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

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих

комп'ютерних систем

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Системне програмування та спеціалізовані комп'ютерні системи»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker»

Виконав:
студент IV курсу, групи КВ-11
Брюханов Олександр Сергійович _____

Керівник:
доцент кафедри СПІСКС, к.т.н., доцент,
Петрашенко Андрій Васильович _____

Консультант з нормоконтролю:
доцент кафедри СПІСКС, к.т.н., доцент,
Клятченко Ярослав Михайлович _____

Рецензент:
доцент кафедри ПЗКС, к.т.н., доцент,
Юрчишин Василь Якович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування і
спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____20__ р.

ЗАВДАННЯ
на дипломний проєкт студента
Брюханова Олександра Сергійовича

1. Тема проєкту «Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker», керівник проєкту доц.каф. СПіСКС, к.т.н., Петрашенко Андрій Васильович, затверджені наказом по університету від 29 травня 2025 р. №1808-С.
2. Термін подання студентом проєкту _____
3. Вихідні дані до проєкту див. ТЗ.
4. Зміст пояснювальної записки:
   - Аналіз існуючих рішень та обґрунтування теми дипломного проєкту;
   - Опис засобів реалізації;
   - Опис розробленого монітору ВМ;
   - Тестування.

5. Перелік графічного матеріалу:
   - Алгоритм роботи менеджеру гіпервізора;
   - Алгоритм монітору процесів гіпервізора;
   - Алгоритм очікування готовності мережі віртуальної машини;
   - Алгоритм монітору оренди IP-адресів;
   - Структура монітору віртуальних машин.

6. Консультанти розділів проєкту:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	доц.каф.СПСКС, к.т.н. Клятченко Ярослав Михайлович		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	15.11.2024	Виконано
2.	Розроблення та узгодження технічного завдання	01.12.2024	Виконано
3.	Аналіз існуючих рішень	05.12.2024	Виконано
4.	Вибір програмних засобів реалізації проєкту	01.03.2025	Виконано
5.	Розробка програмного продукту	07.05.2025	Виконано
6.	Тестування та відлагодження програмного продукту	12.05.2025	Виконано
7.	Написання пояснювальної записки	15.05.2025	Виконано
8.	Підготовка графічної частини дипломного проєкту	20.05.2025	Виконано
9.	Оформлення документації дипломного проєкту	25.05.2025	Виконано
10.	Попередній огляд матеріалів на кафедрі	31.05.2025	Виконано

Студент Олександр БРЮХАНОВ

Керівник Андрій ПЕТРАШЕНКО

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (97 с., 67 рис. 6 табл., 5 додатків).

Об’єкт розробки – програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker.

Програмний комплекс дозволяє: створювати, редагувати та видаляти віртуальні машини; забезпечувати їх роботу; зупиняти віртуальні машини за таймером або за вимогою користувача зі збереженням стану та без; виконувати автоматичну активацію віртуальних машин при надходженні IP-пакету; вести облік витрачених віртуальною машиною ресурсів. Передбачений захист від помилок. В процесі розробки були використано гіпервізор Firecracker, фреймворк Spring Boot, аналітичну базу даних ClickHouse. В якості бази даних використовувалась Oracle Database 23ai.

В ході розробки:

– Проведено аналіз існуючих технологій розподілу обчислювальної потужності;
– Сформульовані вимоги до програмного комплексу розподілу обчислювальної потужності;
– Розроблено архітектуру програмного комплексу;
– Розроблено монітор віртуальних машин на основі гіпервізору Firecracker;
– Підготовлено образи кореневої файлової системи Fedora Linux для запуску у гіпервізорі;

Упровадження цього програмного комплексу дозволить зручне та швидке створення віртуальних машин, що будуть автоматично вивантажуватись із оперативної пам’яті при відсутності мережевого трафіку, що, у свою чергу, дозволить розподіляти ресурси вузлів віртуалізації більш ефективно.

Ключові слова:

віртуалізація, firecracker, oracle database, java, spring boot, netfilter, розподіл обчислювальної потужності, clickhouse, envoy, автоматизація гостингу.

ABSTRACT

The qualification work includes an explanatory note (97 p., 67 fig. 6 tables, 4 appendices).

The object of development is a software system for organising the distribution of cloud computing capacities based on the Firecracker hypervisor.

The software package allows to: create, edit and delete virtual machines; ensure their operation; stop virtual machines by timer or at the request of the user with or without saving the state; automatically activate virtual machines when an IP packet is received; keep track of resources consumed by the virtual machine. Protection from some configuration errors is also included. The Firecracker hypervisor, the Spring Boot framework, and the ClickHouse OLAP server were used in the development process. Oracle Database 23ai was used as the primary database engine.

During the development:

– The analysis of existing technologies of computing power distribution was carried out;
– The requirements for the software system of computing power distribution were formulated;
– The architecture of the software system was developed;
– Virtual machine monitor based on the Firecracker hypervisor was developed;
– Images of the Fedora Linux root file system were prepared to run in the hypervisor;

The implementation of this software package will allow convenient and fast creation of virtual machines that will be automatically unloaded from RAM in the absence of network traffic, which, in turn, will allow allocating the resources of virtualisation nodes more efficiently.

Keywords:

virtualization, firecracker, oracle database, java, spring boot, netfilter, computing power distribution, clickhouse, envoy, hosting automation.

