кінцевий стан програмного додатка: з яких образів запускати контейнери застосування, а також, яка кількість цих контейнерів має бути в підсумку. Після створення об'єкта Deployment всередині Kubernetes Deployment-контролер призводить додаток до описаного стану, запускаючи нові або видаляючи зайві контейнери. Ця абстракція забезпечує також відновлення працездатності застосування в разі, якщо вузол, на якому розташовувалися контейнери застосування, раптово вийшов з ладу: потрібну кількість контейнерів, описану в Deployment, буде знову запущено на вільних вузлах кластера.

Важливо відзначити, що в процесі роботи з Kubernetes керування не відбувається безпосередньо контейнерами застосування. Замість цього відбувається керування абстракцією типу Pod. Pod визначає колекцію контейнерів пов'язаних один з одним і розгорнутих на одному вузлі, наприклад, контейнер з базою даних і веб-сервером. В оболонці pod можна вказати кілька контейнерів, які будуть розгорнуті на одному і тому ж хості Docker. Перевага в цьому випадку полягає в

тому, що контейнери, що знаходяться в одній оболонці pod, можуть разом користуватися одними і тими ж ресурсами, такими як томи зберігання даних, і тим же самим простором імен мережі і адресою IP. Ресурси існують протягом усього часу, поки існує оболонка pod, оскільки в контейнерах немає механізму постійного зберігання даних.

Абстракція Service визначає логічну групу з кількох Pod і те, як до них звертатися. При організації роботи додатка всередині Kubernetes мається на увазі, що будь-який Pod може бути видалений в будь-який момент часу: крах фізичного сервера, динамічне зменшення розмірів кластера, оновлення застосування і т.д. Кожен новий Pod матиме IP-адресу. Це призводить до проблеми: якщо який-небудь Pod (наприклад, база даних), що надає сервіс іншому Pod (наприклад, backend-сервер) буде перезапущений, то backend-серверу треба якось дізнатися про цю зміну. Це завдання вирішується за допомогою абстракції Service (сервіс).

Абстракція Volume (диск) використовується для надання контейнерам дискових ресурсів з будь-якого типу системи зберігання даних. Диски в Kubernetes мають явно заданий час життя, рівно стільки ж, скільки живе і використовує їх Pod. Таким чином, диск запросто може пережити перезапуск будь-якого контейнера всередині Pod. Звичайно, якщо Pod завершує роботу і пов'язаний з ним диск також завершую роботу. Kubernetes підтримує велику кількість різних типів дисків, які Pod може використовувати в будь-якій кількості одночасно: emptyDir, hostPath, gcePersistentDisk, awsElasticBlockStore, nfs, iscsi та інші. Диск для контейнера - це директорія, що містить дані, які доступні всім контейнерам всередині Pod.

Kubernetes підтримує можливість організації безлічі віртуальних кластерів, використовуючи при цьому одні і ті ж сервери, що входять в кластер. Ці віртуальні кластери називаються Namespaces (Простір імен). Простір імен необхідний в середовищах, що використовуються декількома користувачами, поділюваних між різними командами або проектами. Для кластерів, які використовуються менш ніж 10-ю користувачами, думати про використання просторів імен не потрібно. Простір імен визначає область видимості імен (імена ресурсів повинні бути унікальні в

межах одного простору імен) та є способом розділити ресурси кластера між декількома користувачами.

Ярлики – це пари ключ/значення, які прикріплені до об'єктів, наприклад, Pod-ами. Вони призначені для установки особливих властивостей об'єктів, які є значущими та актуальними для користувачів, а також для організації та вибору підмножини об'єктів. Ярлики можуть бути приєднані до об'єктів на момент створення і в подальшому додані і змінені в будь-який час. Кожен об'єкт може мати набір ярликів. Кожен ключ має бути унікальним для даного об'єкта.

5.2 Алгоритм роботи платформи Kybernetes та взаємозв’язок її компонентів

На рисунку 5.2 схематично зображено функціонування платформи Kybernetes. На керуючих вузлах кластера master запускається API-сервер (kube-apiserver), планувальник (kube-scheduler), менеджер контролерів (kube-controller-manager) і сховище etcd. На робочих вузлах кластера Node - запускаються ще два додаткових компонента, а саме kube-proxy/Агенти приймають зі спеціального API сервера дані PodSpec (файл або HTTP) і гарантують працездатність вказаних у ньому об'єктів. Proxy забезпечує перенаправлення потоків між Pod. Майстер кластера містить спеціальні компоненти - kube-controller-manager (менеджер сервісів) і kube-scheduler (планувальник), kube-apiserver, etcd і flannel. Доступ до API керування, крім програмного способу, можна отримати через консольну утиліту kubectl і веб-інтерфейс. З їх допомогою можна переглядати поточну конфігурацію, управляти ресурсами, створювати і розгортати контейнери.

Для створення подів використовується команда kubectl run з певними опціями та параметрами. Нижче наведений алгоритм роботи платформи Kybernetes для виконання команди kubectl run щодо створення подів у середовищі.

а) В першу чергу команда kubectl run виконує валідацію на стороні клієнта. В результаті валідації неробочі запити (наприклад, створення ресурсу, який не підтримується, або використання образу з неправильно вказаною назвою) швидко перервуться і не будуть відправлені в kube-apiserver. Так поліпшується продуктивність системи - завдяки зниженню непотрібного навантаження.

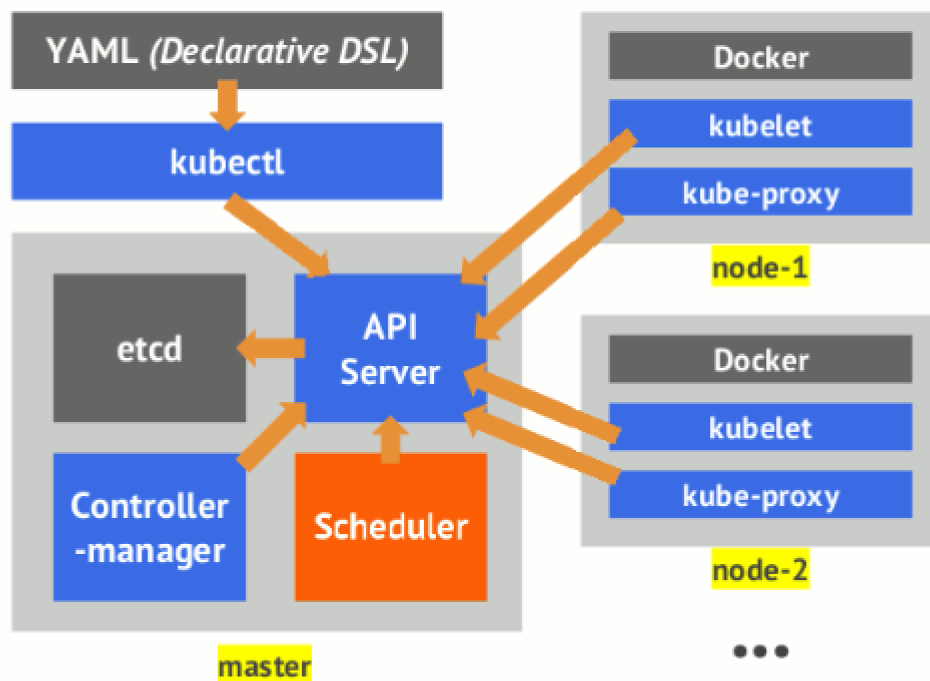


Рисунок 5.2 – Схематичне зображення функціонування компонентів платформи Kybernetes

б) Після валідації kubectl починає складати HTTP-запит, який буде відправлений kube-apiserver. Отримання доступу до стану або зміну його в системі Kubernetes проходить через сервер API, який в свою чергу спілкується з etcd. Щоб скласти HTTP-запит, kubectl використовує так звані генератори (generators) - абстракцію, що реалізує серіалізацію. В kubectl run допускається вказівка безлічі типів ресурсів, не тільки Deployments. Компонет kubectl обчислює тип ресурсу, якщо ім'я генератора не було спеціально зазначено через прапор --generator. Наприклад, ресурси, у яких є --restart-policy = Always, розглядаються як Deployments, а ресурси з --restart-policy = Never - як поди. Розібравшись, що ми хочемо створити Deployment, kubectl скористається генератором DeploymentV1Beta1 для створення runtime-об'єкта з наданих параметрів. Runtime object - це узагальнюючий термін для ресурсу.

в) Після того, як kubectl згенерував runtime-об'єкт, він починає шукати відповідну йому групу API і версію. Kubernetes використовує версійний API, який класифікується за групами API (API groups). Група API призначена для віднесення в

одну категорію схожих ресурсів, щоб з ними було легше взаємодіяти. Крім того, вона є гарною альтернативою єдиному монолітному API. Група API для Deployment називається apps і її остання версія - v1beta2. Після класифікації kubectl збирає клієнта потрібної версії - в ньому враховується різна REST-семантика для ресурсу. Цей етап виявлення називається «Переговори про версії» (version negotiation), і включає в себе сканування вмісту / apis на віддаленому API для отримання всіх можливих груп API. Оскільки kube-apiserver видає структурний документ у форматі OpenAPI по шляху / apis, клієнтам легко виконувати виявлення. Фінальний крок – це відправка HTTP-запиту. Коли вона зроблена і отримана успішна відповідь, kubectl відобразить успішне повідомлення

г) Для успішного відправлення HTTP-запиту kubectl необхідно пройти аутентифікацію. Облікові дані користувача майже завжди зберігаються в файлі kubeconfig, що зберігається на диску, проте він може перебувати в різних місцях. Для його пошуку kubectl робить наступні дії:

1) якщо вказано прапор --kubeconfig - використовує його;
2) якщо визначена змінна оточення $ KUBECONFIG - використовує її;
3) в іншому випадку перевіряє домашній каталог на кшталт ~ / .kube і використовує перший знайдений файл.

Після парсингу файлу визначаються поточний контекст, поточний кластер і аутентифікаційні відомості для поточного користувача. Якщо користувач задав спеціальні значення через прапори (такі, як --username), пріоритет віддається їм і вони переписують значення, зазначені в kubeconfig. Коли інформація отримана, kubectl встановлює конфігурацію клієнта, роблячи її відповідною потребам HTTP-запиту:

1) сертифікати x509 відправляються через tls.TLSConfig;
2) токени клієнта відправляються в HTTP-заголовку Authorization;
3) користувач і пароль відправляються через базову аутентифікацію HTTP;

4) процес аутентифікації через OpenID попередньо здійснюється користувачем вручну, в результаті чого з'являється токен, який відправляється аналогічно відповідним пунктом вище.

Основний інтерфейс, який використовується клієнтами і системними компонентами для збереження і отримання стану кластера, це kube-apiserver. Для виконання цієї функції (збереження і отримання стану кластера) необхідно верифікувати запитуючу сторону, переконавшись, що вона відповідає тому, за кого себе видає. Коли сервер вперше запускається, він перевіряє всі надані користувачем консольні прапори і збирає список відповідних аутентифікатор (authenticators). Якщо був переданий --client-ca-file, буде додано аутентифікатор x509; якщо вказано --token-auth-file - до списку додасться аутентифікатор токенов. Кожен раз при отриманні запиту він проходить через ланцюжок аутентифікаторів, поки один з них не спрацює успішно:

1) обробник x509 верифікує, що HTTP-запит зашифровано за TLS-ключем, підписаним засвідченим кореневим сертифікатом;

2) обробник токенов верифікує, що наданий токен (визначений в HTTP-заголовку Authorization) існує в файлі на диску, зазначеному директивою --token-auth-file;

3) обробник basicauth схожим чином переконається, що облікові дані для базової аутентифікації в HTTP-запиті відповідають локальним даним.

Якщо жоден з аутентифікаторів не пройде перевірку успішно, запит не спрацює і поверне агрегированную помилку. Якщо аутентифікація пройшла успішно, заголовок Authorization забирається із запиту і відомості про користувача додаються в його контекст. Це дає доступ до встановленої ідентичності користувача на наступних етапах (таких, як авторизація та admission controllers).

д) Далі kube-apiserver повинен авторизувати користувача, оскільки ідентичність і право доступу - це не одне і те ж. Спосіб, яким kube-apiserver виконує авторизацію, дуже схожий з аутентифікацією: зі значень прапорів він збирає ланцюжок авторизаторів (authorizers), які будуть використовуватися для кожного вхідного запиту. Якщо всі авторизатори забороняють запит, він завершиться з

відповіддю Forbidden і зупиниться на цьому. Якщо хоч один авторизатор схвалить запит, він пройде далі.

Приклади авторизаторів, що входять до складу релізу Kubernetes v1.8:

1) webhook, взаємодіє з HTTP (S) - сервісом поза кластера K8s;
2) ABAC, який реалізує політики з статичного файлу;
3) RBAC, який реалізує ролі RBAC (Role-based access control), додані адміністратором як ресурси Kubernetes;
4) Node, перевіряючий, що клієнти вузлів кластера - наприклад, kubelet - можуть отримувати доступ тільки до ресурсів, що знаходяться на них самих.

е) Після авторизації і аутентифікації відбувається контроль допуску, який здійснюється компонентом admission controllers. Він перевіряє запит на відповідність, охоплюючи ширший діапазон правил в кластері. Вони є останнім оплотом контролю перед тим, як об'єкт передається в etcd, і мають на меті переконатися, що дія не призведе до несподіваних або негативних наслідків.

Принцип, за яким працюють admission controllers, схожий з аутентифікацією і авторизацією, але має одну відмінність: для admission controllers досить єдиної відмови в ланцюжку контролерів, щоб перервати цей ланцюжок і визнати запит невдалим.

В архітектурі admission controllers реалізована орієнтація на сприяння розширюваності. Кожен контролер зберігається як плагін в директорії plugin / pkg / admission і створюється для реалізації потреб маленького інтерфейсу. Кожен з них компілюється в головний бінарний файл Kubernetes. Зазвичай admission controllers розбиті за категоріями управління ресурсами, безпеки, налаштувань за замовчуванням і еталонної консистентності. Ось деякі приклади контролерів, які займаються управлінням ресурсів:

1) InitialResources встановлює ліміти за замовчуванням для ресурсів контейнера, грунтуючись на попередньому використанні;
2) LimitRanger встановлює значення за замовчуванням для запитів і лімітів контейнера, гарантує верхню межу для певних ресурсів (512 Мб пам'яті за замовчуванням, але не більше 2 Гб);

3) ResourceQuota вважає кількість об'єктів (подів, rc, балансувальник навантаження сервісів) і загальні споживані ресурси (процесор, пам'ять, диск) в просторі імен і запобігає їх перевищення.

ж) Наступним кроком kube-apiserver десеріалізует HTTP-запит, створює runtime-об'єкти з нього (зворотній процес того, що роблять генератори kubectl) і зберігає їх сховище даних. Алгоритм обробки запитів, коли бінарний файл запускається вперше, наступний:

1) Коли бінарник kube-apiserver запущений, він створює ланцюжок server chain, що робить можливою агрегацію Kubernetes apiserver. Це і є основа для підтримки безлічі apiservers.

2) Коли це відбувається, створюється загальний (generic) apiserver, який виступає в ролі реалізації за замовчуванням.

3) Згенерована OpenAPI-схема наповнює конфігурацію apiserver.

4) Потім kube-apiserver послідовно проходить по всіх групах API, зазначеним в схемі, і налаштовує для кожного з них постачальника сховища (storage provider), який виступає в ролі загальної (generic) абстракції сховища. З ним kube-apiserver взаємодіє, коли звертається до стану ресурсу або змінює його.

5) Для кожної групи API послідовно перебираються всі версії групи і встановлюються REST-відповідності кожному HTTP-маршруту. Це дозволяє kube-apiserver знаходити відповідності запитам і делегувати логіку знайденому результату.

6) Реєструється POST-обробник, який потім делегується оброблювачу для створення ресурсу.

До цього моменту kube-apiserver знає, які існують маршрути і має внутрішній mapping, який вказує на те, які обробники і постачальники сховища повинні бути викликані при відповідно запиту. Припустимо, в нього потрапив HTTP-запит, алгоритм обробки запиту буде наступним:

1) Якщо ланцюжок обробників може знайти відповідність запиту шаблоном (тобто зареєстрованим маршрутами), то буде викликаний потрібний

обробник, зареєстрований для цього маршруту. В іншому випадку викликається обробник, заснований на шляхах (те ж саме відбувається при зверненні до / apis). Якщо зареєстрованих обробників для цього шляху немає, викликається обробник not found, який повертає 404.

2) Оброблювач декодує HTTP-запит і виконує базову валідацію, таку як перевірка відповідності наданих JSON-даних з очікуваннями для ресурсу з API потрібної версії.

3) Відбувається аудит і фінальний допуск.

4) Ресурс зберігається в etcd шляхом делегування постачальнику сховища. Зазвичай ключ для etcd представляється у вигляді <namespace> / <name>.

5) Будь-які помилки при створенні перехоплюються і, нарешті, постачальник сховища виконує виклик get, перевіряючи, що об'єкт був дійсно створений. Потім він викликає все обробники, призначені на момент після створення (post-create), і декоратори, якщо потрібна додаткова фіналізація.

6) Створюється HTTP-запит і відправляється назад.

Успішним результатом всіх наведених вище дій є те, що ресурс Deployment тепер існує в etcd.

з) Коли об'єкт збережений в сховище даних, він не є повністю видимим apiserver і не потрапляє в планувальник, поки не відпрацює набір ініціалізаторів (intializers). Ініціалізатор - це контролер, асоційований з типом ресурсу і виконуючий логіку на ресурсі до того, як він стає доступним для apiserver. Якщо у типу ресурсу немає зареєстрованих ініціалізаторів, цей крок пропускається і ресурси видно миттєво. Ініціалізатори дозволяють виконувати загальні операції «початкового завантаження» (bootstrap). Прикладами таких оперцій можуть бути такі:

1) Вставка sidecar-контейнера в под з відкритим 80 портом або з конкретною анотацією (annotation).