Зм.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість листів	№ прим.	Примітки
	А4	ІАЛЦ.045420.002 ТЗ	Програмний комплекс	5		
			організації розподілу			
			потужностей хмарних			
			обчислень			
			Технічне завдання			
	А4	ІАЛЦ.045420.003 ТП	Програмний комплекс	2		
			організації розподілу			
			потужностей хмарних			
			обчислень			
			Відомість технічного проєкту			
	А4	ІАЛЦ.045420.004 ПЗ	Програмний комплекс	97		
			організації розподілу			
			потужностей хмарних			
			обчислень			
			Пояснювальна записка			
	А4	ІАЛЦ.045420.005 Д1	Алгоритм роботи менеджеру	1		
			гіпервізора			
	А4	ІАЛЦ.045420.006 Д2	Алгоритм монітору процесів	1		
			гіпервізора			
	А4	ІАЛЦ.045420.007 Д3	Алгоритм очікування	1		
			готовності мережі віртуальної			
			машини			

Зм.	*Лист*	*№ докум.*	*Підпис*	*Дата*	

ІАЛЦ. 045490.001 ОА

Розробив	Брюханов О.			Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker	
Перевірив	Петрашенко А.В.				
Н.контр.	Клятченко Я.М.			***Опис альбома***	
Затвердив	Тарасенко В.П.				

Літ.	*Лист*	*Листів*
	1	2

НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-11

Зм.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість листів	№ прим.	Примітки
	А4	ІАЛЦ.045420.008 Д4	Алгоритм роботи монітору	1		
			оренди IP-адресів			
	А4	ІАЛЦ.045420.009 Д5	Структура монітору	1		
			віртуальних машин			
		Диск CD-ROM	Текст пояснювальної записки	1		
			Текст анотації			
			Архівований код програми			
			у форматі .zip			
			Презентація дипломного			
			проєкту			
			Графічний матеріал			

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker».

Галузь застосування: організація ефективного розподілу обчислювальної потужності серверів між клієнтами у середовищі провайдера послуг хмарних обчислень.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проекту є створення програмного комплексу, що складається із монітору віртуальних машин та панелі керування віртуальними машинами та їх мережевими налаштуваннями.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації у періодичних виданнях та електронні статті у мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

- сумісність з операційною системою Red Hat Enterprise Linux 9;
- можливість створення та редагування віртуальних машин;
- можливість об’єднання машин в приватні мережі в межах одного вузла кластеру;
- можливість об’єднувати віртуалізаційні вузли в кластер;
- можливість призначення однієї IP-адреси одній віртуальній машині;
- можливість призначення різних портів однієї IP-адреси різним віртуальним машинам;
- можливість визначення диапазону IP-адресів віртуальних машин в локальних мережах;
- можливість ведення обліку використаних віртуальною машиною ресурсів;
- можливість маршрутизації запитів до віртуальних машин шляхом аналізу HTTP-заголовків;

5.2. Вимоги до апаратного забезпечення

- Процесор: AMD EPYC 3451 або аналог з підтримкою AMD-V;
- Оперативна пам’ять: 16 Гб (вузол з БД), 6 Гб (звичайний вузол);
- Твердотілий накопичувач: 80 Гб, NVMe;
- Операційна система: RHEL-подібний дистрибутив Linux з підтримкою KVM та netfilter;
- Наявність доступу до мережі Internet.

5.3. Вимоги до програмного та апаратного забезпечення користувача

- Браузер: Firefox ESR 128.10.1 або новіший;
- Оперативна пам’ять: 2 Гб;
- Наявність доступу до мережі Internet.

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів
1.	Вивчення літератури за тематикою проекту	15.11.2024
2.	Розроблення та узгодження технічного завдання	01.12.2024
3.	Аналіз існуючих рішень	05.12.2024
4.	Вибір програмних засобів для реалізації продукту	01.03.2025
5.	Розробка програмного продукту	07.05.2025
6.	Тестування та відлагодження програмного продукту	12.05.2025
7.	Написання пояснювальної записки	15.05.2025
8.	Підготовка графічної частини дипломного проекту	20.05.2025
9.	Оформлення документації дипломного проекту	25.05.2025
10.	Попередній огляд матеріалів диплому на кафедрі	31.05.2025

Зм.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість листів	№ прим.	Примітки
	А4	ІАЛЦ.045420.004 ПЗ	Програмний комплекс	97		
			організації розподілу			
			потужностей хмарних			
			обчислень			
			Пояснювальна записка			
	А4	ІАЛЦ.045420.005 Д1	Алгоритм роботи менеджеру	1		
			гіпервізора			
	А4	ІАЛЦ.045420.006 Д2	Алгоритм монітору процесів	1		
			гіпервізора			
	А4	ІАЛЦ.045420.007 Д3	Алгоритм очікування	1		
			готовності мережі віртуальної			
			машини			
	А4	ІАЛЦ.045420.005 Д1	Алгоритм роботи менеджеру	1		
			гіпервізора			
	А4	ІАЛЦ.045420.006 Д2	Алгоритм монітору процесів	1		
			гіпервізора			
	А4	ІАЛЦ.045420.008 Д4	Алгоритм роботи монітору	1		
			оренди IP-адресів			

Зм.	*Лист*	*№ докум.*	*Підпис*	*Дата*			
Розробив		Брюханов О.			Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker	*Літ.*	*Лист* 1 / *Листів* 2
Перевірив		Петрашенко А.В.					
Н.контр.		Клятченко Я.М.			*Опис альбома*	НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-11	
Затвердив		Тарасенко В.П.					

ІАЛЦ. 045490.003 ТП

Зм.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість листів	№ прим.	Примітки
	А4	ІАЛЦ.045420.009 Д5	Структура монітору	1		
			віртуальних машин			
		Диск CD-ROM	Текст пояснювальної записки	1		
			Текст анотації			
			Архівований код програми			
			у форматі .zip			
			Презентація дипломного			
			проєкту			
			Графічний матеріал			

ЗМІСТ

ДОДАТКИ

Додаток 1. Копії графічних матеріалів

- ІАЛЦ.045490.005 Д1. Алгоритм роботи менеджеру гіпервізора
- ІАЛЦ.045490.006 Д2. Алгоритм монітору процесів гіпервізора
- ІАЛЦ.045490.007 Д3. Алгоритм очікування готовності мережі віртуальної машини
- ІАЛЦ.045490.008 Д4. Алгоритм монітору оренди IP-адресів
- ІАЛЦ.045490.009 Д5. Структура монітору віртуальних машин

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ОС – операційна система

ВМ – віртуальна машина

ІТ – інформаційні технології

FTP – File Transfer Protocol, протокол для керування та завантаження файлів

DNS – Domain Name System, система домених імен

SQL – Structured Query Language, мова структурованих запитів для роботі із базою даних

CGI – Common Gateway Interface, проткол взаємодії програмного забезпечення із веб-сервером

LSAPI – протокол взаємодії програмного забезпечення із веб-сервером, специфічний для серверу LiteSpeed та модулів CloudLinux для nginx

FastCGI – протокол, схожий на CGI, що дозволяє програмам продовжувати виконання після кінця життєвого циклу запиту

Shared-гостинг – послуга розміщення веб-сайту або веб-застосунку на одному сервері з іншими застосунками інших клієнтів

Віртуалізація – технологія створення штучного, зазвичай ізольованого від інших, середовища в якому можна виконувати обчислення

Контейнеризація – підхід, що використовує механізми ядра ОС для ізоляції на рівні окремих процесів

FaaS – Function-as-a-Service, послуги з надання обчислювальних потужностей для stateless-мікросервісів, що запускаються, зазвичай в контейнері, при отриманні запиту ззовні

Stateful – модель обчислень із збереженням внутрішнього стану між запитами

Stateless – модель обчислень без збереження внутрішнього стану між запитами

Firecracker – гіпервізор 2 типу від Amazon

QEMU – гіпервізор 2 типу з відкритим кодом

MicroVM – модель роботи гіпервізора із використанням паравіртуалізації та зменшеної кількості пристроїв, що використовується з метою швидкого запуску ВМ

Кластер – група з'єднаних логічно серверів, що можна використовувати як єдиний ресурс

CGroups – механізм ядра Linux для встановлення обмежень на використання ресурсів комп'ютера по процесам

EXT4 – 4th eXtended File System, журнальна файлова система

XFS – журналюєма файлова система, створена для великих накопичувачів

Overselling – практика продажу послуг у обсягах, що перевищують фактичні можливості їх надання

CloudLinux – дистрибутив Red Hat Enterprise Linux, створенний для використання на вузлах shared-гостингу

Псевдопристрій – пристрій, що не існує фізично, але взаємодіяти з яким можна через драйвер

Квота – визначена максимально доступна частка ресурсу серверу

HTTP – HyperText Transfer Protocol

DHCP – Dynamic Host Configuration Protocol

BTRFS – Binary Tree File System

ARP – Address Resoultion Protocol

MVC – Model View Controller

SNI – Server Name Indication

TLS – Transport Layer Security

DTO – Data Transfer Object

API – прикладний програмний інтерфейс

TAP – віртувальна точка доступу

ВСТУП

В сучасному світі, де питання присутності в Інтернеті стає все більш актуальним, як для бізнесу, так і для індивідуальних осіб, стає все важче уявити життя без хмарних технологій. Самостійна підтримка комп'ютерної інфраструктури зазвичай вимагає велику кількість витрат, що спонукає людей та компанії, замість володіння своїм обладнанням, звертатися за послугами до інфраструктурних провайдерів.

Оренда обчислювальної потужності економить час за рахунок відсутності необхідності займатись обслуговуванням обладнання та програмного забезпечення, бо цим будуть займатись інженери провайдера. Така модель також дозволяє зменшити і грошові витрати, адже провайдер може обслуговувати одразу велику кількість проєктів невеликою командою працівників, що спеціалізується на підтримці роботи інфраструктури та вже має налагоджені процеси роботи, за рахунок чого пропонувати зручний сервіс за меншу ціну.

Такий ефект масштабу також поширюється і на вартість обчислювальної потужності: можливість динамічно розподіляти ресурси дозволяє пропонувати недорогі послуги більшій кількості потенціальних клієнтів, при цьому підвищуючі рентабельність за рахунок зменшення простою обладнання. У наш час, коли навіть малий бізнес, що не пов'язаний з IT, має потребу у використанні комп'ютерних технологій, це є дуже актуальним, адже пропозиції від хмарних провайдерів є більш фінансово привабливими та пропонують більшу стабільність, не потребуючи при цьому великих стартових інвестицій [1].

Задача розподілу обчислювальної потужності при цьому не є новою. Описані вище переваги для малого бізнесу були актуальні ще у 20 столітті, а отже і моделей розподілу існує багато. Сьогодні для вирішення цієї задачі використовується віртуалізація, контейнеризація та, у деяких випадках,

провайдери можуть розмістити на одному сервері декілька клієнтів і обмежувати їх на рівні ОС та модулів програмного забезпечення.

Остання згадана модель розподілу є найбільш дешевою, адже покладаючись на механізми ОС, можна досягти великої швидкодії при низькому використанні ресурсів вузла за умови відсутності запитів, що дозволяє розмістити більше клієнтів на одному вузлі. Проте через відсутність гнучкості, такий підхід є дещо нішевим та використовується здебільшого у shared-гостингах. Схожа модель використовується FaaS-провайдерами, проте там замість механізмів ОС використовується контейнерізація або гіпервізори нового покоління, як от Firecracker чи QEMU в режимі роботи MicroVM.