2) Вставка тома з тестовими сертифікатами в усі поди певного простору імен;

3) Запобігання створення секрету з довжиною менше 20 символів (наприклад, для пароля).

Приклад ініціалізатора, який буде запускатися при кожному випадку створення пода, зображений на рисунку 5.3.

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: InitializerConfiguration
metadata:
  name: custom-pod-initializer
initializers:
  - name: podimage.example.com
    rules:
      - apiGroups:
          - ""
        apiVersions:
          - v1
        resources:
          - pods
```

Рисунок 5.3 – Приклад створення ініціаліатору

Об'єкти initializerConfiguration дозволяють визначати, які ініціалізатори повинні запускатися для певних типів ресурсів. Після створення цього конфігураційного налаштування в поле очікування (metadata.initializers.pending) кожного пода буде додано назву ыныцыалызатора (в наведеному прикладі - custom-pod-initializer). Контролер ініціалізатора буде вже розгорнуто і почне регулярно сканувати кластер на нові поди. Коли ініціалізатор виявить под зі своєю назвою в поле очікування, він виконає свої дії. Після завершення роботи він видалить свою назву зі списку очікування. Тільки ініціалізатори, назви яких є першими в списку, можуть управляти ресурсами. Коли всі ініціалізатори відпрацювали і список очікування порожній, об'єкт буде вважатись ініціалізованим. Для того, щоб контролер з робочих просторів міг обробляти ресурси, коли kube-apiserver ще не зробив їх видимими, у kube-apiserver є спеціальний параметр запиту Include Uninitialized, що дозволяє повертати всі об'єкти, в тому числі і неініціалізовані.

Дізнавшись, що потрібних записів не існує, контролер починає процес масштабування, щоб прийти до очікуваного стану. Цей процес здійснюється за

допомогою викочування (наприклад, створення) ресурсу ReplicaSet, призначаючи йому label selector і привласнюючи першу ревізію. PodSpec та інші метадані для ReplicaSet копіюються з маніфесту Deployment. Іноді після цього може знадобитися також оновити запис Deployment (наприклад, якщо встановлено progress deadline, тобто визначено поле специфікації .spec.progressDeadlineSeconds). Після цього статус оновлюється і починається цикл звірки, що порівнює Deployment з бажаним. Оскільки контролер знає тільки про створення ReplicaSet, етап звірки триває наступним контролером, відповідальним за ReplicaSet.

и) Контролер ReplicaSets стежить за життєвим циклом ReplicaSet і залежних ресурсів (подів). Як і у більшості інших контролерів, це відбувається завдяки обробникам тригерів певних подій. Коли створюється ReplicaSet (в результаті діяльності контролера Deployments), контролер ReplicaSets інспектує стан нового ReplicaSet і розуміє, що є різниця між тим, що існує, і тим, що потрібно. Тому він коригує стан, викочуючи поди, які будуть належати ReplicaSet. Операції створення для подів теж виконуються пачками, починаючи з SlowStartInitialBatchSize і збільшуючи це значення вдвічі при кожній успішній ітерації операції «повільного старту». Такий підхід покликаний знизити ризик навантаження kube-apiserver непотрібними HTTP-запитами в разі частих помилок завантаження подів (наприклад, через квоти на ресурси). Якщо результат буде безуспішним, то це відбудеться з мінімальним впливом на інші компоненти системи.

Kubernetes реалізує створення ієрархії об'єктів через посилання на власника, Owner References (поле в дочірньому ресурсі, що посилається на ID батька). Це не тільки гарантує, що збирач сміття знайде всі дочірні ресурси при видаленні ресурсу, керованого контролером (каскадне видалення), а й забезпечує ефективний спосіб для батьківських ресурсів не боротися за своїх дітей (ситуація, при якій два потенційних батька думають, що володіють одним і тим самим дочірнім ресурсом). Іноді в системі з'являються «осиротілі» ресурси - зазвичай це відбувається через те, що:

4) видаляється батьківський ресурс, але не його дочірні;
5) політики збору сміття забороняють видалення дочірніх ресурсів.

Коли така ситуація відбувається, контролери перевіряють, щоб «осиротілий» ресурс був прийнятий новим батьком. Безліч батьків можуть претендувати на «осиротілий» ресурс, але тільки один з них доб'ється успіху, а всі інші отримають помилку валідації.

Ще одна перевага архітектури з Owner References - вона є stateful: якщо будь-який контролер повинен перезавантажитися, це не торкнеться інших частини системи, оскільки топологія ресурсів не залежить від контролера. Орієнтація на ізоляцію проникла і в архітектуру самих контролерів: вони не повинні працювати з ресурсами, якими не володіють явно.

к) Для функціонування таких контроллеров, як авторизатор RBAC або контроллер Deployments, потрібно отримати стан кластера за допомогою інформаторів (informers). Інформатор - це паттерн, що дозволяє контролерам підписатися на події з хранилища та отримати список ресурсів, в яких вони зацікавлені. Крім надання зручної роботи абстракції, він також реалізує безліч базових механізмів, таких як кешування, що зменшує кількість підключень до kube-apiserver і необхідність повторної серілізації на стороні сервера та контролера. Крім того, такий підхід дозволяє контролерам взаємодіяти з обліком потокової безпеки (thread safety). Потокова безпека - це концепція програмування, що застосовується до багатопоточних програм. Код потоковобезпечний, якщо він функціонує нормально при його використанні з декількох потоків одночасно. Зокрема, він повинен забезпечувати правильний доступ декількох потоків до розділених даних.

л) Після того, як відпрацювали всі контролери, в результаті є Deployment, ReplicaSet і певна кількість подів (заданих при виконанні команди kubectl run), які зберігаються в etcd і доступні в kube-apiserver. Однак, поди знаходяться в стані Pending, тому що ще не були заплановані / призначені на вузол. Планувальник (scheduler) - останній контролер, який це і робить. Він запускається як самостійний компонент control plane і працює аналогічно іншим контролерам: стежить за подіями і намагається привести стан до потрібного. Він вибирає поди з порожім полем NodeName в PodSpec і намагається знайти підходящий вузол, на який можна

призначити под. Для пошуку відповідного вузла використовується спеціальний алгоритм планування. За замовчуванням він працює так:

1) Коли планувальник запускається, реєструється ланцюжок предикатів за замовчуванням. Ці предикати є функціями, які при своєму виклику відфільтровують вузли, які підходять для розміщення пода. Наприклад, якщо в PodSpec задані явні вимоги по ресурсах CPU або RAM, а вузол не задовольняє таким вимогам через нестачу ресурсів, цей вузол не буде обраний для пода (ресурсомісткість вузла рахується як загальна ємність за вирахуванням суми ресурсів контейнерів, що запитуються, запущених в даний момент).

2) Коли відповідні вузли були обрані, запускається набір функцій пріорітетізації, щоб ранжувати їх, вибравши найбільш пріоритетні. Наприклад, для кращого поширення робочого навантаження по системі пріоритет віддається вузлам, у яких менше запитів на отримання ресурсів, оскільки це служить індикатором наявності меншого робочого навантаження. У міру запуску цих функцій кожному вузлу привласнюється числовий рейтинг. Вузол з найвищим рейтингом вибирається для планування (призначення).

Предикати і функції пріорітетізації розширюються і можуть визначатися прапором --policy-config-file. Це надає певну гнучкість. Адміністратори також можуть запускати свої планувальники для окремих Deployments. Якщо в PodSpec міститься schedulerName, Kubernetes передасть планування цього пода будь-якому планувальнику, зареєстрованому під відповідною назвою.

Коли алгоритмом визначено вузол, планувальник створює Binding-об'єкт, значення Name і UID якого відповідають поду, а поле ObjectReference містить назву обраного вузла. Він відправляється в apiserver через POST-запит. Коли kube-apiserver отримає Binding-об'єкт, registry десеріалізует його і оновить такі поля в об'єкті пода: встановить йому NodeName з ObjectReference, додасть відповідні анотації (annotations), встановить статус PodScheduled значенням True.

м)Коли планувальник призначив поду вузол, на цьому поді починає свою роботу kubelet. Це агент, який запускається на кожному вузлі в кластері Kubernetes і

є відповідальним за забезпечення життєвого циклу подів. Таким чином, він обслуговує всю логіку інтерпретації абстракції пода в контейнери. Також він обробляє всю логіку, пов'язану з монтуванням томів, логування контейнера, збором сміття і багатьма іншими важливими речами. Kubelet опитує поди в kube-apiserver з певною частотою часу, фільтруючи ті з них, які мають значення NodeName, що відповідають назві вузла, де запущений kubelet. Отримавши список подів, він порівнює його зі своїм внутрішнім кешем, тим самим виявляє нові поповнення та починає синхронізацію стану, якщо відмінності існують. Процес синхронізації має наступні етапи:

1) При створенні пода, kubelet реєструє початкову метрику, яка використовується в Prometheus для трекінгу затримок у подів.

2) Створюється об'єкт PodStatus, що представляє стан поточної фази (phase) пода. Фаза пода - це високорівневе позначення положення поду в його життєвому циклі:

– Pending (очікування): API Server створив ресурс пода і зберіг його в etcd, але под ще не був запланований, а образи його контейнерів - не отримані з реєстру;

– Running (функціонує): под був призначений вузлу і всі контейнери створені kubelet'ом;

– Succeeded (успішно завершений): функціонування всіх контейнерів пода успішно завершено і вони не будуть перезапускатися;

– Failed (завершено з помилкою): функціонування всіх контейнерів пода припинено і як мінімум один з контейнерів завершився зі збоєм;

– Unknown (невідомо): API Server не зміг опитати статус пода, зазвичай через помилки у взаємодії з kubelet.

3) Після того, як PodStatus створений, він буде відправлений менеджеру станів пода, який асинхронно оновлює запис в etcd через apiserver.

4) Далі запускаються набір обробники допуску, що перевіряють, що у пода коректні права безпеки. Зокрема, застосовуються профілі AppArmor і

NO_NEW_PRIVS. Поди, які отримали відмову на цьому етапі, залишаться в стані Pending на невизначений час.

5) Якщо вказано runtime-прапор cgroups-per-qos, kubelet створить cgroups для пода і застосує параметри за ресурсами.

6) Створюються директорії з даними подів. До них відносяться каталоги пода (зазвичай / var / run / kubelet / pods / <podID>), його томів (<podDir> / volumes) і плагінів (<podDir> / plugins).

7) Менеджер томів підключить всі необхідні томи, що визначені в Spec.Volumes, і дочекається їх. Деяким подам може знадобитися більше часу в залежності від типу томів, що вмонтовуються (наприклад, хмарні або NFS-томи).

8) З apiserver будуть отримані всі секрети, зазначені в Spec.ImagePullSecrets, для можливості їх подальшої вставки в контейнер.

н) Основна підготовча частина завершена і контейнер готовий до запуску. Програмне забезпечення, яке здійснює цей запуск, називається середовищем виконання контейнерів (Container Runtime) - наприклад, docker або rkt. Прагнення стати більш розширюваним призвело kubelet до того, що з версії 1.5.0 він використовує концепцію під назвою CRI (Container Runtime Interface) для взаємодії з конкретними виконуваними середовищами контейнерів. CRI пропонує абстракцію між kubelet і конкретною реалізацією виконуваної середовища. Взаємодія відбувається через Protocol Buffers і gRPC API. Такий підхід дозволяє додавати нові виконувані середовища з мінімальними витратами, оскільки основний код Kubernetes міняти не потрібно.

Коли под запускається вперше, kubelet виконує віддалений виклик процедури (RPC) RunPodSandbox. «Sandbox» - це термін CRI, що описує набір контейнерів, що на мові Kubernetes означає под. Цей термін є досить широким, щоб не втрачати своє значення для інших виконуваних середовищ, які в дійсності можуть використовувати не контейнери, а, наприклад, віртуальні машини.

В якості виконуваного середовища в даній роботі використовується Docker. У цьому виконуваному середовищі створення пісочниці включає в себе створення

«припиненого» (pause) контейнера. Pause-контейнер виконує роль батька для всіх інших контейнерів в поді, розміщуючи у себе безліч ресурсів рівня пода, які будуть використовуватися навантаженими контейнерами. Ці «ресурси» належать простору імен Linux (IPC, мережа, PID). У ядра Linux є концепція просторів імен (namespaces), що дозволяють хостовій операційній системі забирати певний набір ресурсів (наприклад, CPU або пам'яті) і призначати його на процеси так, немов тільки вони споживають цей набір ресурсів. Важливі ще й cgroups, оскільки вони є способом, яким Linux управляє розподілом ресурсів. Docker використовує обидві ці можливості ядра для розміщення процесу, для якого гарантовані ресурси і забезпечена ізоляція.

Pause-контейнер надає спосіб розміщення всіх цих просторів імен і дозволяє дочірнім контейнерам спільно їх використовувати. Будучи частиною єдиного мережевого простору імен, контейнери одного пода можуть звертатися один до одного через localhost. Друга роль pause-контейнера пов'язана з тим, як працює простор імен PID. У просторі імен такого типу процеси утворюють ієрархічне дерево, і верхній процес «init» бере на себе відповідальність за «витяг» мертвих процесів, тобто видалення їх записів з таблиці процесів операційної системи. Після того, як pause-контейнер був створений, для нього робиться контрольна точка (checkpoint) на диску і він запускається.

о) Kubelet налаштовує мережу для пода, передаючи це завдання плагіну CNI - Container Network Interface і працює за принципом, схожим на Container Runtime Interface. CNI - абстракція, що дозволяє різним мережевим провайдерам використовувати різні мережеві реалізації для контейнерів. Модулі реєструються, і kubelet взаємодіє з ними через дані в JSON (конфігураційні файли розташовані в /etc/cni/net.d), що відправляються відповідного виконуваного файлу CNI (знаходиться в / opt / cni / bin) через stdin. Приклад JSON-файлу з мережевими конфігураціями зображений на рисунку 5.4. Алгоритм роботи плагіну Container Network Interface наведений нижче:

1) Насамперед плагін налаштовує локальний Linux-міст в кореневому мережевому просторі назв для обслуговування всіх контейнерів хоста.

2) Потім додається інтерфейс (один кінець veth-пари) в мережевий простір імен pause-контейнера, а інший його кінець підключається до мосту.

3) Потім інтерфейсу pause-контейнера призначається IP і налаштовуються маршрути. Результатом стане отримання подом своєї IP-адреси. Призначення IP делегується IPAM-провайдерам, заданим в JSON-конфігурації. Модулі IPAM схожі на основні мережеві плагіни: вони викликаються через виконуваний файл і мають стандартизований інтерфейс. Кожен повинен визначити IP / підмережу інтерфейсу контейнера, шлюз, маршрути і повернути цю інформацію основному плагіну. Самий загальний IPAM-плагін називається host-local і призначає IP-адреси з зумовленого набору діапазону адрес. Він зберігає стан локально на файлової системи хоста, гарантуючи таким чином унікальність IP-адрес на одному хості.

4) Для роботи DNS kubelet визначить внутрішню IP-адресу DNS-сервера плагіна CNI, який подбає про правильне налаштування файлу resolv.conf в контейнері. Коли цей процес завершено, плагін поверне в kubelet дані в форматі JSON, повідомляючи про результат операції.