Ця дипломна робота покликана дослідити можливість та потенційні переваги використання гіпервізору Firecracker для організації розподілу потужностей для stateful застосувань загального призначення, де зараз використовуються класичні гіпервізори.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЄКТУ

1.1. Огляд існуючих рішень

З метою обгрунтування теми цієї дипломної роботи, в цьому розділі буде розглянуто декілька сучасних продуктів, що використовуються для розподілу потужності одного серверу або кластеру з декількох. Список розглягутих продуктів не є вичерпним, але тим не менш є достатнім для того, щоб можна було обговорити існуючі рішення, їх переваги, особливості та недоліки.

1.2. Shared-гостинги. DirectAdmin та CloudLinux

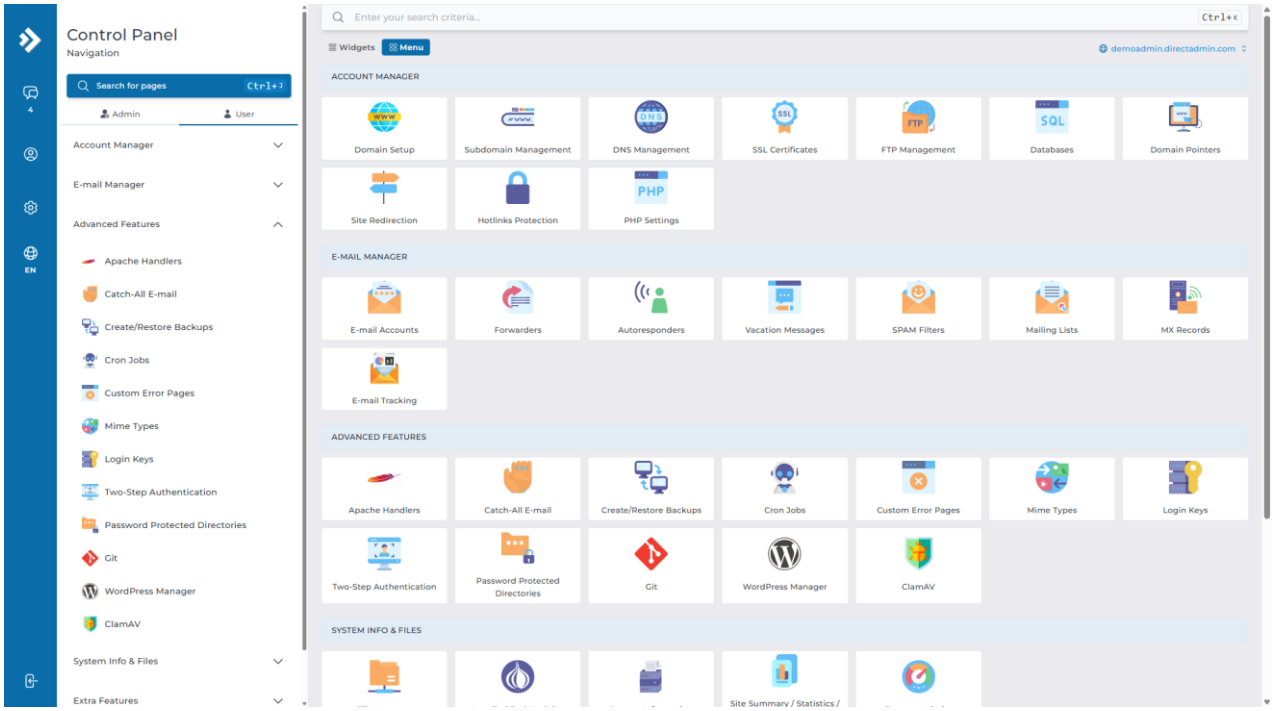


Рисунок 1.2.1 – Панель керування DirectAdmin

DirectAdmin – одне із популярних рішень для організації роботи shared-гостингу. За інформацією компанії з маркетингової аналітики 6sense, DirectAdmin належить доля ринку панелей керування веб-гостингом розміром

приблизно 5% [2]. Цей продукт має типові для своєї категорії функції, зокрема [3]:

- Керування базами даних MySQL
- Керування файлами та FTP-акаунтами
- Керування DNS-записами
- Керування акаунтами електронної пошти
- Керування резервним копіюванням
- Керування налаштуваннями PHP
- Перегляд статистики
- Створення реселерів
- Створення
- Кластерізація DNS

Розподіл обчислювальної потужності DirectAdmin виконує за допомогою механізму CGroups v2, що входить до ядра Linux [4] та дозволяє обмежити використання ресурсів серверу різними процесами. Адміністратор гостингу може створювати шаблони-тарифи, в яких допускається налаштування квоти на використання процесору, пам'яті та швидкості виконання операцій вводу-виводу. Також можна вказати ліміт на кількість процесів, що запускаються користувачем.

Для обмеження використання місця на диску використовується механізм квот, доступний на файлових системах ext4 та XFS [5]. Такий підхід дозволяє зробити т.н. overselling та створити на одному кластері користувачів, сума дозволеного використання диску яких буде більше за фактичний об’єм доступного місця на вузлах кластеру, адже використання квот не передбачає обов’язкове виділення місця на диску у повному обсязі.

DirectAdmin дозволяє користувачам створювати окремі налаштування роботи PHP [6]. Наявна підтримка декількох режимів роботи PHP, наприклад через CGI, FastCGI або за допомогою імплементації LSAPI від CloudLinux. Усі

режими роботи запускають процеси від імені користувача, тому вони автоматично обмежуються згідно із квотами групи, до якої належить користувач.

Використання серверу баз даних в DirectAdmin за умовчанням є доволі примітивним: файл бази даних зберігається у директорії користувача, а тому підпадає під обмеження на об'єм диску, але використання пам'яті та процесору SQL-запитами користувачів не відбувається. Даний функціонал не імплементований у рушіях баз даних MySQL, але його можна додати у середовищі CloudLinux за допомогою модулю SQL Governor, що дозволяє додати квоти на використання процесорного та звичайного часу [7]. Такий самий механізм використовується і в інших панелях керування, як от cPanel [8].

Загалом, використання DirectAdmin чи інших панелей керування подібного типу є ефективним рішенням, яке дозволяє розподілити ресурсу таким чином, щоб вони використовувались тільки коли вони насправді потрібні. В таких середовищах використовується спільний сервер баз даних, процеси запускаються тільки за потреби та місце на диску виділяється поступово. Це все дозволяє розмістити велику кількість користувачів і за рахунок масштабу зменшити вартість послуг.

Варто зазначити, що такі рішення не є гнучкими та загалом є більш вразливими до атак. Хоча у середовищі DirectAdmin можна дозволити запускати не тільки PHP та Perl сценарії, а будь-які програми, їх робота не за протоколом HTTP не передбачена, а використання спільного середовища не дозволить користувачам встановити свої бібліотеки або програми через пакетний менеджер. До того ж, успішне використання вразливостях в ядрі ОС або сервера баз даних призведе до компроментації даних всіх користувачів вузла.

Отже, подібні рішення є нішевими та ситуація з безпекою їх використання не є найкращою.

1.3. Віртуалізація. VMware ESXi

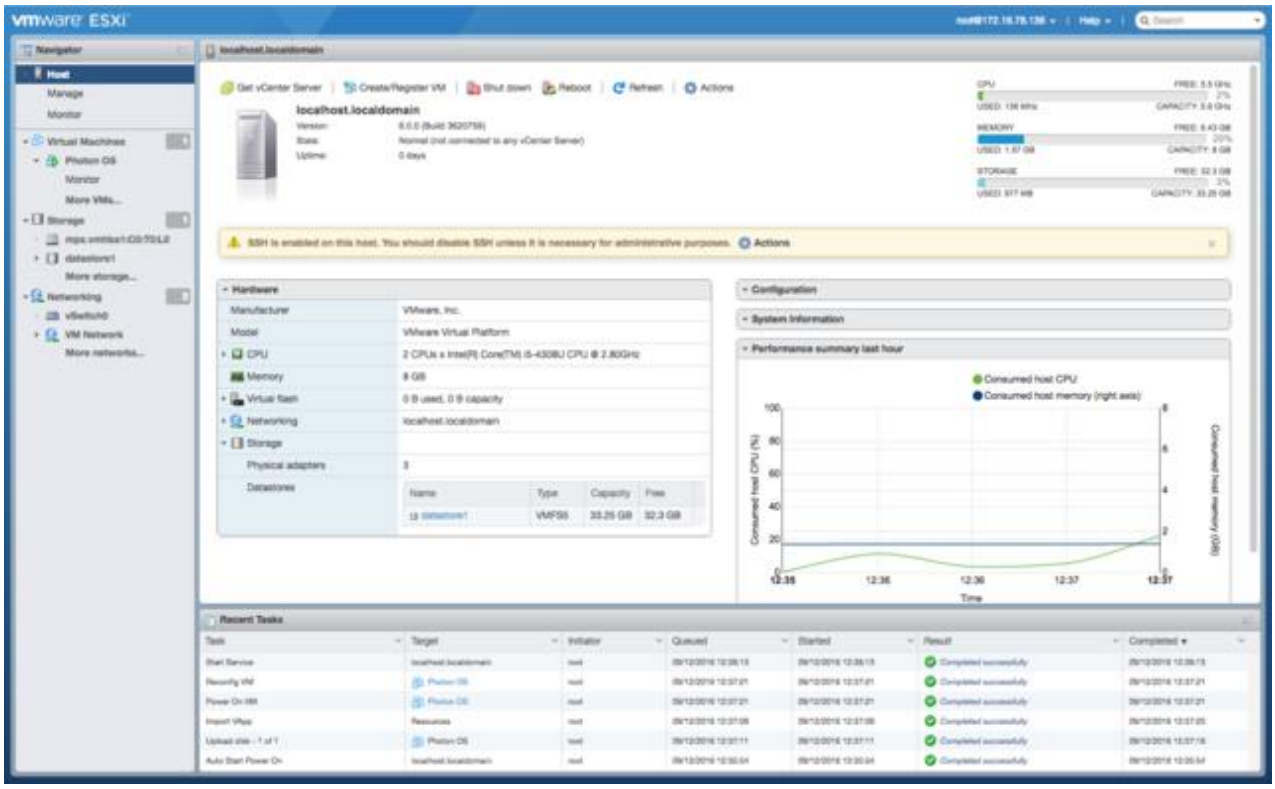


Рисунок 1.3.1 – Панель керування VMware ESXi

VMware ESXi – гіпервізор першого типу, що використовується для створення та керування віртуальними машинами. На відміну від DirectAdmin, в цьому продукті ізоляція досягається за допомогою віртуалізації. Це дозволяє кожному користувачу мати під своїм контролем більшу частину програмного забезпечення, від ядра ОС до сервісів, що будуть виконуватись на ВМ.

Такий підхід є значно більш гнучким, але разом із тим є не завжди ефективним. Зокрема, віртуальні машини запускаються значно повільніше ніж процеси. Це відбувається через те, що при запуску віртуальної машини треба запустити і ОС на ній, а це займає певний час, як і в самій ОС на етапі активації служб, так і в гіпервізорі під час налаштування віртуалізовних пристроїв.