```json
{
    "cniVersion": "0.3.1",
    "name": "bridge",
    "type": "bridge",
    "bridge": "cnio0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "ranges": [
          [{"subnet": "${POD_CIDR}"}]
        ],
        "routes": [{"dst": "0.0.0.0/0"}]
    }
}
```

Рисунок 5.4 – Приклад JSON-файлу з мережевою конфігурацією для контейнера

п) Для того, щоб хости могли взаємодіяти між собою використовується концепція під назвою оверлійна мережа (overlay networking), що пропонує спосіб

динамічної синхронізації маршрутів на безлічі хостів. Одним з популярних провайдерів оверлейної мережі є Flannel. Він забезпечує L3 IPv4-мережу між вузлами кластера. Flannel тільки доставляє трафік між хостами і не контролює, як контейнери приєднані до хосту, оскільки цим займається CNI. Для цього він вибирає підмережу для хоста і реєструє її в etcd. Потім зберігає локальне уявлення маршрутів кластера і інкапсулює пакети в UDP-датаграми, гарантуючи їх доставку до потрібного хоста.

р) Після мережевих налаштувань відбувається безпосередній запуск робочих контейнерів. Коли пісочниця закінчила ініціалізацію і стала активною, kubelet може почати створення контейнерів для неї. Насамперед він запускає init-контейнери, визначені в PodSpec, а потім - основні. Цей процес виглядає наступним чином:

1) Збирається образ для контейнера. Для приватних реєстрів використовуються секрети, зазначені в PodSpec.

2) Через CRI створюється контейнер. Для цього наповнюється структура ContainerConfig, в якій визначаються команда, образ, лейбли, монтування, пристрої, змінні оточення і т.п. за даними батьківського PodSpec і відправляється через protobufs в плагін CRI. Для Docker перед відправкою даних в Daemon API десеріалізуется payload і наповнюються структури конфігов. У процесі також контейнеру додаються кілька лейблів такі, як тип контейнера, шлях до балки, ID пісочниці.

3) Після цього контейнер реєструється у CPU Manager - це нова можливість, що з'явилася у версії 1.8 в статусі alpha і призначає контейнери наборам CPU локального вузла за допомогою методу UpdateContainerResources в CRI. Після цього контейнер запускається.

4) Якщо зареєстровані будь-які хукі після запуску контейнера (post-start), вони запускаються. Хукі можуть бути типу Exec, що виконують конкретну команду всередині контейнера, або HTTP, що виконують HTTP-запит до endpoint контейнера. Якщо хук виконується занадто довго, зависає або завершується з помилкою, контейнер ніколи не перейде в статус Running.

Результатом успішного виконання всіх вище наведених дій буде запуск необхідної кількості контейнерів на одному чи декількох робочих вузлах. Мережа, томи та секрети будуть впроваджені kubelet'ом в контейнери через плагін CRI.

5.3 Програмне забезпечення Docker. Архітектура платформи та її компоненти управління

Docker - програмне забезпечення для автоматизації розгортання і управління додатками в середовищі віртуалізації на рівні операційної системи, наприклад LXC [4]. Дозволяє створити додаток з його оточенням і залежностями в контейнер, який може бути перенесений на будь-яку Linux-систему з підтримкою cgroups в ядрі, а також надає середовище з управління контейнерами.

Docker дає можливість створити і запустити додаток в слабо ізольованому оточенні, званому контейнер. Ізоляція і безпеку рішення дозволяють запускати безліч контейнерів одночасно на потрібному хості. Завдяки легковагій природі контейнерів, без додаткових витрат на гіпервізор можна запускати більше контейнерів на доступному залізі, ніж використання віртуальних машин.

Docker надає утиліти і платформу для управління життєвим циклом контейнерів:

– Инкапсулювати додатки (і підтримувані компоненти) в контейнери Docker;

– Поширювати і доставляти ці контейнери командам програмістів для подальшої розробки та тестування;

– Розгортати ці додатки в продуктивному оточенні, що б це не було, локальний датацентр або хмара.

Основні переваги при використанні Docker контейнерів полягають у наступному:

– Заснована на контейнерах docker платформа дозволять легко перенести корисне навантаження. Docker контейнери можуть працювати як на реальній локальній машині або машині в датацентрі, так і на віртуальній машині в хмарі.

– Портіруемість і легкість dockera дозволяє легко динамічно управляти навантаженням. Можна використовувати docker, щоб розгорнути або погасити ваш

додаток або сервіси. Швидкість docker дозволяє робити це майже в режимі реального часу.

– Docker-контейнери легковагі. Можна створити і запустити Docker-контейнер за секунди, на відміну від віртуальних машин, кожен раз запускають повноцінну віртуальну ОС.

– Docker дозволяє легко розділяти функціональність програми в окремі контейнери. Наприклад, можна запускати базу даних Postgres в одному контейнері, сховище Redis в іншому, в той час як додаток Node.js знаходиться в третьому. Це дозволить легко масштабувати і оновлювати компоненти програми незалежно один від одного.

Платформа Docker використовує клієнт-серверну архітектуру, що зображена на рисунку 5.5. Docker-клієнт спілкується з демоном Docker, який виконує такі дії: створення, запуск, розподіл контейнерів. І клієнт і сервер можуть працювати на одній системі, можна підключити клієнт до віддаленого демона docker. Клієнт і сервер спілкуються через сокет або через RESTful API:

– Docker-демон запускається на хості. Користувач не взаємодіє з сервером безпосередньо, а використовує для цього Docker-клієнт.

– Docker-клієнт, програма docker - головний інтерфейс до Docker. Вона отримує команди від користувача і взаємодіє з docker-демоном.

Основними компонентами функціонування платформи Docker є:

– образи (images);

– реєстр (registries);

– контейнери (container).

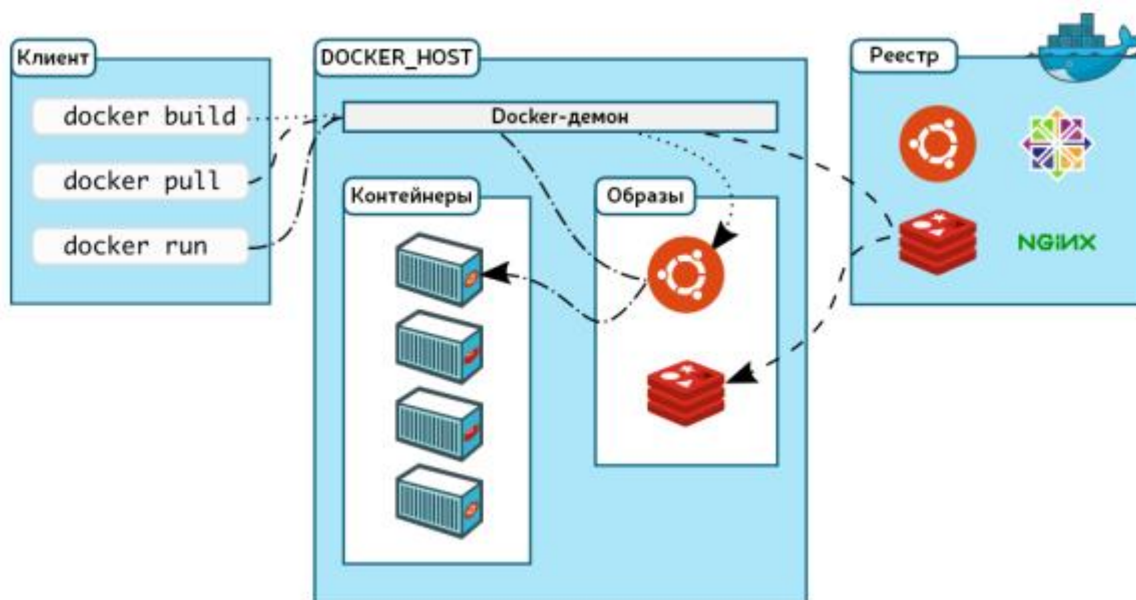


Рисунок 5.5 – Архітектура платформи програмного забезпечення Docker

Docker-образ - це read-only шаблон. Наприклад, образ може містити операційну систему Ubuntu с веб-сервером Nginx і додатком на ній. Образи використовуються для створення контейнерів. Docker дозволяє створювати нові образи і оновлювати існуючі. Кожен образ складається з набору рівнів. Docker використовує UnionFile System для поєднання цих рівнів в один образ. Одна з причин, по якій docker легковагий - це використання таких рівнів. При зміні способу (наприклад, при оновленні програми) створюється новий рівень. Так, без заміни всього образу або його пересборки (як це довелося б зробити з віртуальною машиною) відбувається лише додавання або видалення потрібних рівнів. Це також дозволяє поширювати образи простіше і швидше. В основі кожного образу знаходиться базовий образ. Наприклад, ubuntu, базовий образ Ubuntu, або debian, базовий образ дистрибутива Debian. Docker образи можуть створитися з базових образів, кроки опису для створення цих образів називаються «інструкціями». Кожна інструкція створює новий образ або рівень. Інструкціями будуть наступні дії:

– запуск команди;
– додавання файлу або директорії;
– створення змінної оточення;
– вказівки, що запускати, коли запускається контейнер цього образу.

Ці інструкції зберігаються в файлі Dockerfile. Docker зчитує Dockerfile при складанні образу, виконує ці інструкції і повертає кінцевий образ.

Docker-реєстр зберігає образи. Є публічні і приватні реєстри, з яких можна скачати або завантажити образи. Docker Hub і Docker Cloud - це загальнодоступні реєстри, котрі можуть використовувати будь-хто, а Docker налаштовано за замовчуванням шукати зображення в Docker Hub. Можливо навіть запустити свій власний реєстр. Docker store дозволяє купувати та продавати образи Docker або розповсюджувати їх безкоштовно. Наприклад, можна придбати зображення Docker, що містить додаток або послугу від постачальника програмного забезпечення, і використовувати зображення для розгортання програми у вашій тестовій, постановочної та виробничій середовищах. Є можливість оновити додаток, витягнувши нову версію образу та переміщуючи контейнери.

Контейнери схожі на директорії. У контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для додатка. Контейнер - це незмінний екземпляр зображення. Можливо створювати, запускати, зупиняти, переміщавти або видаляти контейнери за допомогою Docker API або CLI. Є можливість підключення контейнера до однієї або декількох мереж, приєднати для нього пам'ять або навіть створити нове зображення на основі поточного стану. Контейнер визначається його образом, а також будь-якими параметрами конфігурації, які надані йому при створенні або запуску. Коли контейнер видаляється, будь-які зміни в його стані, які не зберігаються в постійному зберіганні, зникають.

5.4 Основні технології платформи Docker

– Простір назв «namespaces»

Docker використовує технологію «namespaces» для організації ізольованих робочих просторів. При запуску контейнера docker створює набір просторів імен для даного контейнера. Це створює ізольований рівень, кожен аспект контейнера запущений в своєму просторі імен і не має доступ до зовнішньої системи. На

поточний момент існує шість просторів імен. Кожне з них може незалежно від інших надавати певного процесу або його нащадку різні ресурси машини. Простори імен бувають:

PID: Це простір імен надає процесу і його нащадкам власне бачення набору процесів в системі. Це можна уявити собі як таблицю відповідності. Коли процес в просторі PID запитує у системи список процесів, то ядро спочатку дивиться в цю таблицю відповідності. Якщо процес існує в цій таблиці, то використовується відповідний ID замість реального ID. Якщо ж процесу немає в таблиці, то ядро вдає, що процесу взагалі не існує. Простір PID задає першому процесу ID 1, таким чином, можемо ізолювати дерево процесів для кожного контейнера.

MNT: Це одне з найважливіших просторів імен. Воно надає процесу окремі таблиці монтування. Це означає, що процес може монтувати і отмонтировать необхідні директорії не зачіпаючи інших просторів імен (в тому числі і основне). І ще один важливий момент: в поєднанні з використанням системного виклику pivot_root дозволити процесу мати свою власну файлову систему. Саме завдяки цій фічи, просто перемикаючи файлові системи, які бачить контейнер, можливо зробити вигляд, що процес запущений на ubuntu, busybox або alpine.

NET: Мережеве простір імен надає процесу можливість використовувати свій власний мережевий стек. Звичайно, тільки процеси з головного мережевого простору імен (ті, які запускаються, коли вмикається комп'ютер) мають доступ до реальних мережних карт. Але можливо створювати віртуальні ізернет пари - віртуально пов'язані мережеві карти в різних просторах імен. Це виглядає як наявність декількох ip стеків на одній машині, які можуть спілкуватися один з одним. Якщо додати трохи магії роутінга, то можна надати контейнерів доступ в реальний світ, незважаючи на ізоляцію в своєму власне ip стеку.

UTS: Простір імен надає процесу можливість мати свої власні імена для хоста і домену. Після настройки простору імен UTS, будь-які зміни імені хоста або домену не торкнуться інші процеси.

IPC: Це простір імен ізолює механізм межпроцессорного взаємодії, таких як черга повідомлень. Для більш глибокого розуміння, загляньте в документацію.

USER: Це простір імен було доданий зовсім недавно і саме настройка цього простору має найбільший вплив на безпеку роботи контейнера. Простір USER виконує мапінг між UID'амі в рамках процесу до UID'амі (і GID'амі) самого хоста. Це дуже корисно. Використовуючи цей простір імен, можна прив'язати ID користувача root в контейнері (наприклад 0) до довільного і непривілейованих UID на реальному хості. Таким чином, надається root доступ до ресурсів усередині контейнера, але, насправді, котрі дають root права в рамках всього хоста. Контейнер може запускати процеси з UID 0 (що означає root користувача) але в ядрі буде відбуватися мапінг цього UID з деяким непривілейованом реальним UID. Більшість систем контейнеризації не мапят UID всередині контейнера на нульовий UID.

– Контрольні групи cgroups

Docker Engine на Linux також покладається на іншу технологію, яка називається контрольними групами (cgroups). Cgroup обмежує застосування до певного набору ресурсів. Контрольні групи дозволяють Docker Engine обмінюватися наявними апаратними ресурсами з контейнерами та, можливо, забезпечити обмеження. Наприклад, можна обмежити пам'ять, доступну для певного контейнера.

– Union file systems

Мережева файлова система або UnionFS [5]- це файлова системи, якf працює шляхом створення шарів, що робить їх дуже легкими і швидкими. Docker Engine використовує UnionFS, щоб забезпечити будівельні блоки для контейнерів. Docker Engine може використовувати кілька варіантів UnionFS, включаючи AUFS, btrfs, vfs та DeviceMapper. UnionFS - допоміжна файлова система для Linux і FreeBSD, яка виробляє каскадно-об'єднане монтування інших файлових систем. Це дозволяє файлам і каталогам ізольованих файлових систем, відомих як гілки, прозоро перекриватися, формуючи єдину пов'язану файлову систему. Каталоги, які мають той же шлях в об'єднаних гілках, буде спільно відображати вміст в об'єднаному каталозі нової віртуальної файлової системи. Docker використовує UnionFS для створення блоків, з яких будується контейнер. Docker може використовувати кілька реалізацій UnionFS, включаючи: AUFS, btrfs, vfs і DeviceMapper.

5.5 Моніторинг Docker-контейнерів

Моніторинг додатків і серверів додатків - важлива частина DevOps. Необхідно постійно моніторити стан програми та серверів, завантаження центрального процесора, споживання пам'яті, дискову утилізацію і т.д. Необхідно отримувати повідомлення, якщо у сервера закінчується доступна пам'ять або додаток перестає відповідати на запити, що дозволить запобігти проблемам. Для моніторингу є ряд безкоштовних і платних інструментів, таких як Amazon CloudWatch, Nagios, New Relic, Prometheus, Zabbix і інші.

Prometheus [6] - популярний CNCF-проект з відкритим вихідних кодом. Prometheus повністю сумісний з Docker і доступний на Docker Hub. Prometheus має центральний компонент, званий Prometheus Server. Його основне завдання - зберігати і моніторити певні об'єкти. Об'єктом може стати що завгодно: Linux-сервер, сервер Apache, один з процесів, сервер бази даних або будь-який інший компонент системи, яку необхідно контролювати. У термінах Prometheus головна служба моніторингу називається сервером Prometheus, а об'єкти моніторингу - цільовими об'єктами. Цільовим об'єктом може бути один сервер, або цільові об'єкти для перевірки кінцевих точок через HTTP, HTTPS, DNS, TCP і ICMP (* Black-Box Exporter), або проста кінцева точка HTTP, яку видає додаток. Через кінцеву точку HTTP сервер Prometheus перевіряє статус додатка. Кожен елемент цільового об'єкта, який необхідно моніторити (статус центрального процесора, пам'ять або будь-який інший елемент), називається метрикою. Таким чином, Prometheus збирає через HTTP метрики цільових об'єктів, зберігає їх локально або віддалено і відображає. Сервер Prometheus зчитує цільові об'єкти з інтервалом, який заданий на збір метрик, і зберігає їх в базі даних часових рядів. Цільові об'єкти і часовий інтервал зчитування метрик задається в конфігураційному файлі. На рисунку 5.6 показана архітектура програмного забезпечення Prometheus.

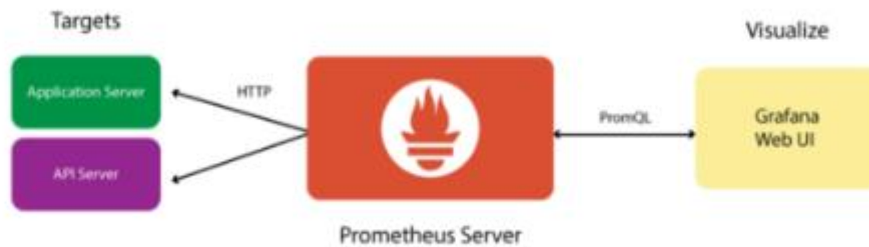


Рисунок 5.6 – Архітектура програмного забезпечення Prometheus

Користувач робить запит у бази даних часових рядів Prometheus інформацію про місце зберігання метрик, використовуючи мову запитів PromQL. Іншими словами, за допомогою PromQL йде запит до сервера Prometheus показати статус конкретного цільового об'єкта в даний момент часу і користувач отримує метрики.

Prometheus надає клієнтські бібліотеки на декількох мовах, які можна використовувати для забезпечення працездатності програми. Але Prometheus - це не тільки моніторинг додатків. Він надає можливість використовувати експортери (exporters) для моніторингу сторонніх систем (таких як сервер Linux, демон MySQL і т.д.). Експортер - частина програмного забезпечення, яке отримує існуючі метрики від сторонньої системи і експортує їх в формат, зрозумілий сервера Prometheus.

Grafana [6] використовується в якості стороннього компонента для візуалізації метрик, що зберігаються в базі даних часових рядів Prometheus. Замість того, щоб писати запити PromQL безпосередньо на сервер Prometheus, використовують дошки графічного інтерфейсу Grafana для запиту метрик з сервера Prometheus і візуалізації їх на панелі моніторингу Grafana.

6 ПРАКТИЧНА ЧАСТИНА

6.1 Обгрунтування системи

В сучасному світі існує безліч додатків та інтернет ресурсів, котрі потребуютm для своїх операції деяку кількість ресурсів. Проте, ця кількість може змінюватися протягом часу, тому що кожен додаток вирішує певні задачі. Для прикладу розглянемо таблицю 6.1 «Залежність ресурсних потреб від типу додатка та його регіонального значення». З таблиці видно, що від типу додатку та його регіональності залежить кількість необхідних йому ресурсів. Наприкла, для довідкової служби міста Києва необхідно надати достатню кількість ресурсів вдень. А ось для соціальної мережі світового рівня вирахувати пікову потребу в ресурсах неможливо тому, що на даний показник впливає безліч факторів, таких як свято в країни, що понизить трафік, або навпаки, запущена акція, тому попит буде більший. Для пікових годин користування додатку необхідно більше ресурсів, а в інші години - менше. Кожен наданий ресурс для обчислення задач додатку потребуэ певноъ оплати, надати додатку максимальні ресурси та сплачувати їх, коли додаток простоює або виконує малу кількість своїх функції, не має сенсу. Отже, це питання необхідно вирішити способом управління ресурсами для застосунків. Тому була розроблена система управління обчислювальними ресурсами застосунків в умовах контейнерної віртуалізації.

Таблиця 6.1 - Залежність ресурсних потреб від типу додатка та його регіонального значення

Тип додатку	Территороіальне розташування клієнтської аудиторії	Часовий проміжок потреби в ресурсах	
		пікова	мінімальна
Довідкова служба міста Києва	місто	8:00-21:00	23:00-5:00
Банк України	країна	7:00-23:00	01:00-5:00
Соціальна мережа	світова	не визначено	не визначено

ПІд час розробки системи управління обчислювальними ресурсами застосунків було розглянуто наступні проблеми:

– Автоматичне розгортання та згортання контейнерів;

– Оптимальне використання обчислювальних машин;

– Алгоритм для обчислення необхідних контейнерів.

Кожен застосунок розміщується на віртуальному або фізичному сервері, та йому виділяється певна кількість обчислювальних потужностей. Для малих додатків оптимальніше застосовувати сервіси, які надають не виділену потужність, а автомасштабовану, тобто застосунку буде надана визначена кількість потужності, якої він потребує. Для потужних застосунків, які мають в своєму розпорядженні окремий фізичний, доцільно раціонально використовувати її потужності.

Розроблювана система управління обчислювальними ресурсами застосунків вирішує задачу оптимального навантаження сервера методом автоматичного додавання та видалення контейнерів. Загальна кількість наданої потужності ділиться на декілька частин (node), в котрих розміщуються контейнери для виконання задач. Це надає змогу використовувати раціонально потужність, та не перевантажувати сервери. Система автоматично розподіляє де розміщувати контейнери так, щоб кожен з них не був перевантажаний, додатково вирішує задачу використання найменшої кількості необхідних серверів серверів. Якщо один з серверів завантажений на 15%, а інший на 50%, то система автоматично перенесе контейнери з менш завантаженого до більш навантаженого сервера та вимкне перший. Це надає можливість в раціональному використанні електроенергії та обчислювальних ресурсів сервера в цілому.

В ході виконання задач магістерської дисертації було розроблено два алгоритми, котрі в залежності від вхідних параметрів підраховують необхідну кількість працюючих контейнерів. Розроблені алгоритми використовують арифметичну та геометричну прогресії при додаванні або виключенні контейнерів. Аналізуючи два види алгоритмів було визначено, що кожен з них добре справляється зі своєю задачею, проте кожен з них більше підходить для визначених типів застосунків. Загальний алгоритм роботи системи наведений у Додатку А.

6.2 Алгоритм розміщення контейнерів

Розроблювана система управління обчислювальними ресурсами розміщується на сервері, який має наступні обчислювальні характеристики:

- SSD- 2048Gb (2 Tb);
- RAM-32Gb;
- Intel Xeon Silver 4110;
- швидкісний інтернет 1Gb/сек.

Надана потужність була розділена на 8 віртуальних серверів, кожен з яких мав:

- SSD- 256Gb (2 Tb) ;
- RAM-4Gb.

Така конфігурація надає змогу більш ефективно масштабувати контейнери на кожному з серверів. Вхідні параметри контейнера:

- займана пам’ять на сервері 4Gb.

Оптимальне навантаження на сервер- OptLS = 80%.

Навантаження одного контейнера на сервер вираховується формулою 6.1.

$$lk = \frac{100\%}{\frac{Sc}{Sk}}, \tag{6.1}$$

де Sc – пам’ять сервера;

Sk – використана пам’ять контейнера на сервері.

$$lk = \frac{100\%}{\frac{256}{4}} = 1{,}6\%$$

Отже, підставивши відомі дані в формулу 6.1, маємо, що навантаження одного контейнера на сервер дорівнює 1,6%.

Оптимальну кількість розміщуваних контейнерів на одному сервері вираховується формулою 6.2.

$$Npt = \frac{OptLS}{lk}, \quad (6.2)$$

де *OptLS* - оптимальне навантаження на сервер;

lk- навантаження одного контейнера на сервер.

$$Npt = \frac{80\%}{1.6\%} = 50 \text{ контейнерів.}$$

Отже, підставивши відомі дані в формулу 6.2, маємо, що оптимальна кількість розміщуваних контейнерів на одному сервері дорівнює 50 контейнерам.

Введемо додаткову зміну k-кількість розташованих контейнерів на одному сервері. Реальне навантаження на сервер визначається формулою 6.3.

$$RL = \frac{100\%}{\mathrm{k} \cdot \mathrm{lk}}, \quad (6.3)$$

де k – кількість розташованих контейнерів на одному сервері;

lk - навантаження одного контейнера на сервер.

Значення, на яке зменшиться відклик системи розраховується формулою 6.4.

$$St_i = \mathrm{N} \cdot \mathrm{k}, \quad (6.4)$$

де N – кількість розгорнутих контейнерів;

k – навантаження, яке оброблює контейнер.

Навантаження на сервер розраховується формулою 6.5.

$$Ts_i = \mathrm{t_i} - \mathrm{St_i} + \mathrm{Ts_{i-1}}, \quad (6.5)$$

де t_i – навантаження на сервер;

St_i – значення, на яке зменшиться відклик системи.

Додаткова умова для функціонування системи: якщо реальне навантаження на сервер $RL \leq 25\%$, то контейнери будуть розміщені на тому сервері, де найменший показник навантаження . Алгоритм розміщення контейнерів на сервері наведений у Додатку Б.

6.3 Алгоритм керування контейнерами з використанням арифметичної прогресії

Для оптимального застосування алгоритму керування контейнерами з використанням арифметичної прогресії слід використовувати застосунки з послідовним навантаженням, де показник швидкості функціонування системи не є головним чинником. Алгоритм керування контейнерами з використанням арифметичної прогресії наведений в Додатку В та в Додатку Г

Арифметична прогресія має вид $a_n = a_m + (n - m) \cdot d$. Контейнери в данному алгоритмі будуть додаватись в наступній кількості: 1, 2, 3, 4 і так далі поки на сервері буде вистачать пам'яті. Кожен контейнер може обробити певну кількість навантаженя, що дорівнює діапазону значень [1.5; 2.1].

Програмний код для алгоритму з використанням арифметичної прогресії для розгортання контейнерів наведено нижче.

```
//функція арифметичної прогресії розготрання контейнерів
function ArProg(n) {
  var sum = 0;
  for (var i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
// функція отримання навантаження контейнера
function random() {
```

```
    var rand = 1.5 - 0.5 + Math.random() * (2.1 - 1.5 +
0.1)
    rand = Math.round(rand);
    return rand;
  }
  //функція роботи алгоритму
  function Prog(t, k0, Ts0) {
    var k = k0 + ArProg(1);
    var St = k * random();
    var Ts = t-St-Ts0;

    switch (ts) {
      case ts < -0.1:
      return awaiting();
      break;
      case ts > 0.1:
      break;
      case ts = 0.1:
        return awaiting();
      break;
   }

   if (N = 5) {
     function undoCont();
   } else {awaiting()}
```

Програмний код для алогритму з використанням арифметичної прогресії для згортання контейнерів наведено нижче.

```
  //функція згортання контейнерів арифметичної прогресії
  function UnArProg(n) {
```

```
    var sum = 0;
    for (var i = n; i < n; i--) {
      sum -= i;
    }
    return sum;
  }
//функція очікування
  function awaitingPlus(n){
    var sum = 0;
    for (var i = 0; i <= n; i++) {
      sum += i;
    }
    return sum;
  }
//функція роботи алгоритму
 function undoCont(k) {
   unArgProg(n)
   awaiting()
   if (d <= 0.1 && d >= -0.1) {
     awaiting()
   } else {
     if (d > 0.1) {
       awaitingPlus()
     } else {
       if (d < -0.1){
         awaitingPlus()
         if (n = 3) {
           return StatusNew;
         }
       } else awaitingPlus();   } }
```

Умови функціонування алгоритму розгортання та згортання контейнерів з використанням арифметичної прогресії:

– Якщо навантаження на застосунок дорівнює знаходиться в рамках допустимого значення, то система знаходиться в стані «очікування», тобто додаткові контейнери на сервері не розгортаються;

– Якщо навантаження на сервер дорівнює значенню більш, ніж 0.1, то система розгортає певну кількість контейнерів на сервері відповідно до арифметичної прогресії, в іншому випадку, система знаходиться в стані «очікування»;

– Якщо при зростанні навантаження на сервер тричі підряд система надала більшу за необхідність кількість контейнерів, то на наступному кроці система переходить в стан «очікування»;

– Якщо на сервері після стану «очікування» кількість контейнерів знову перевищує необхідну, система переходить в стан «очікування»;

– Якщо система розгорнула більшу за необхідну кількість контейнерів п’ять разів підряд, то наступним кроком система виконує згортання контейнерів на ту кількість, яка була розгорнута в попередньому кроці;

– Якщо відбувається зменшення навантаження на сервер, то система переходить в стан «очікування» на три кроки, щоб вирівняти кількість розгорнутих контейнерів до необхідної. Якщо кількості контейнерів буде недостатньо, система запускає алгоритм з початку та розгортає контейнери з початкової кількості, що дорівнює 1.

Алгоритм розгортання та згортання контейнерів з використанням арифметичної прогресії було застосовано в двох випадках зміни навантаження на сервер:

а) при поступовому навантаженні:

Таблиця 6.2 – Значення змінних t_i при поступовому навантаженні на сервер

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
+1	+2	+5	+7	+11	+18	+25	+18	+10	+7	+5	+2	-4	-8	-14

t0 = 10	t1 = +1	t2 = +2	t3 = +5	t4 = +7	t5 = +11	t6 = +18	t7 = +25	t8 = +18
k0 = 2	k1 = k0+1 = 3	k2 = k1+2 = 5	k3 =k2+3 = 8	k4 = k3 = 8	k5 = k4(1) + 5 = 17	k6=k5+6 = 23	k7 = k6(1) + 8 = 38	k8 = k7(1) + 10 = 57
St0 = 10	St1 = 1.8	St2 = 3	St3 = 6	St4 = 0	St5 = 8.75	St6 = 12	St7 = 15.2	St8 = 19
Ts0 = 0	Ts1 = 1-1.8+0 = -0.8	Ts2 = 2-3-0.8=-1.8	Ts3 = 5-6-1.8=-2.8	Ts4 = 7 - 0 - 2.8 = 4.2	Ts5 = 11 - 8.75 -2.2= 0.05	Ts6 = 18-12 + 0.05=6.05	Ts7 = 25-15.2-5.85 = 3.95	Ts8 = 18-19-9.55 = -10.5
необхідна кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	недостатня кіл-ть	необхідна кіл-ть	недостатня кіл-ть	недостатня кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів
			система переходить в стан очікування					
				k4(1) = k4 + 4 = 12		k6(1) = k6 + 7 = 30	k7(1) = k7 + 9 = 47	
				St4(1) = 6.4		St6(1) = 11.9	St7(1) = 13.5	
				Ts4(1) = Ts4 - St4(1) = 4.2 - 6.4 = -2.2		Ts6(1) = Ts6 - St6(1) = 6.05-11.9= - 5.85	Ts7(1) = Ts7 - St7(1) =3.95-13.5= -9.55	
				надлишкова кіл-ть		надлишкова кіл-ть	надлишкова кіл-ть	
				контейнерів		контейнерів	контейнерів	

t9 = +10	t10 = +7	t11 = +5	t12 = 0	t13 = -4	t14 = -8	t15 = -14
k9 = k8 +11 = 68	k10=k9	k11 = k10	K12 = k11(1) = 57	k13 = k12 = 57	k14 = k13 = 57	k15 = k14 = 57
St9 = 17.6	St10=0	St11 = 0	St12 = 0	St13 = 0	St14 = 0	St15 = 0
Ts9 = 10-17.6 -10.55 = -18.15	Ts10 = 7 - 18.15 = -11.15	Ts11 = 5 - 11.15 = -6.15	Ts12 = 0 + 11.45 = 11.45	Ts13 = -4 + 11.45 = 7.45	Ts14 = -8 + 7.45 = -0.55	Ts15 = -15 -0.55 = -15.55
надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	недостатня кіл-ть	недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів	контейнерів	контейнерів при спаданні	контейнерів при спаданні	контейнерів	контейнерів
система переходить в стан очікування	система знаходиться в стані очікування	система згортає контейнери				
		k11(1) = k11 - 11 =57				
		St11(1) = -17.6				
		Ts11(1) = Ts11 + St11(1)=-6.15+17.6=11.45				
		недостатня кіл-ть				
		контейнерів				

б) при динамічному навантаженні:

Таблиця 6.3 – Значення змінних t_i при динамічному навантаженні на сервер

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
+3	+7	+17	+26	+35	+44	+35	+19	+11	+5	-3	-7

t0=10	t1=3	t2=7	t3=17	t4=26	t5=35	t6=44
k0 = 2	k1=k0+1=3	k2=k1(1)+3=8	k3=k2(1)+5=17	k4=k3(1)+7=27	k5=k4(1)+9=44	k6=k5(1)+11=65
St0 = 10	St1=1,5	St2=5.1	St3=9.5	St4=12.6	St5=13.5	St6=19.8
Ts0 = 0	Ts1=3-1,5+0=1,5	Ts2=7-5.1-1.7=0.2	Ts3=17-9.5-6.2=1.3	Ts4=26-12.6-11.3=2.1	Ts5=35-13.5-14.7=6.8	Ts6=44-19.8-11.2=13
необхідна кіл-ть	недостатня кіл-ть	недостатня кіл-ть	недостатня кіл-ть	недостатня кіл-ть	недостатня кіл-ть	недостатня кіл-ть
контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів
	k1(1)=k1+2=5	k2(1)=k2+4=12	k3(1)=k3+6=20	k4(1)=k4+8=35	k5(1)=k5+10=54	k6(1)=k6+12=77
	St1(1)=3,2	St2(1)=6.4	St3(1)=12.6	St4(1)=16.8	St5(1)=18	St6(1)=25.2
	Ts1(1)=1.5-3.2=-1.7	Ts2(1)=0.2-6.4=-6.2	Ts3(1)=1.3-12.6=-11.	Ts4(1)=2.1-16.8=-14.7	Ts5(1)=6.8-18=-11.2	Ts6(1)=13-25.2=-12.2
	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів

t7=35	t8=19	t9=11	t10=5	t11=-3	t12=-7
k7=k6(1)+13=90	k8=k7(1)+15=119	k9=k8+16=135	k10=k9=135	k11=k10=135	k12=k11(1)=119
St7=19.5	St8=27	St9=25.6	St10=0	St11=0	St12=0
Ts7=35-19.5-12.2=3.3	Ts8=19-27-26.1=-34.1	Ts9=11-25.6-34.1=-48.7	Ts10=5-0-48.7=-43.7	Ts11=-3-0-43.7=-46.7	Ts12=-7-0-21.1=-28.1
недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів при спаданні	контейнерів при спадані	контейнерів	контейнерів	контейнерів
		система переходить в стан очікування	система знаходиться в стані очікування	система згортає контейнери	система згортає контейнери
k7(1)=k7+14=104				k11(1)=k11-16=119	k12(1)=k12-15=104
St7(1)=29.4				St11(1)=-25.6	St12(1)=-27
Ts7(1)=3.3-29.4=-26.1				Ts11(1)=-46.7+25.6=-21.1	Ts12(1)=-28.1+27=-1.1
надлишкова кіл-ть				надлишкова кіл-ть	надлишкова кіл-ть
контейнерів при спаданні				контейнерів	контейнерів

6.4 Алгоритм керування контейнерами з використанням геометричної прогресії

Для застосування алгоритму керування контейнерами з використанням геометричної прогресії слід використовувати застосунки з послідовним навантаженням, де показник швидкості функціонування системи є головним чинником. Алгоритм керування контейнерами з використанням геометричної прогресії наведений в Додатку Д та в Додатку Е.

Геометрична прогресія має вид $b_n = b_1 \cdot q^{n-1}$, q=2. Контейнери в даному алгоритмі будуть додаватись в наступній кількості: 1, 2, 4, 8 і так далі поки на сервері буде вистачати пам’яті. Кожен контейнер може обробити певну кількість навантаження, що дорівнює діапазону значень [1.5: 2.1].

Програмний код для алгоритму з використання геометричної прогресії для розгортання контейнерів наведено нижче.

```
//функція геометричної прогресії розгортання
function GeomProg(n) {
    var sum = 0;
    for (var i = 1; i <= n; i+2) {
      sum *= i;
    }
    return sum;
  }
```

```
// функція отримання навантаження контейнера
  function random() {
    var rand = 1.5 - 0.5 + Math.random() * (2.1 - 1.5 +
0.1)
    rand = Math.round(rand);
    return rand;
  }
//функція роботи алгоритму
  function Prog(t, k0, Ts0) {
    var k = k0 + GeomProg(2);
    var St = k * random();
    var Ts = t - St - Ts0;

    switch (ts) {
      case ts < -0.1:
        return awaiting();
        break;
      case ts > 0.1:
        break;
      case ts = 0.1:
        return awaiting();
        break;
    }

    if (N = 5) {
      function undoCont();
    } else { awaiting() }
```

Програмний код для алгоритму з використанням геометричної прогресії для згортання контейнерів наведено нижче.