Це компенсується тим, що більшість ВМ зазвичай не зупиняють після запуску, але це призводить до того, що деякі ресурси, виділені для використання певною ВМ будуть недоступні іншим. Розподіл потужності

процесору виконується доволі легко, але механізми розподілу оперативної пам’яті та диску є менш надійними.

Для оптимізації розподілу оперативної пам’яті у VMware використовується Memory Balooning. Цей підхід передбачає встановлення драйверу у віртуальні машини, що буде слідкувати за використанням пам’яті та, у випадку коли віртуальна машина не використовує всю виділену пам’ять, резервувати її. Таким чином гіпервізор може перерозподіляти фізичну пам’ять між віртуальними машинами. Проте для роботи цього механізму потрібна кооперація зі сторони програмного забезпечення, що встановлене у ВМ. Якщо користувач видалить або модифікує драйвер, він може змусити гіпервізор не забирати у ВМ пам’ять. Це універсальне обмеження такого підходу і не є специфічним для гіпервізору ESXi [9].

Для оптимізації роботи з диском використовується Thin Provisioning, що дозволяє виділяти місце на диску поступово, але такий підхід теж дуже залежить від поведінки ОС, що встановлена на ВМ, зокрема від файлової системи. Хоча це і є гарною оптимізацією, яка за умовчанням увімкнена в ESXi, вона не настільки надійна щоб її можна було розглядати в контексті зменшення вартості.

Описані вище проблеми змушують резервувати ресурси для кожної ВМ, навіть якщо програми, що запущені всередені їх не потребують, що негативно впливає на вартість послуг. Не зважаючи на великий ступінь гнучкості, для навантажень змінного характеру підхід віртуалізації може бути не самим оптимальним.

Деякі провайдери надають знижки за використання ВМ, які гіпервізор може вимкнути у випадку потреби обчилювальної потужності, але в такому випадку можливість запуску ВМ у будь-який момент часу не гарантується і, після зупинки, такі ВМ автоматично не перезапускаються [10].

1.4. Контейнеризація. OpenShift

OpenShift Container Platform – сертифікований дистрібутив Kubernetes від компанії Red Hat [11], що дозволяє запускати контейнери. На контейнери можна накласти квоти по використанню процесора, оперативної пам’яті та диску [12]. Також контейнери можна згрупувати по просторам імен (namespaces), на які теж можна накласти квоти [13].

Контейнеризація працює через механізми ядра CGroups та Network Namespaces. В контейнерах можна запустити як і цілком образ певної ОС, так і просто застосунок. У першому випадку, запуск контейнерів буде повільніший через причини, що були описані у секції про віртуалізацію.

При такому підході за кожним користувачем зберігається контроль над програмним забезпеченням, що працює у контейнері, проте є певні обмеження. По-перше, процеси в контейнерах запускаються тим же самим ядром, що працює на хості контейнеризації, а отже контролю над ядром та його модулями у користувача немає. По-друге, у контейнерах відсутній доступ до привілейованих пристроїв, що впливає на доступний функціонал ядра у контейнері. Зокрема, у контейнерах не можна створювати віртуальні машини або запускати інші контейнери безпечним чином, на відміну від ВМ, в яких, за умови коректного налаштування гіпервізору, можна без особливих труднощів запускати, хоч ВМ, хоч контейнери.

Подібні проблеми, хоч і не є суттєвими, обмежують використання контейнеризації для старих чи незвичайних застосунків. Але при цьому контейнеризація є більш гнучкою, ніж середовища на shared-гостингах і більш безпечною, адже підміна кореневої директорії під час запуску процесів є додатковим шаром ізоляції.

Загалом, контейнеризація є більш ефективною у плані розподілу ресурсів ніж віртуалізація, проте необхідність адаптації застосунків для роботи в контейнері треба враховувати.

1.5. Function-as-a-Service. Knative

Knative – розширення платформи оркестрації контейнерів Kubernetes [14], що дозволяє динамічно запускати контейнери для реагування на зовнішні події.

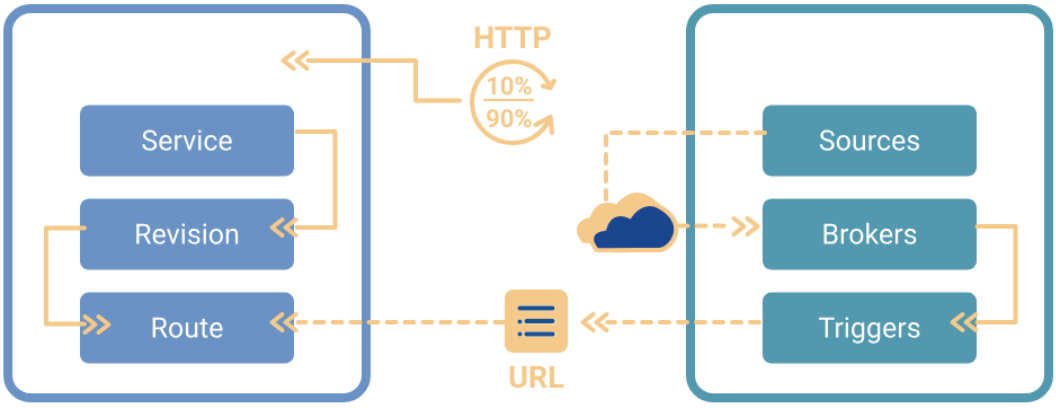


Рисунок 1.5.1 – Діаграма роботи Knative

Цьому підходу притаманні проблеми, специфічні для контейнеризації, зокрема:

- Необхідність адаптувати код або конфігурацію застосунків для роботи в контейнерному середовищі
- Обмеження в використанні деяких модулів ядра
- Обмеження в використанні псевдо- та звичайних пристроїв

Також Knative використовується здебільшого для stateless-контейнерів, тобто таких, що не зберігають свій стан між запитами. Зазвичай це вирішується тим, що контейнери-обробники запитів використовують спільну базу даних, з якої береться інформація про стан системи.

Але, звісно, не всі застосунки можна адаптувати до такого підходу. Зокрема, такий підхід не підійде для серверів розважальних застосунків, що мають працювати у реальному часі, але при цьому він чудово підійде для застосунків з мікросервісною архітектурою та простими обробниками викликів API.

На відміну від простого Kubernetes або OpenShift, розподіл ресурсів при навантаженнях змінного характеру є більш оптимальним, адже контейнери запускаються тільки коли для них надходять запити.

1.6. OpenShift із CRI-O та Kata Containers

Для вирішення проблеми, пов’язаної із тим, що контейнеризація накладає обмеження на використання модулів ядра можна замінити рушій контейнеризації в OpenShift або Kubernetes з Docker на CRI-O. Це дасть змогу використовувати будь-яку сумісну систему виконання контейнерів [15], зокрема Kata Containers.

Система виконання контейнерів Kata насправді не використовує контейнеризацію, а натомість створює мікро-ВМ за допомогою гіпервізорів Firecracker, QEMU або Cloud-Hypervisor [16].

Програмне забезпечення все одно потрібно буде адаптувати для роботи із контейнерами, проте якщо воно вже працює з OpenShift або Knative, зазвичай зміни не будуть потрібними.

Варто зазначити, що при використанні Knative або OpenFaaS разом із Kata Containers, застосунки мають бути stateless, адже ці системи будуть вимикати ВМ та запускати її знову при надходженні запита. Хоча гіпервізори QEMU та Firecracker підтримують збереження стану ВМ із вивантаженням на диск, цей функціонал не можна використати через Kata Containers [17]. При цьому тимчасова зупника ВМ без вивантаження оперативної пам’яті на диск підтримується.

1.7. Обгрунтування теми дипломного проєкту

Як можна побачити, задача розподілу обчислювальної потужності є доволі актуальною і способи її вирішення продовжують розроблятись і в наш час. Найбільшого розвитку зазнали системи контейнеризації, які вже в деяких випадках почали переходити з класичного підходу контейнеризації до

віртуалізації за допомогою гіпервізорів, що оптимізовані для масштабування у хмарному середовищі.

Яскравим прикладом таких гіпервізорів може послужити Cloud-Hypervisor чи Firecracker, що використовуються у Kata Containers. Разом із такими системами як Knative чи OpenFaaS їх можна використовувати у stateless-системах, побудованих за принципом Serverless.

Проте ці гіпервізори не обов’язково використовувати разом із Kata, вони можуть бути вбудованими в інші системи, що використовуватимуть їх функціонал ширше. На разі немає такої системи, яка могла б подібно до OpenFaaS запускати ВМ по зовнішній події та яка б підійшла для не stateless-систем.

Враховуючи, що не всі програми можуть бути адаптовані для роботи за принципом Serverless та не в кожному разі часовитратне налаштування контейнеризації буде доречним, розробка програмного комплексу, що буде запускати ВМ із повною операційною системою загального призначення за наявності мережевого трафіку є актуальною задачею. Вирішення цієї задачі якраз і є метою цього дипломного проєкту.

.

2. ОПИС ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Обґрунтування вибору мови програмування

Велика кількість сучасних мов програмування є мовами загального призначення, які можуть бути застосованими для написання будь-якого програмного забезпечння. Проте навіть універсальні мови можуть суттєво відрізнятися одна від одної, адже мову визначає не тільки синтаксис та парадигми, а ще і єкосистема, зокрема доступні інструменти та бібліотеки, активна спільнота та кількість якісної документації, наявність потрібних реалізацій компілятору чи інтерпретатору та їх особливості. А отже, вибір правильної мови програмування дуже залежить від потреб проєкту та є одним із найважливіших технічних рішень.

Розробка монітору віртуальних машин, що працює над існуючим гіпервізором не потребує використання спеціалізованої мови програмування, при цьому варто сформулювати перелік вимог та на його основі обрати найбільш підходящу мову. Враховуючи технічне завдання, найкращою мовою буде така, що зможе забезпечити високу швидкість роботи, можливість зручної інтеграції з низькорівневими бібліотеками, але при цьому буде відмовостійкою та сприятиме легкому написанню коректного коду.

Популярними в контексті віртуалізації мовами є C та C++, вони використовуються як для написання низькорівневих компонентів (гіпервізори), так і для більш високорівневих (зовнішні інтерфейси для налаштування, панелі керування тощо). Такі рішення, як Cockpit, VMware ESXi, VMmanager написані здебільшого на C/C++, інструменти, що забезпечують їх роботу (QEMU, libvirt) також. Сучасні компілятори цих мов враховують особливості процесорів, через що здатні генерувати дуже ефективний машинний код. Бібліотеки для них розроблені таким чином, щоб абстракції не накладали додаткових витрат під час виконання. Компоненти

більшості ОС також написані на цих мовах або принаймні експортують для них прямі інтерфейси.

Отже, ці мови є максимально швидкими та добре інтегруються з компонентами системи та низькорівневими бібліотеками. Варто зазначити, шо керування пам’яттю в C та C++ є ручним, що забезпечує додаткову гнучкість, але при цьому накладає на розробника відповідальність за правильне її використання. Як показує практика, такий підхід призводить до виникнення помилок, які можуть суттєво вплинути на стабільність та безпеку програмного забезпечення. Зокрема, більшість вразливостей, знайдених проєктом OSS-Fuzz, були пов’язані з пам’яттю [19], а державне агенство з кібербезпеки США рекомендує не використовувати C, C++ та інші мови з ручним керуванням пам’яттю для об’єктів критичної інфраструктури [20].