```
// функція геометричної прогресії згортання контейнерів
function UnGeomProg(n) {
      var sum = 0;
      for (var i = n; i < n; i++) {
        sum *= i;
      }
      return sum;
    }
// функція очікування
    function awaitingPlus(n) {
      var sum = 0;
      for (var i = 0; i <= n; i++) {
        sum += i;
      }
      return sum;
    }
//функція зворотньої геометричної прогресії
    function undoCont(k) {
      unGeomProg(n)
      awaiting()
      if (d <= 0.1 && d >= -0.1) {
        awaiting()
      } else {
        if (d > 0.1) {
          awaitingPlus()
        } else {
          if (d < -0.1) {
            awaitingPlus()
            if (n = 3) {
              return StatusNew;
```

```
            }
          } else awaitingPlus();
        }   } }
```

Умови функціонування алгоритму розгортання та згортання контейнерів з використанням геометричної прогресії:

– Якщо навантаження на застосунок знаходиться в допустимих значеннях, то система знаходиться в стані «очікування», тобто додаткові контейнери на сервері не розгортаються;

– Якщо навантаження на сервер дорівнює значенню більш, ніж 0.1, то система розгортає певну кількість контейнерів на сервері відповідно до геометричної прогресії, в іншому випадку, система знаходиться в стані «очікування»;

– Якщо при зростанні навантаження на сервер тричі підряд система надала більшу за необхідність кількість контейнерів, то на наступному кроці система переходить в стан «очікування»;

– Якщо на сервері після стану «очікування» кількість контейнерів знову перевищує необхідну, система переходить в стан «очікування»;

– Якщо система розгорнула більшу за необхідність кількість контейнерів п’ять разів підряд, то наступним кроком система виконує згортання контейнерів на ту кількість, яка була розгорнута, починаючи з першого кроку;

– Якщо відбувається зменшення навантаження на сервер, то система переходить в стан «очікування» на три кроки, щоб вирівняти кількість розгорнутих контейнерів до необхідної. Якщо кількості контейнерів буде недостатньо, система запускає алгоритм з початку та розгортає контейнери з початкової кількості, що дорівнює 1.

Алгоритм розгортання та згортання контейнерів з використанням геометричної прогресії було застосовано в двох випадках зміни навантаження на сервер:

а) при поступовому навантаженні:

Таблиця 6.4 – Значення змінних t_i при поступовому навантаженні на сервер

t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
+7	+11	+18	+25	+18	+10	+7	+5	+2	-4	-8	-14

t0 = 10	t1 = +1	t2 = +2	t3 = +5	t4 = +7	t5 = +11	t6 = +18	t7 = +25	t8 = +18
k0 = 2	k1=k0+1=3	k2 = k1+2 = 5	k3=k2+4=9	k4=k3=9	k5=k4(1)+16=33	k6=k5+32=65	k7=k6=65	k8=k7=65
St0 = 10	St1 = 1.8	St2 = 3	St3=7.6	St4=0	St5=28.8	St6=51.2	St7=0	St8=0
Ts0 = 0	Ts1 = 1-1.8+0 = -0.8	Ts2 = 2-3-0.8=-1.8	Ts3=5-7.6-1.8=-4.4	Ts4=7-4.4=2.6	Ts4=11-28.8-13.4=-31.2	Ts6=18-51.2-31.2=-64.4	Ts7=25-64.4=-39.4	Ts8=18-39.4=-21.4
необхідна кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів
			система переходить в стан очікування			система переходить в стан очікування	система знаходиться в стані очікування	система згортає контейнер
				k4(1)=k4+8=17				k8(1)=k8-1=64
				St4(1)=16				St8(1)=-1.8
				Ts4(1)=Ts4-St4(1)= 2.6-16=-13.4				Ts8(1)=Ts8-St8(1)=-21.4+1 .8=-19.6
				надлишкова кіл-ть				надлишкова кіл-ть
				контейнерів				контейнерів при спаданні
								система переходить в стан очікування

t9 = +10	t10 = +7	t11 = +5	t12 = 0	t13 = -4	t14 = -8	t15 = -14
k9=k8(1)=64	k10=k9=64	k11=k10=64	k12=k11	k13=k12	k14=k13	k15=k14
St9=0	St10=0	St11=0	St12=0	St13=0	St14=0	St15=0
Ts9=10-0-19.6=-9.6	Ts10=7-9.6=-2.6	Ts11=5-2.6=2.4	Ts12=0+2.4=2.4	Ts13=-4+2.4=-1.6	Ts14=-8-1.6=-9.6	Ts15=-14-9.6
надлишкова кіл-ть	надлишкова кіл-ть	недостатня кіл-ть	недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів при спаданні	контейнерів	контейнерів при спаданні	контейнерів при спаданні	контейнерів	контейнерів	контейнерів
система знаходиться в стані очікування	система знаходиться в стані очікування	система переходить в стан очікування	система знаходиться в стані очікування	система переходить в стан очікування	система знаходиться в стані очікування	система знаходиться в стані очікування

б) при динамічному навантаженні:

Таблиця 6.5 – Значення змінних t_i при динамічному навантаженні на сервер

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
+3	+7	+17	+26	+35	+44	+35	+19	+11	+5	-3	-7

t0=10	t1=3	t2=7	t3=17	t4=26	t5=35	t6=44
k0=2	k1=k0+1=3	k2=k1(1)+4=9	k3=k2+8=17	k4=k3=17	k5=k4(1)+32=65	k6=k5+64=129
St0=10	St1=1.5	St2=7.6	St3=16.8	St4=0	St5=54.4	St6=96
Ts0=0	Ts1=3-1.5+0=1.5	Ts2=7-7.6-2.3=-2.9	Ts3=17-16.8-2.9=-2.	Ts4=26-0-2.7=23.3	Ts5=35-54.4-7.1=-26.5	Ts6=44-96-26.5=-78.5
необхідна кіл-ть	недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	недостатня кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів	контейнерів
			система переходить в стан очікування			система переходить в стан очікування
	k1(1)=k1+2=5			k4(1)=k4+16=33		
	St2(1)=3.8			St4(1)=30.4		
	Ts2(1)=1.5-3.8=-2.3			Ts4(1)=23.3-30.4=-7.1		
	надлишкова кіл-ть			надлишкова кіл-ть		
	контейнерів			контейнерів		

t7=35	t8=19	t9=11	t10=5	t11=-3	t12=-7
k7=k6=129	k8=k7=129	k9=k8(1)=128	k10=k9=128	k11=k10=128	k12=k11=128
St7=0	St8=0	St9=0	St10=0	St11=0	St12=0
Ts7=35-0-78.5=-43.5	Ts8=19-0-43.5=-24,5	Ts9=11-0-23=-12	Ts10=5-0-12=-7	Ts11=-3-0-7=-10	Ts12=-7-0-10=-17
надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть	надлишкова кіл-ть
контейнерів	контейнерів	контейнерів при спаданні	контейнерів при спаданні	контейнерів при спаданні	контейнерів при спаданні
система знаходиться в стані очікування	система згортає контейнер	система знаходиться в стані очікування	система знаходиться в стані очікування	система знаходиться в стані очікування	система згортає контейнери
	k8(1)=k8-k1=129-1=128				k12(1)=k11-2=126
	St8(1)=-1.5				St12(1)=-3.8
	Ts8(1)=-24.5+1.5=-23				Ts12(1)=-17+3.8=-14.8
	надлишкова кіл-ть				
	контейнерів при спаданні				надлишкова кіл-ть
	система переходить в стан очікування				контейнерів при спаданні

6.5 Аналіз роботи алгоритмів

Під час аналізу прогресій було введено наступні поняття:

- Допуск системи з першого показання;
- Недостатня кількість розгорнутих контейнерів з першого показання;
- Допуск системи з другого показання.

Допуск системи з першого показання - цей показник визначає наскільки ефективно система зуміла скоригувати навантаження розгортаючи контейнери з першого показання. Ефективність роботи системи з першого показання обчислюється за формулою 6.6.

$$L_1 = \frac{100\%}{k} \cdot d_1, \tag{6.6}$$

де k – кількість отриманих значень ti;

d_1- кількість допуску з першого показання.

Недостатня кількість розгорнутих контейнерів з першого показання - означає, що алгоритм не зумів скоригувати відразу додане навантаження, тому після цього статусу викликається наступний крок для коригування показань. Цей показник не впливає на ефективність системи.

Допуск системи з другого показання - це показник за яким систем зуміла скоригувати навантаження після першого показання. Ефективність роботи системи з другого показання обчислюється за формулою 6.7.

$$L_2 = \frac{100\%}{k} \cdot d_2, \tag{6.7}$$

де k – кількість отриманих значень ti;

d_2- кількість допуску з першого показання.

6.5.1 Аналіз арифметичної прогресії

Під час досліджень навантажень на застосунок при вхідних параметрах t_i для поступового навантаження було виявлено, що даний алгоритм з першого показання зумів збалансувати 60% всіх навантажень. Недостатня кількість контейнерів була виявлена лише в 27% всіх операцій. Можно зробити загальний висновок, що даний алгоритм чудово підходить для систем з поступовим навантаженням і в тих системах, де коригування навантаження не є головним аспектом в роботі застосунка. Аналіз даних наведено в таблиці 6.6.

Таблиця 6.6 - Аналіз даних арифметичної прогресії для поступового навантаження

Допуск системи з першого показання	9/15	60%
Допуск системи з другого показання	3/15	20%
Недостатня кількість розгорнутих контейнерів з першого разу	4/15	27%

Наведені результати досліджень графічно зображені на рисунку 6.1.

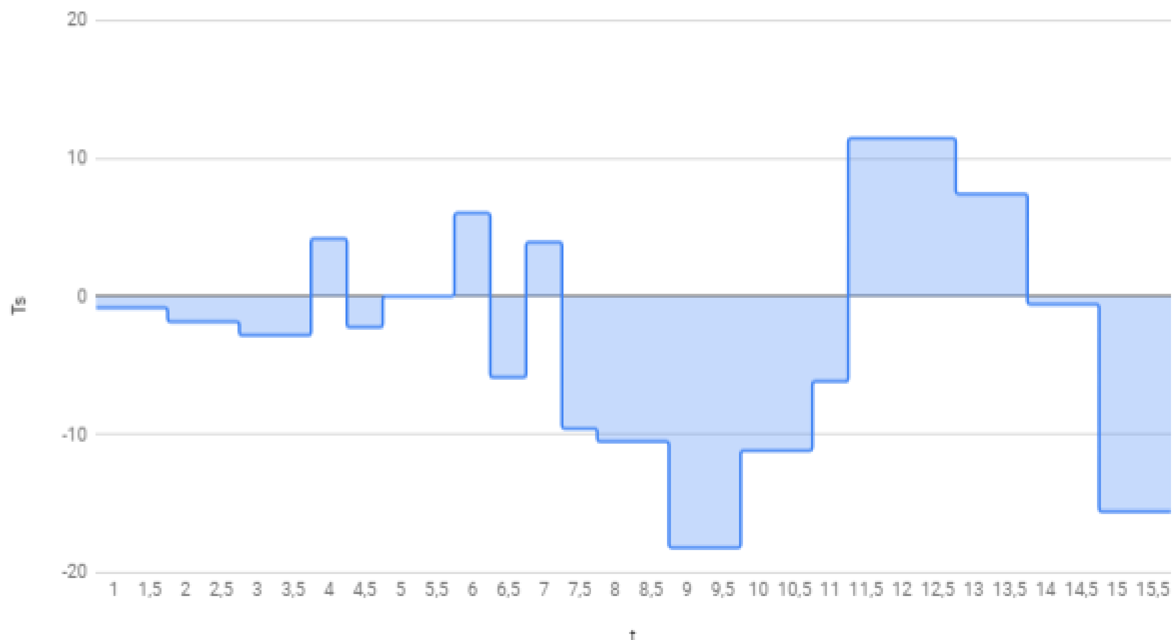


Рисунок 6.1 – Аналіз роботи системи з використанням алгоритму з арифметичною прогресією при поступовому навантаженні на сервер

Під час досліджень навантажень на застосунок при вхідних параметрах t_i для динамічного навантаження було виявлено, що даний алгоритм з першого показання зумів збалансувати 42% всіх навантажень. Недостатня кількість контейнерів була виявлена в 58% всіх операцій. Можно зробити загальний висновок, що даний алгоритм не підходить для систем з динамічним навантаженням. В системах, де головним критерієм є швидкість обробки даних даний алгоритм має недоліки. Аналіз даних наведено в таблиці 6.7.

Таблиця 6.7 - Аналіз даних арифметичної прогресії для динамічного навантаження

Допуск системи з першого показання	5/12	42%
Допуск системи з другого показання	7/12	58%
Недостатня кількість розгорнутих контейнерів з першого разу	7/12	58%

Наведені результати досліджень графічно зображені на рисунку 6.2.

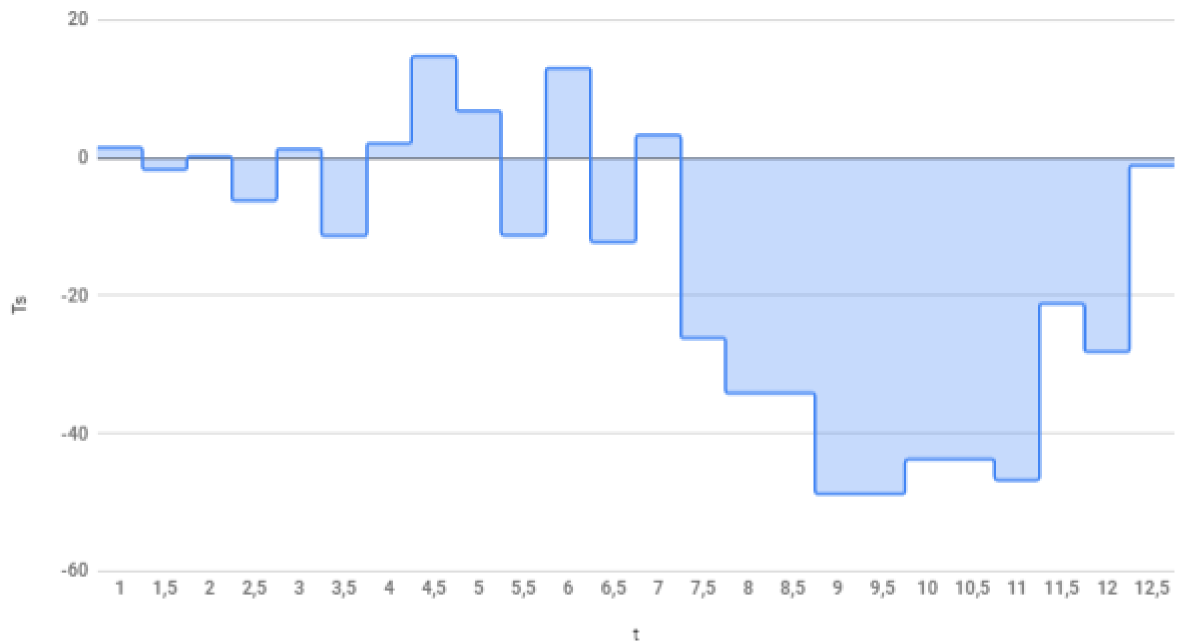


Рисунок 6.2 – Аналіз роботи системи з використанням алгоритму з арифметичною прогресією при динамічному навантаженні на сервер

6.5.2 Аналіз геометричної прогресії

Під час досліджень навантажень на застосунок при вхідних параметрах t_i для поступового навантаження було виявлено, що даний алгоритм з першого показання зумів збалансувати 80% всіх навантажень. Недостатня кількість контейнерів була виявлена лише в 13% всіх операцій. Проте 7% цих навантажень система збалансувала з другого єтапу. Можно зробити загальний висновок, що даний алгоритм чудово підходить для систем з поступовим навантаженням і в тих системах, де коригування навантаження є головним аспектом в роботі застосунка, що для цього необхідно швидко обробляти інформації. Аналіз даних наведено в таблиці 6.8

Таблиця 6.8 - Аналіз даних арифметичної прогресії для поступового навантаження

Допуск системи з першого показання	12/15	80%
Допуск системи з другого показання	1/15	7%
Недостатня кількість розгорнутих контейнерів з першого разу	2/15	13%

Наведені результати досліджень графічно зображені на рисунку 6.3.

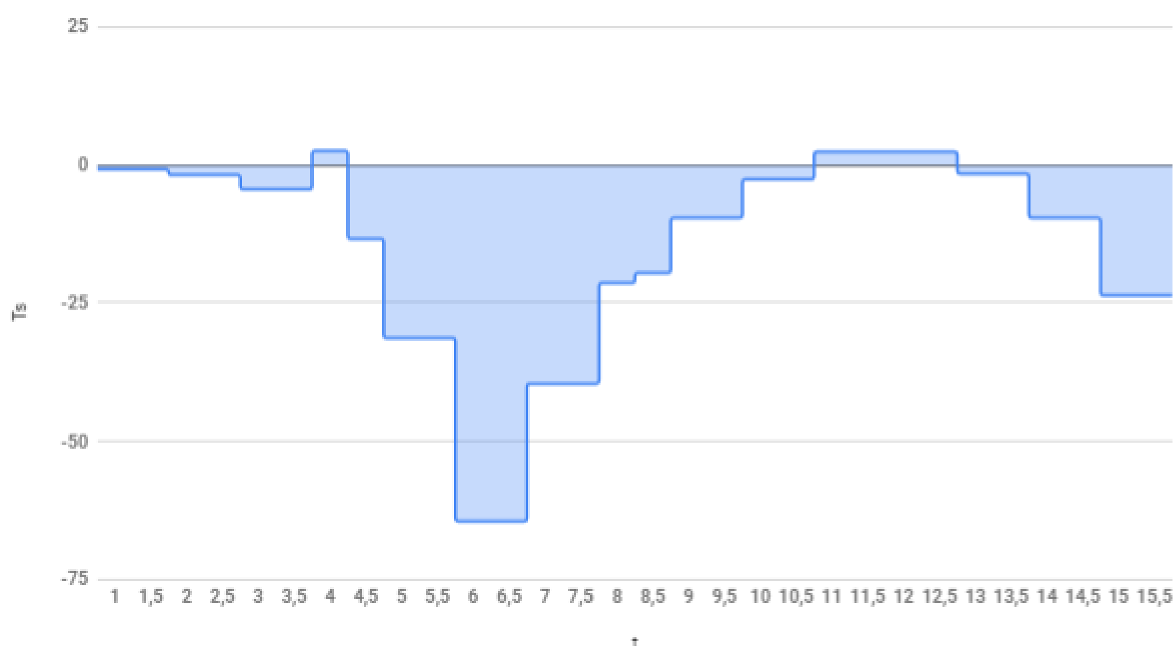


Рисунок 6.3 – Аналіз роботи системи з використанням алгоритму з геометричною прогресією при поступовому навантаженні на сервер

Під час досліджень навантажень на застосунок при вхідних параметрах t_i для динамічного навантаження було виявлено, що даний алгоритм з першого показання зумів збалансувати 83% всіх навантажень. Недостатня кількість контейнерів була виявлена в 17% всіх операцій. Проте 17% цих навантажень система збалансувала з другого етапу. Можно зробити загальний висновок, що даний алгоритм підходить для систем з динамічним навантаженням. В системах, де головним критерієм є швидкість обробки даних даний алгоритм має недоліки. Аналіз даних зображено в таблиці 6.9

Таблиця 6.9 - Аналіз даних арифметичної прогресії для динамічного навантаження

Допуск системи з першого показання	10/12	83%
Допуск системи з другого показання	2/12	17%
Недостатня кількість розгорнутих контейнерів з першого разу	2/12	17%

Наведені результати досліджень графічно зображені на рисунку 6.4.

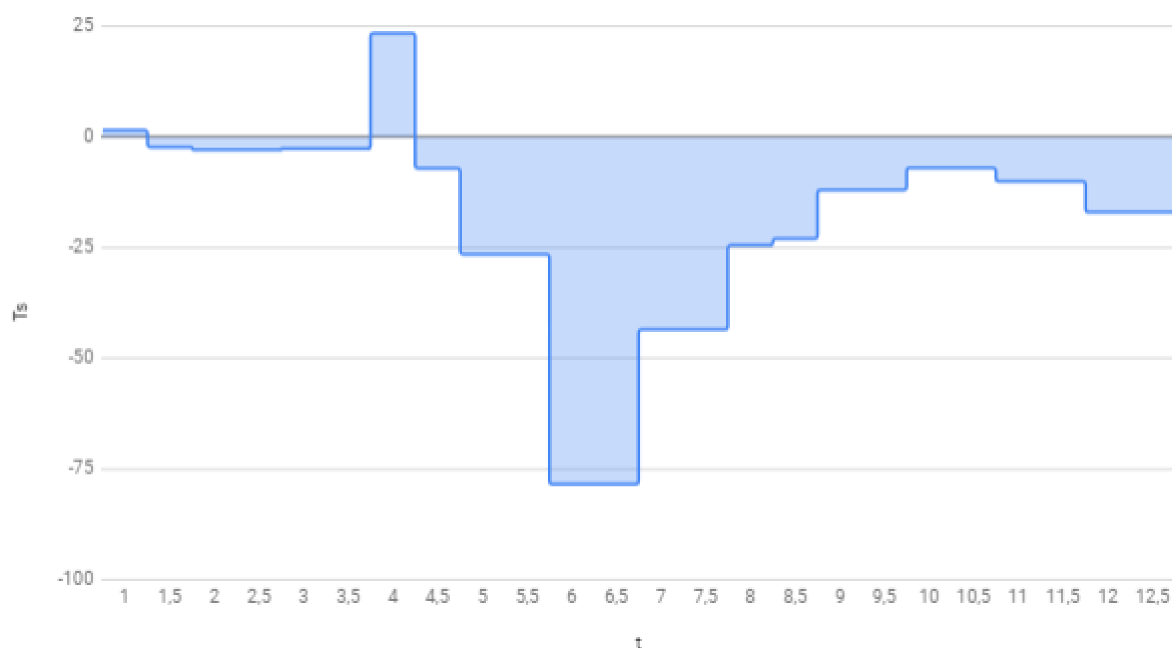


Рисунок 6.4 – Аналіз роботи системи з використанням алгоритму з геометричному прогресією при динамічному навантаженні на сервер

6.5.3 Порівняння ефективності алгоритмів, що застосовується

Порівнюючи два розроблених алгоритма з використанням арифметичної та геометричної прогресій видно, що кожен з методів компенсування навантаження має свої переваги та недоліки.

Алгоритм з використання арифметичної прогресії вирішує проблеми застосунків котрим необхідно відношення ціни та швидкості. Просто якщо є вірогідність, що на цей застосунок може прийти велика кількість навантаження, то

ліпше застосувати алгоритм з геометричною прогресією, адже вона зуміє швидше збалансувати навантаження.

Використання алгоритма з геометричною прогресією має перевагу над арифметичною тим, що він збалансовує навантаження відразу. Ця перевага чудово підходить для застосунків в яких головним параметром є швидкодія всієї системи, проте надлишком контейнерів можно знехтувати.

Наведені результати досліджень графічно зображені на рисунках 6.5 та 6.6.

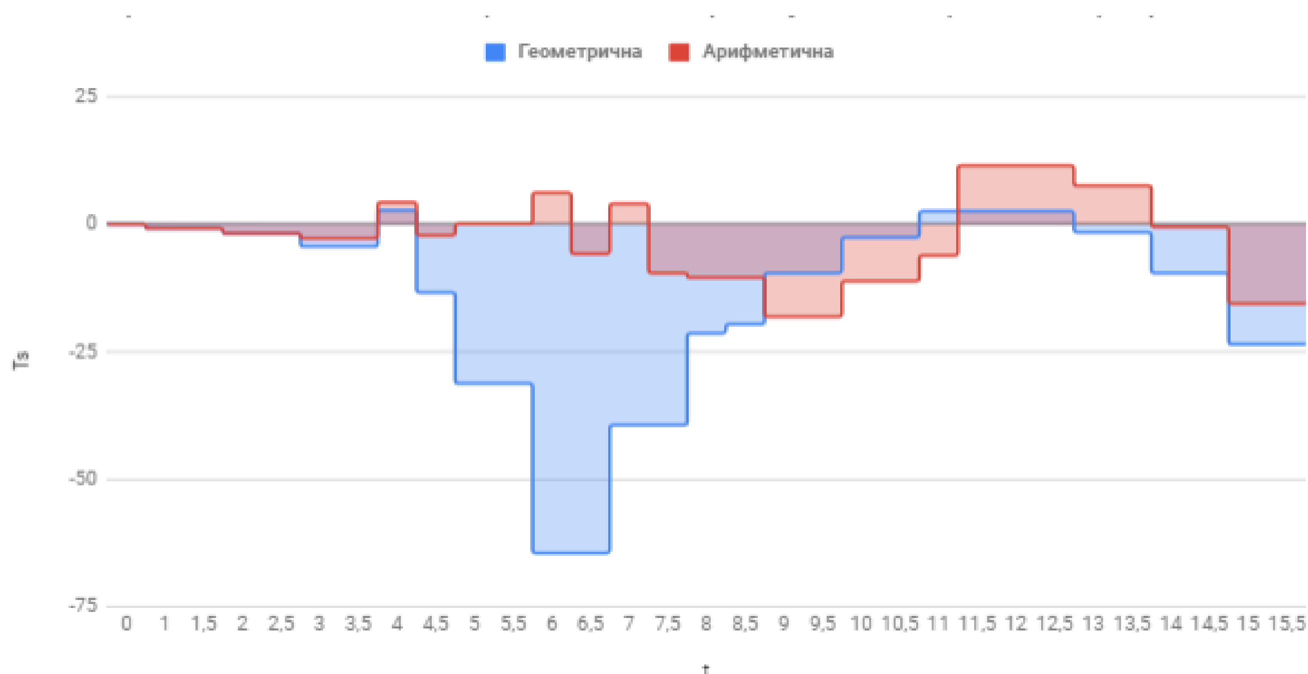


Рисунок 6.5 – Порівняльний аналіз роботи системи з використанням арифметичної та геометричної прогресій при поступовому навантаженні на сервер

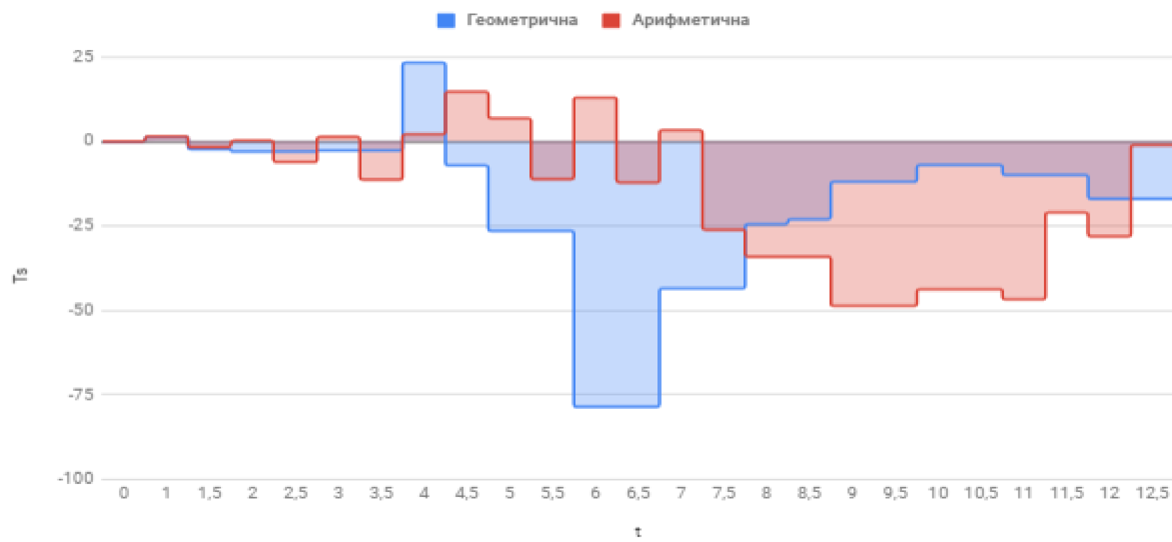


Рисунок 6.6 – Порівняльний аналіз роботи системи з використанням арифметичної та геометричної прогресій при динамічному навантаженні на сервер

7 РОЗРОБКА СТАРТАП – ПРОЕКТУ

7.1 Опис ідеї проекту (товару, послуги, технології)

Система відноситься до засобів обробки данних та управління обчислювальними ресурсами в умовах контейнерної віротуалізації. Призначена для автоматичного розгортання/згортання контейнерів на сервері реагуючи на вхідний параметр відклику часу взаємодії користувача з додатком.

Автоматизация масштабування - комплекса система контролю навантаження серверів з автоматичним управлінням кластерами контейнерів.

Таблиця 7.1 - Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Автоматизація розгортання та згортання контейнерів для застосунків	Будь-яка система, яка використовує обчислювання	Автоматизація, економія фінансів, збільшення ефективності використання обчислювальних ресурсів

Застосування ефективних алгоритмів та сучасних рішень робить системи управління конкурентоспроможною. В Україні подібних рішень не має, отже наявність продукту, що є ефективни та не гіршим за зарубіжних є дуже перспективним.

Таблиця 7.2 - Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Gigacloud	Balt			
1.	Вартість ПЗ	Низька	Висока	Висока			+

Продовження таблиці 7.2

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Gigacloud	Balt			
2.	Час обробки	Високий	Середній	Низький		+	
3.	Автоматизація	85 %	95%	70%		+	
4.	Споживачі (відомий бренд)	-	+	+	+		

7.2 Технологічний аудит ідеї проекту

Таблиця 7.3 - Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1.	Розгортання/згортання контейнерів	Програмне забезпечення на серверній технології Nodejs	Наявні	Так
2.	Алгоритми розподілення	Алгоритм вирішення розміщення	Наявні	Так

7.3. Аналіз ринкових можливостей запуску стартап-проекту

Найголовнішою характеристикою даної продукції є співвідношення ціни та можливостей, що відповідають високому рівню якості. Тому, зависока ціна іноземних аналогів та їх недоступність для широкого кола споживачів робить даний товар перспективним для застосування як в Україні,так і закордоном.

Таблиця 7.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од.	0. Тільки закордонні, немає аналогів в Україні.
2	Загальний обсяг продаж, ум.од/час	Поодиничний продаж, закордонних товарів. В середньому – 600 доларів за річний доступ.
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень).	Новітня технологія потребує ресурсів, практичної перевірки та доробки універсального алгоритму.
5	Специфічні вимоги до стандартизації та сертифікації.	Не потребує сертифікації
6	Середня норма рентабельності в галузі (по ринку), %	70

Основною споживчою аудиторією є банківські системи, провайдери хмаревих технологій, дата-центри. Так, як вони процюють з обробкою інформації «клиєнт-сервер-клієнт», то використання системи управління обчсилювальними ресурсами є перспективним вирішенням для покращення послуг, котрий не залежить від обсягу виконуваних навантажень.

Таблиця 7.5 - Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів
1. Дешевизна алгоритму	Збільшення продуктивності систем надання обчислювальних потужностей з клієнтами.	Зацікавлені в купівлі дешевшого.	Задовільна цінова політика, відповідність ДСТУ/ ISO, сумісність з ПК, висока якість, гарні технічні характеристики (час обробки, точність).
2. Зменшення ризику виникнення людського фактору.		Зацікавлені в кращій якості.	
3. Купувати вітчизняне		Зацікавлені в підтримці національного виробника.	
4.Отримати закордонний аналог по якості за українською ціною		Зацікавлені за менші гроші отримати високоякісний матеріал.	

Основною загрозою є впровадження – в Україні застосовують закордонні аналоги, тому необхідна велика кількість тестування для уточнення критеріїв алгоритму, щоб перевірити ефективність та мінімізувати час впровадження до центрів обробки даних. Також, так як технологія є новою та унікальною важливим пунктом є ретельний аналіз роботи системи.

Таблиця 7.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1.	Конкурентоспроможність	Новий товар на ринку-невідомий бренд-низька конкурентоспроможність	Реклама, вдалий маркетинговий проект, залучення дилерів, спонсорів до співпраці.
2.	Невідомий бренд	Невідомий новий товар невідомої фірми.	Розкрутка товару, розповсюдження серед ЦОД і в банківських центрах.
3.	Комерційна таємниця	Можливість відкриття технології невідомими робітниками.	Підписання строгих контрактів з несенням матеріальної компенсації.
4.	Впровадження	Аналіз створених алгоритмів, час впроваджень системи.	Чіткий аналіз системи, моніторинг роботи алгоритмів.

Найголовнішою можливістю є залучення іноземних спонсорів та можливостей реалізації товару закордоном.

Таблиця 7.7 - Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1.	Залучення іноземних спонсорів.	Залучення іноземних капіталів, підтримка спонсорів.	Укладення договорів про співпрацю та налагодження торгових контактів.

Продовження таблиці 7.7

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
2.	Налагодження зв'язків з ЦОД та приватними хостинговими компаніями	Залучення дистриб'юторів.	Пошук та налагодження зв'язків з компаніями.

Таблиця 7.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможни
Тип конкуренції	Монополія, на даний час в Україні немає аналогів.	Розкрутка національного товару, залучення інвесторів.
За рівнем конкурентної боротьби	Міжнаціональний	Випуск аналогу закордонних продуктів, дешевша ціна на аналогічну продукцію з новітнім алгоритмом.
За галузевою ознакою	Зовнішньогалузева	Використовується в усіх центрів обробки даних.
Конкуренція за видами товарів	Товарно-родова	Власні розробки, унікальна технологія, реклама.
За характером конкурентних переваг	Цінова	Ціна нижча за іноземні аналоги.
За інтенсивністю - марочна/не марочна	Не марочна	Реклама, покращення технологій

Таблиця 7.9 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Немає, або закордонні	Гнучкі ціни,розмір капіталовкладень	Велика концентрація постачальників	Якісна продукція	Вища ціна, вищі змінні витрати
Висновки	Невисока конкурентна боротьба	Є можливості входу в ринок, наявні потенційні конкуренти	Постачальники диктують умови роботи на ринку, наприклад, ціну та швидкість розповсюдження	Впровадження та оновлення алгоритмів	Обмежень немає

Робота на ринку є можливою, залежить від купівельної спроможності установ. Так як вони купують набагато дорожчий товар закордонних виробників, можна зробити висновок про те, що даний товар буде користуватись періодичним попитом. Також, важливим є фактор розповсюдження шляхом постачальників. Конкурентна боротьба є високою, проте лише з закордонними аналогами, адже таких в Україні немає.

Таблиця 7.10. Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Час	Менша необхідність машинної праці та економія часу.
2	Ціна	Ґрунтується на собівартості-отже є нижчою.
3	Якість	Не гірша за закордонні аналоги.

Таблиця 7.11 - Порівняльний аналіз сильних та слабких сторін власного про

№	Фактор конкурентоспр	Бали 1-20	Рейтинг товарів-конкурентів						
			–3	–2	–1	0	+1	+2	+3
1	Час	15				+			
2	Ціна	18	+						
3	Якість	14						+	

Таблиця 7.12 - SWOT- аналіз стартап-проекту

Сильні сторони: Собівартість Ціна Рентабельність Новітність	Слабкі сторони: Невідомий бренд Великі капіталовкладення Необхідність встановленого додаткового ПЗ
Можливості: Вихід на закордонний ринок Забезпечення споживчих потреб Дохід	Загрози: Викриття комерційної таємниці Недостатня реалізація

Таблиця 7.13 - Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Практичне використання алгоритму,	+	1 рік
2	Вдосконалення	+	1 рік
3	Залучення іноземних фахівців	+	0,5 року
4	Рекламна кампанія	+	2 роки

Продовження таблиці 7.13

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
5	Отримання міжнародних сертифікатів	+	2 роки

Обраною альтернативою є залучення рекламної кампанії, розкрутка нової системи.

7.4. Розроблення ринкової стратегії проекту

Таблиця 7.14 - Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	ЦОД	Не готові	Необхідно	Мінімальна	Важка
2	Хостингові компанії	Готові	Необхідно	Мінімальна	Легка
Які цільові групи обрано: наукові центри.					

Таблиця 7.15 - Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Хмарні обчислення	Стратегія лідерства по витратах	• ціна • простота застосування • продуктивність роботи • низькі витрати	Ретельний контроль за постійними витратами, зниження виробничих, збутових і рекламних витрат, проведення інвестиції, спрямованої на зменшення витрат, ретельне опрацювання алгоритму.

Таблиця 7.16 - Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1	Так	Так	Ні, продукт застосовує власний алгоритм для ПЗ.	Стратегія наслідування лідера

Таблиця 7.17 - Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Дешевизна закупівлі підтримки ПЗ.	Стратегія наслідування лідера	- ціна - простота виготовлення - кількість виробленої продукції - низькі витрати	-зниження цін -вітчизняне виробництво -невеликі обсяги - універсальність для кожного пацієнта
2	Зменшення ризику людської помилки.			
3	Купувати вітчизняне.			
4	Отримати закордонний аналог по якості за українською ціною.			

7.5. Розроблення маркетингової програми стартап-проекту

Таблиця 7.18 - Визначення ключових переваг концепції потенційного товару

№п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Дешевизна ПЗ	Низька ціна	Ціна

Продовження таблиці 7.18

№п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
2	Купувати вітчизняне	Національний продукт	Якість
3	Отримати закордонний аналог по якості за українською ціною	Національний продукт	Універсальність

Таблиця 7.19 - Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Алгоритм прискорює роботу центрів обробки даних, що дозволяє клієнтав покращувати свої показники.		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1. Висока швидкість 2. Точність у обробці зображень 3. Сумісний з більшістю ПК		
	Якість: стандарти, нормативи, параметри тестування тощо Стандартизація відповідно до ДСТУ,ISO. Регламентується НД, СРМ.		
	Пакування відсутнє.		
	Марка: назва організації-розробника + назва товару		

Потенційний товар буде захищено від копіювання: патентування, сертифікати відповідності.

Таблиця 7.20. Визначення меж встановлення ціни (із розрахунку на 1 рік користування)

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
60000 грн.	30-40 тис. грн.	50-60 тис. грн	20-25 тис. грн

Таблиця 7.21 - Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Закупівля можлива після інструктажу та презентації ПЗ, з детальним роз’ясненням можливостей та обмежень програми.	Маркетингові дослідження, обслуговування проданих товарів, Прийняття на себе ризику торгових угод	Канал нульового рівня (прямий маркетинг)	Торгівля через веб-сайт, що належить виробнику.

Таблиця 7.22. Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Довіра до інженерів, що рекомендують товар	Державні інститути, наукові установи, ЦОД.	Налагодження контактів з ЦОД	Донести основну ідею, цінову політику, якість	Продемонструвати переваги перед існуючим товарами

ВИСНОВКИ

В роботі розроблено систему управління обчислювальними ресурсами застосунків в умовах контейнерної віртуалізації з двома алгоритмами, котрі використовують геометричну та арифметичну прогресії. Розроблена система дозволяє автоматично розгортати та згортати контейнери на сервері та автоматично реагувати на зміну навантаження на застосунок,

Розроблено структурну схему, що пояснює роботу системи в комплексі з додатковими програмами для управління контейнерів. В комплексі розроблювана система використовується в компанії, що має свої застосунки та автоматично контролює навантаження.

Основною работаю над системою є розробка унікально нових двох алгоритмів для автоматичного реагування на зміну навантаження. За основу було взято дві прогресії: арифметична та геометрична. Під час досліджень було використано безліч варіантів компоновки чих прогресії, проте найоптимальнішим було використання саме таких зв’язків: арифметична для розгортання починаючи від 1 та арифметична для згортання з останньо додатнього числа під час розгортання; геометрична починаючи від 2 для розгортання та геометрична для згортання починаючи від 2.

Систему можуть використовувати центри обробки даних, постачальники послуг хостингу та інші компанії, які використовують віртуалізацію.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Технології віртуалізації [Електронний ресурс] – Режим доступу до ресурсу: https://studfiles.net. (https://studfiles.net/preview/5740672/page:19/)

2. Віртуалізація на рівні операційної системи [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org. (https://uk.wikipedia.org/wiki/%D0%92%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F_%D0%BD%D0%B0_%D1%80%D1%96%D0%B2%D0%BD%D1%96_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D1%97_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8)

3. Обзор технологии контейнеризации [Електронний ресурс] – Режим доступу до ресурсу: https://www.cloud4y.ru. (https://www.cloud4y.ru/about/news/obzor-tekhnologii-konteynerizatsii/)

4. Установка Docker и его использование [Електронний ресурс] – Режим доступу до ресурсу: https://unihost.com (https://unihost.com/help/ru/how-to-install-docker/)

5. Мережева файлова система [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/. (https://ru.wikipedia.org/wiki/UnionFS)

6. Мониторинг с помощью Prometheus [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/southbridge/prometheus-monitoring-ba8fbda6e83.

7. An Introduction to Virtualization [Електронний ресурс]. – 2004. – Режим доступу до ресурсу: http://www.kernelthread.com/publications/virtualization/.

8. The evolution of virtualization [Електронний ресурс] – Режим доступу до ресурсу: https://searchservervirtualization.techtarget.com/definition/virtualization.

9. Innovative data virtualization tools that offer a radically new approach to delivering real time business insights [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://www.datawerks.com/.

10. Production-Grade Container Orchestration Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/.

11. Amazon Elastic Container Service [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/ru/ecs/.

12. Official site of Nomad [Електронний ресурс] – Режим доступу до ресурсу: https://www.nomadproject.io/.

13. Swarm mode overview [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/engine/swarm/.

14. Official site of Rancher [Електронний ресурс] – Режим доступу до ресурсу: https://rancher.com/.

15. Технологические подходы к виртуализации ПО [Електронний ресурс] – Режим доступу до ресурсу: https://www.bytemag.ru/articles/detail.php

16. Анализ современных технологий виртуализации [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/company/southbridge/blog/212985/.

17. Погорілий С.Д.,. Технологія віртуалізації. Динамічна реконфігурація ресурсів обчислювального кластера / Погорілий С.Д.,, Білоконь І.В.,, Бойко Ю.В.,. // Математичні машини і системи. – 2012. – №3. – С. 3–18.

18. Палагин А.В. Об ЭВМ с виртуальной архитектурой / А.В. Палагин // Управляющие системы и машины. − 1999. − № 3. − С. 33 – 43.

19. Концепція створення гнучких гомогенних архітектур кластерних систем / С.Д. Погорілий, Ю.В. Бойко, Д.Б. Грязнов [та ін.] // Матеріали шостої міжнар. наук.-практ. конф. з програмування УкрПРОГ`2008. Проблеми програмування, (Київ, 27–29 травня 2008 р.). − 2008. − № 2-3. − С. 84 – 90.

20. Not All Server Virtualization Solutions Are Created Equal [Електронний ресурс] / Andre Metelo // IBM Corporation. – Режим доступу: ftp://ftp.software.ibm.com/software/systemz/Not_All_Server_ Virtualization_Solutions_Are_Created_Equal.pdf.

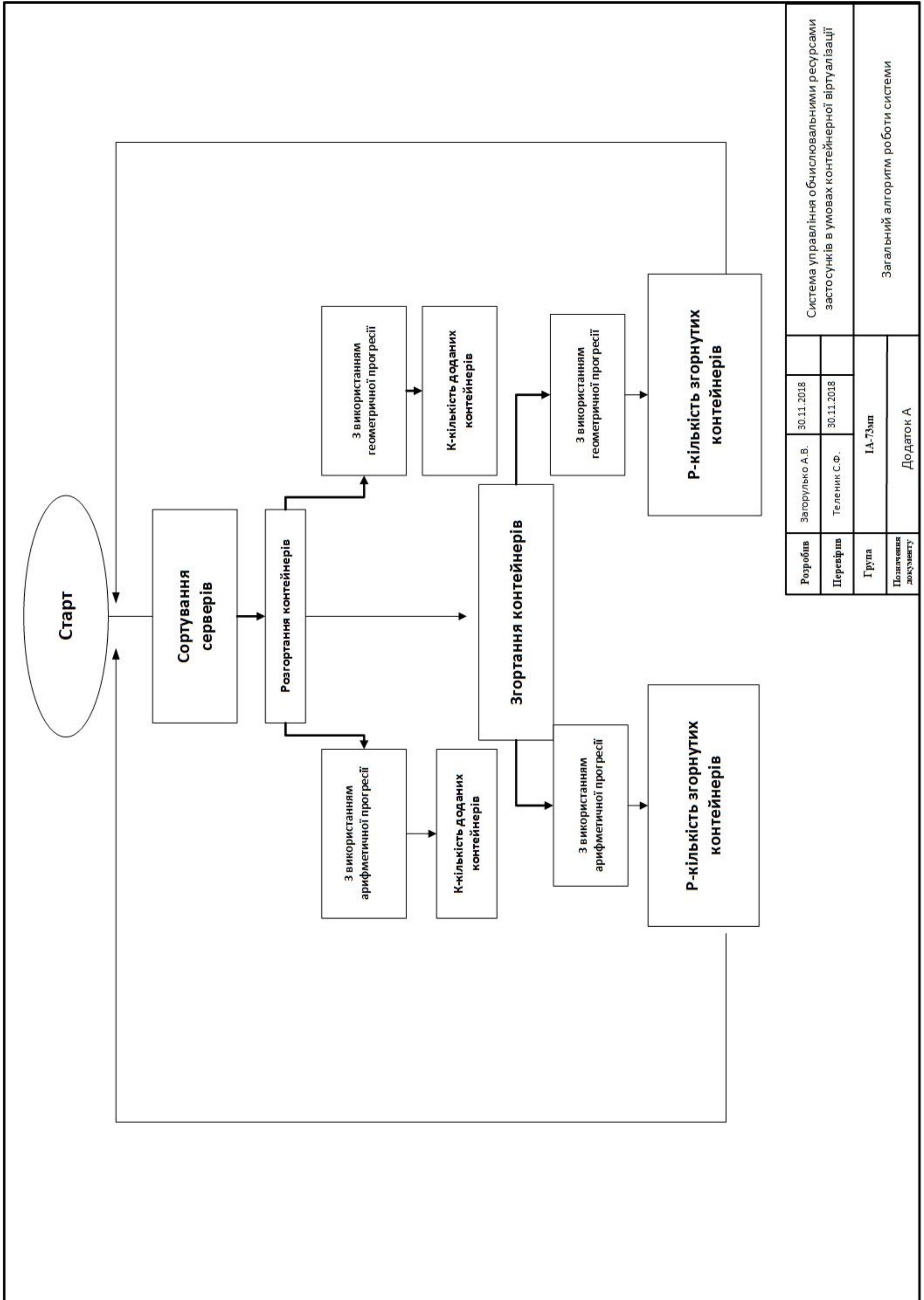
21. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors / S. Soltesz, H. Potzl, M. Fiuczynski [et al.] // Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. – New York, 2007. – 14 p.

22. Pod в Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/company/flant/blog/427819/.

23. Сети Docker изнутри [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/post/333874/.
24. Инфраструктура с Kubernetes как доступная услуга [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/company/flant/blog/341760/.
25. Разница между виртуализацией на уровне ОС и аппаратной виртуализацией [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/company/infobox/blog/247595/.
26. Контейнери і віртуалізація: швидше, надійніше [Електронний ресурс] – Режим доступу до ресурсу: http://it-ua.info/news/2017/01/20/konteyneri-vrtualzacya-shvidshe-nadynshe.html.
27. Боттомли Д. Ажиотаж вокруг контейнеров / Джеймс Боттомли. // Журнал сетевых
решений/LAN. – 2014. – №10.
28. Нанян С.М. Виртуальные контейнеры Docker: назначение и особенности применения
/ Нанян С.М., Ничушкина Т.М.. // Инженерный вестник. – 2015. – №2.
29. Контейнерна віртуалізація: стандарти [Електронний ресурс] – Режим доступу до ресурсу: http://it-ua.info/news/2016/08/15/konteynerna-vrtualzacya-skoro-budut-standarti.html.
30. Сравнение ECS, Nomad, Docker Swarm, Kubernetes, DC/OS и Rancher [Електронний ресурс] – Режим доступу до ресурсу: https://dev-ops-notes.ru/cloud/сравнение-ecs-nomad-docker-swarm-kubernetes-dcos-rancher/amp/.
31. Что происходит в Kubernetes при запуске kubectl run [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/company/flant/blog/342658/.
32. Обзор технологии контейнеризации [Електронний ресурс] – Режим доступу до ресурсу: https://www.cloud4y.ru/about/news/obzor-tekhnologii-konteynerizatsii/.
33. Docker from the Ground Up: Working with Containers [Електронний ресурс] – Режим доступу до ресурсу: https://code.tutsplus.com/series/docker-from-the-ground-up-working-with-containers--cms-1161.

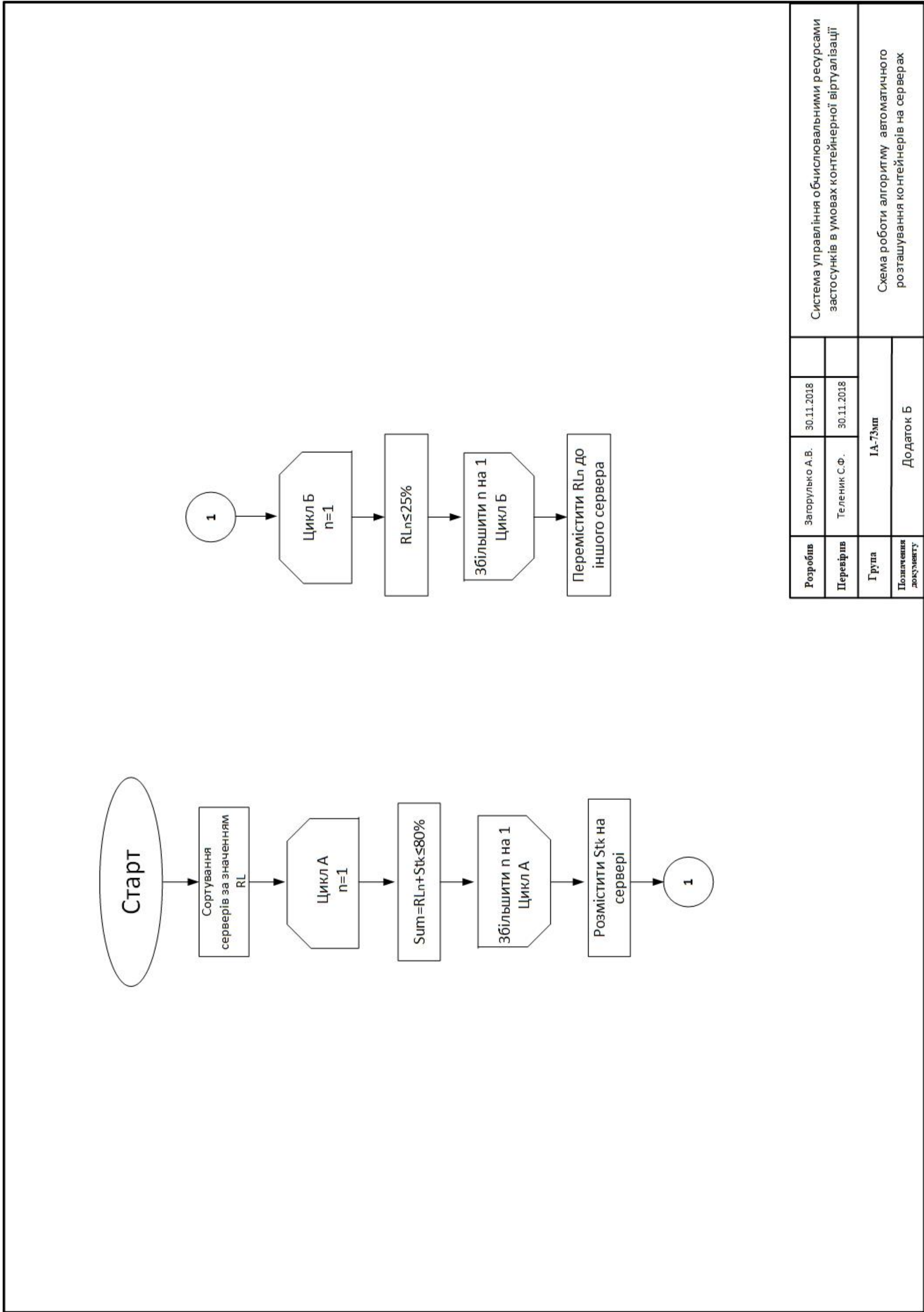
ДОДАТОК А

Загальний алгоритм роботи системи



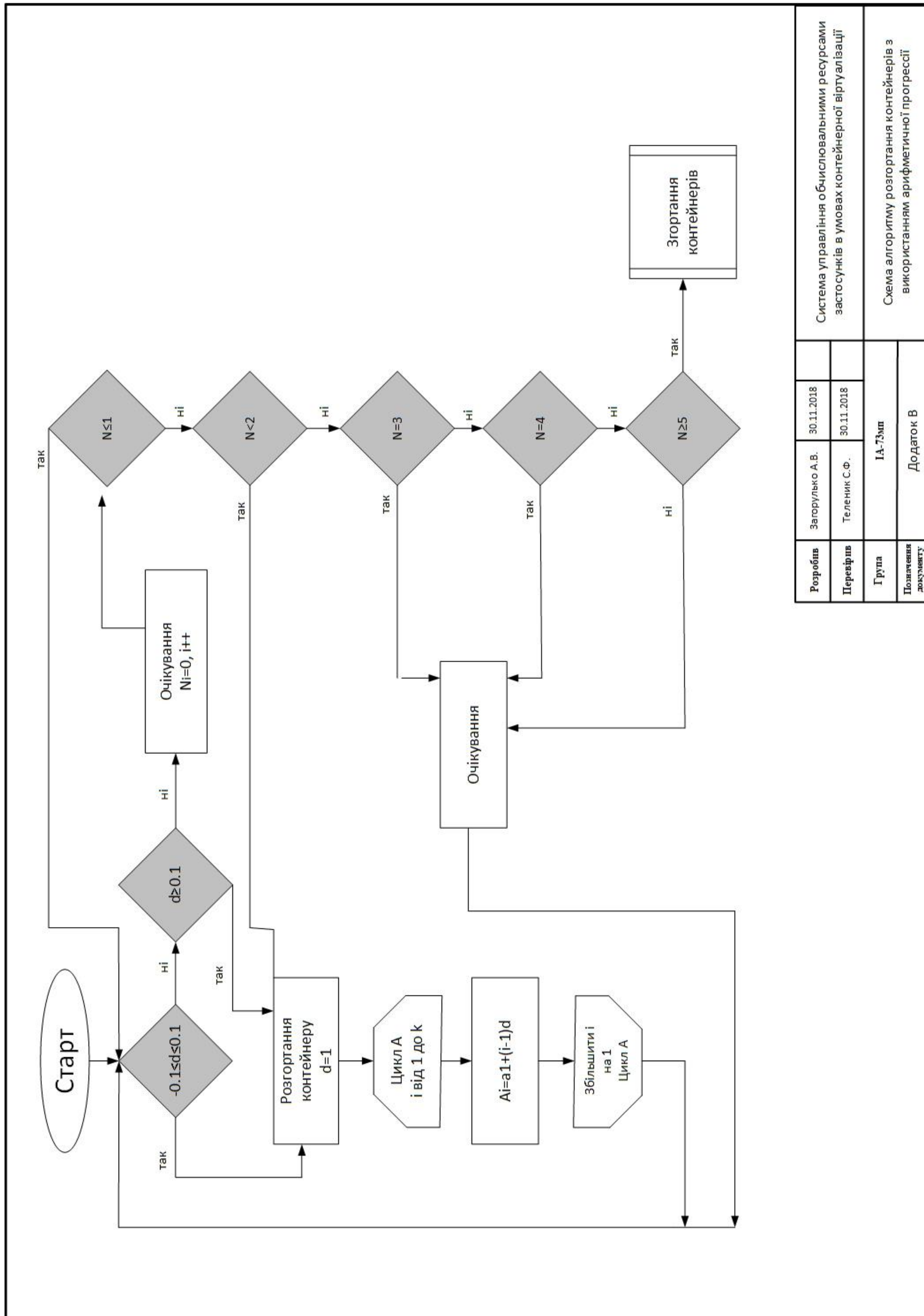
ДОДАТОК Б

Схема роботи алгоритму автоматичного розташування контейнерів на серверах



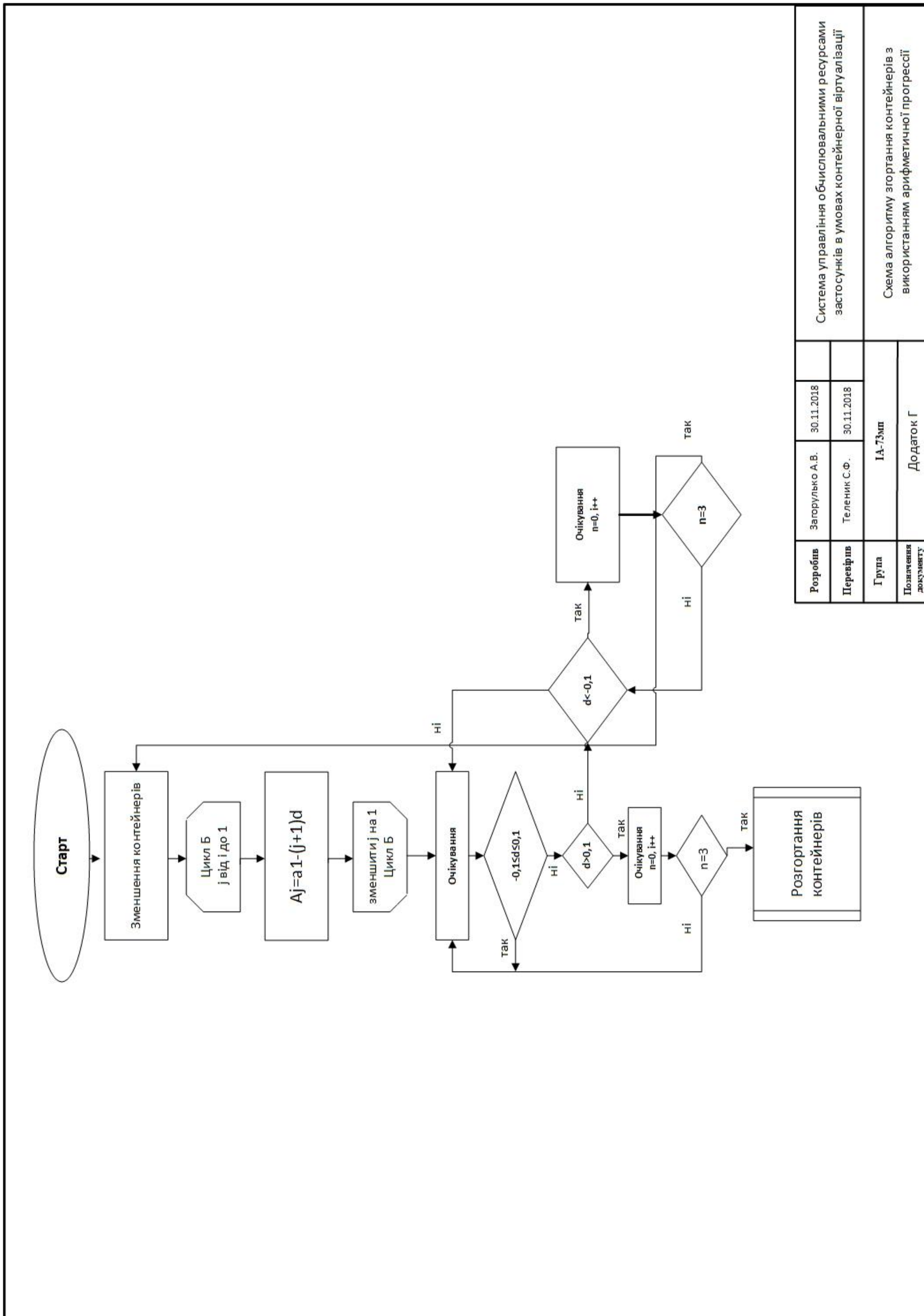
ДОДАТОК В

Схема алгоритму розгортання контейнерів з використанням арифметичної прогресії



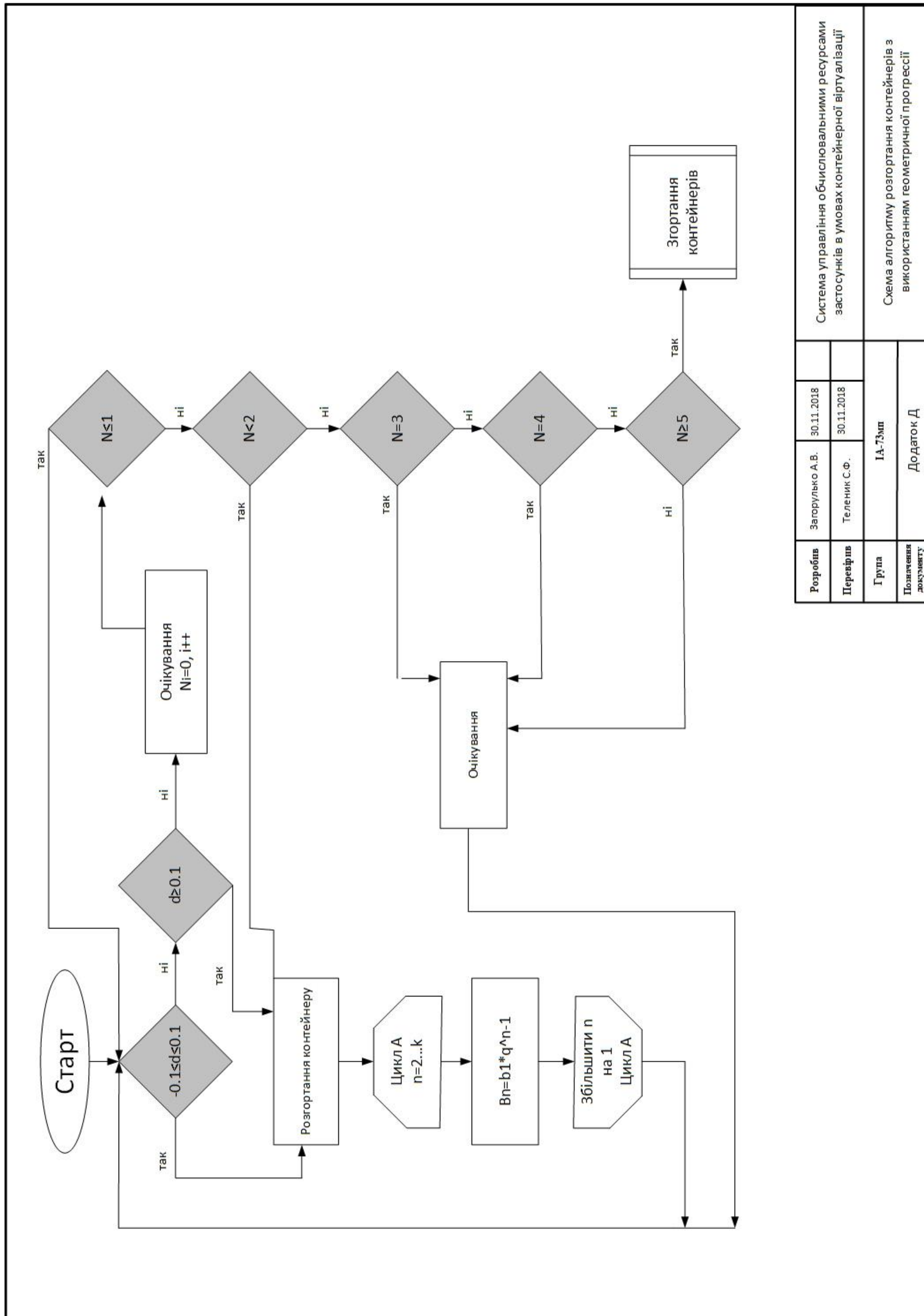
ДОДАТОК Г

Схема алгоритму згортання контейнерів з використанням арифметичної прогресії



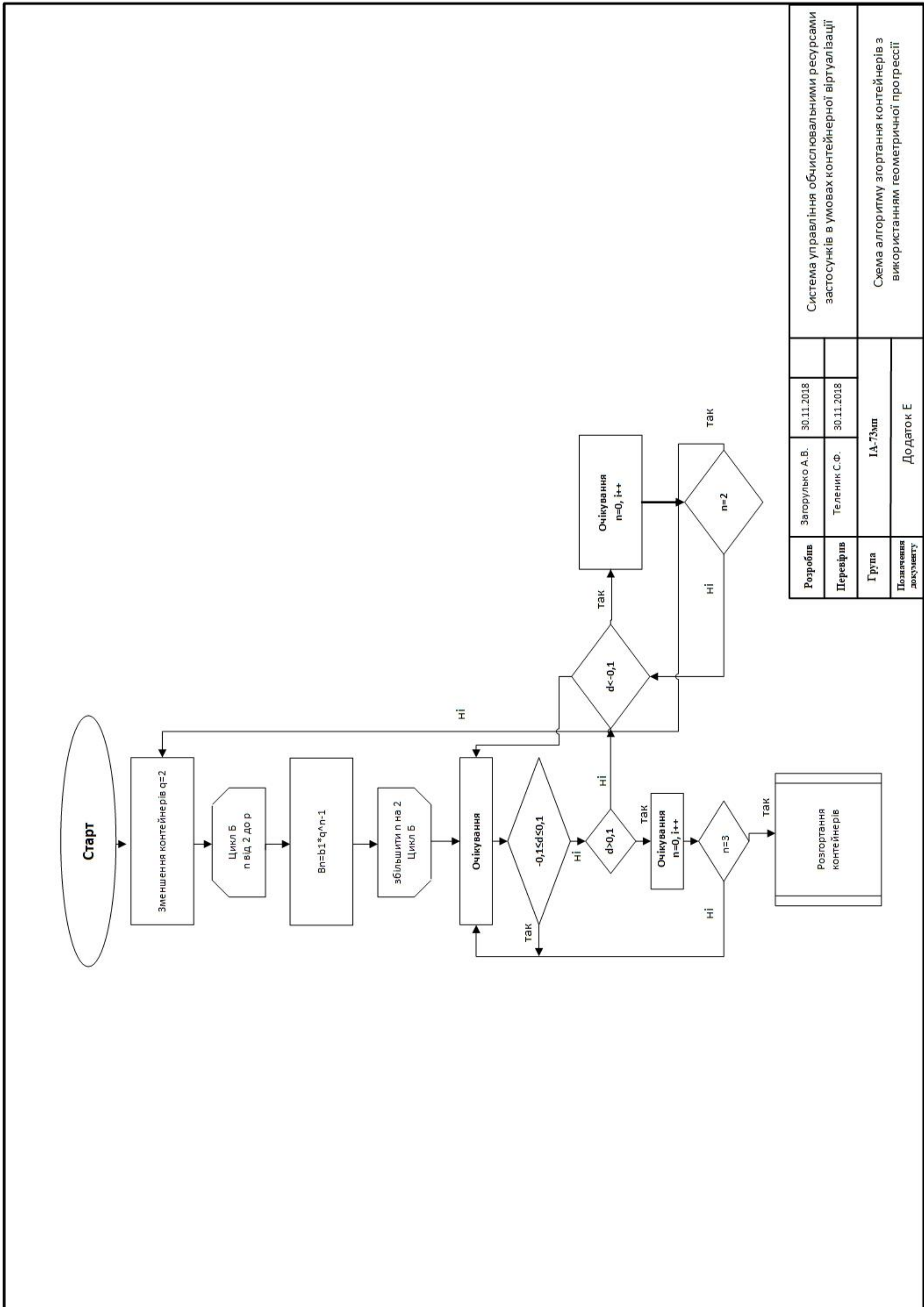
ДОДАТОК Д

Схема алгоритму розгортання контейнерів з використанням геометричної прогресії



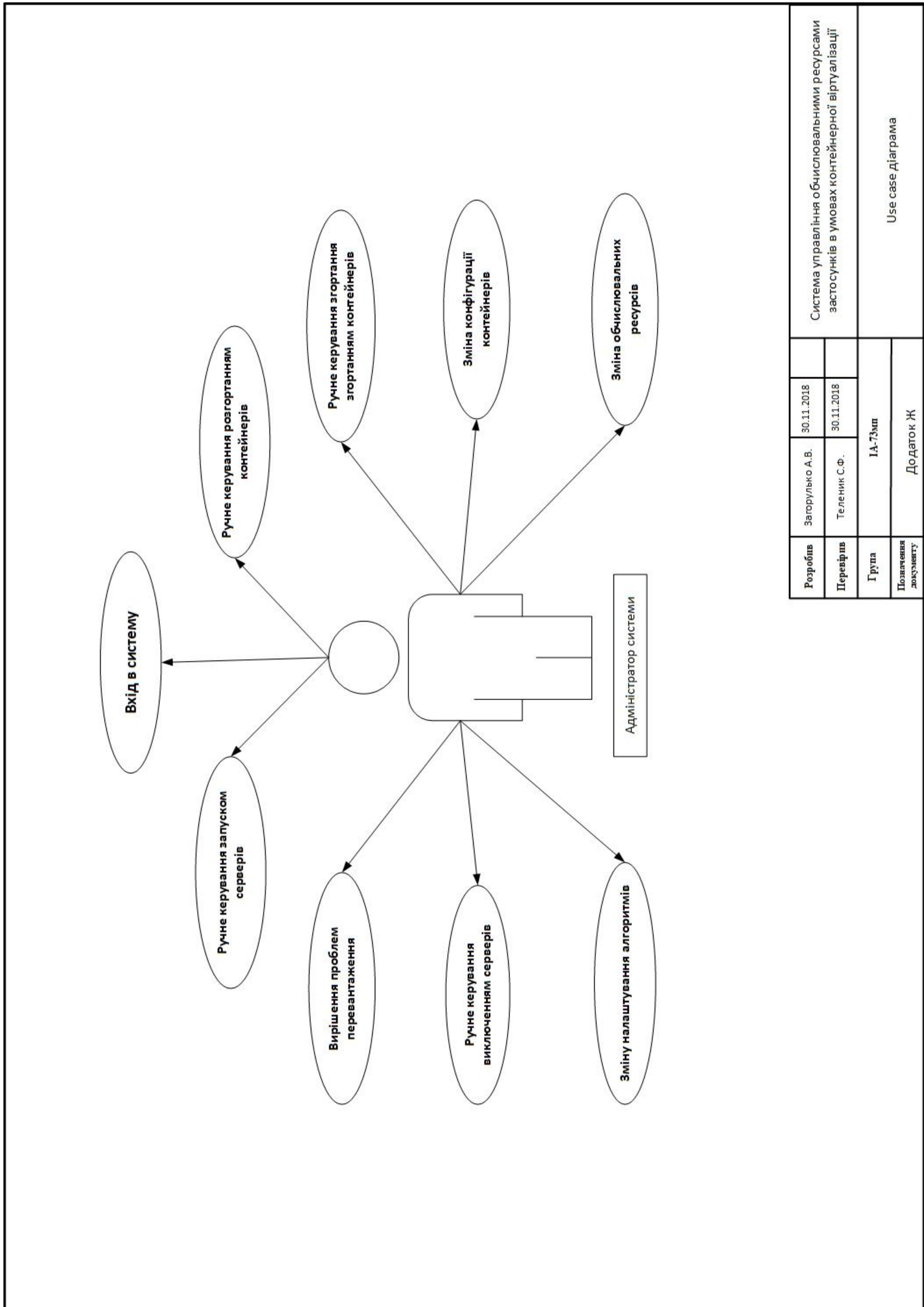
ДОДАТОК Е

Схема алгоритму згортання контейнерів з використанням геометричної прогресії



ДОДАТОК Ж

Use case діаграма



ДОДАТОК И

Структурна схема