Оскільки помилки в моніторі віртуальних машин можуть призвести до часткового чи повного пошкодження даних користувачів, в рамках цього дипломного проєкту було прийнято рішення не використовувати жодну з цих мов в якості основної. При цьому деякі компоненти програми, що працюють з мережею, потребують високої швидкості та тісної інтеграції з компонентом ядра netfilter, через що вони і були частково реалізовані мовою C.

В якості основної мови програмування було обрано Java. Програми, написані цією мовою компілюються у байт-код, який потім виконується інтерпретатором. Інтерпретатор перетворює байт-код на машинний під час виконання програми та застосовує різні оптимізації в залежності від того, як поводиться програма. У бенчмарках Java демонструє високу швидкість, витрачаючи на виконання на 40% більше часу, ніж аналогічні програми, написані мовою C [21].

Згаданий вище бенчмарк порівнює швидкодію різних мов програмування на прикладі простої програми, основна складність якої полягає у великій кількості математичних операцій. Такі порівняння зазвичай не використовуються при виборі мови програмування, адже по ним можна лише

оцінити накладні витрати інтерпретації чи ефективність роботи компілятору, коли більш впливовим чинником є архітектура застосунку, яка часто випливає із підходів, що є ідіоматичними для певної мови. Наприклад, орієнтованість таких мов, як Java та C# на ООП призводить до створення великої кількості об'єктів, що на практиці призводить до втрати швидкості. У цьому конкретному випадку результати бенчмарку є корисними, бо в програмі будуть також присутні прості, слабко об'єктно-орієнтовані компоненти на швидкодію яких здебільшого впливатиме саме оверхед інтерпретатору.

Для Java також доступна велика кількість інструментів розробки: від інтегрованих середовищ до фреймворків. В цій роботі використовується фреймворк Spring Boot, який є надійним та модульним, що дозволяє швидко додати функціонал для роботи з базою даних, користувачами, валідаторами даних та WebSockets. Для інтеграції з компонентами, написаними на C використовується Project Panama, що входить у стандартну бібліотеку Java.

З метою повноти аналізу також варто розглянути такі мови як Rust, PHP та C#/F#. Rust – це компільована мова, що за швидкістю дуже близька до C/C++ і має автоматичне керування пам'яттю, що не потребує збірки мусору (GC), адже працює на етапі компіляції. Коректність роботи з пам'яттю перевіряється за допомогою потужної системи типів. При цьому екосистема Rust ще не настільки розвинута і не може надати стабільний та зручний інструментарій, який є у Java.

Схожі інструменти наявні у PHP, де аналогом фреймворку Spring може послужити Symfony, а функціонал Project Panama доступний через розширення FFI. В PHP також є і JIT-компіляція, але основна реалізація PHP є не дуже ефективною і навіть з JIT показує погану швидкодію. Необхідність створення потоків та процесів у застосунку також унеможливлює використання PHP, адже навіть потоко-безпечні збірки інтерпретатору цієї мови схильні до проблем в роботі з пам'яттю та синхронізацією [22].

Найбільш близькою альтернативою Java може бути C# або інша CLR-мова. Інтерпретатор платформи .NET Core показує швидкодію трохи краще ніж Java (на 30% повільніше C), має інструменти для інтеграції з низькорівневими бібліотеками, споживає менше оперативної пам’яті. І Java, і C# мають автоматичне керування пам’яттю і потребують для цього GC, під час роботи якого виконання програми може бути призупинено. В C# можна обрати серверний варіант GC та увімкнути багатопоточний режим [23], що дозволить зменшити вірогідність призупинення програми. Різні інтерпретатори Java мають різні політики роботи GC. Зокрема, OpenJ9 має політику “metronome”, що дозволяє не тільки зменшити вірогідність призупинення, а ще і задати максимальний час, який інтерпретатор може використати для збірки мусору [24]. Також OpenJ9 більш оптимально працює з пам’яттю і може в деяких випадках наблизитись до показників, типових для C#.

Отже, з урахуванням наведеного вище аналізу мов програмування, можна зробити висновок, що найбільш оптимальним варіантом для цього дипломного проєкту буде написання застосунку мовою програмування Java з використанням мови C для деяких компонентів. В якості інтерпретатора Java, з метою оптимізації витрат пам’яті та збільшення швидкодії програми, буде використано IBM Semeru з OpenJ9, налаштованим на використання політики metronome для збірки мусору.

2.2. Обґрунтування вибору сховища конфігурації

При розробці серверних застосунків, вибір підходу до зберігання конфігурації є критично важливим рішенням, яке може суттєво вплинути на зручність обслуговування, масштабованість та надійність системи. Не залежно від складності, обране сховище конфігурації впливатиме на те, наскільки легко буде оновлювати, переносити, перевіряти та захищати налаштування. Невдалий вибір може призвести до проблем із розгортанням та експлуатацією

програмного забезпечення, в той час як влучно обраний варіант може підвищити надійність та полегшити роботу.

В цьому розділі розглядається зберігання даних у конфігураційних файлах, у розподілених сховищах типу «ключ-значення», а також у деяких реляційних системах керування базами даних.

2.2.1. Текстові конфігураційні файли

Одним із найрозповсюджених підходів до зберігання налаштувань програми є текстові конфігураційні файли. Вони зберігаються на диску у форматі, який може легко прочитати або відредагувати людина без спеціалізовних інструментів. Зазвичай для цього використовуються XML, JSON, YAML чи інші формати серіалізації загального призначення. Але цей підхід є доволі гнучким і, якщо дозволяє ситуація, можна реалізувати власну предметно-орієнтовану мову конфігурації, як це роблять такі програми як nginx, HAProxy, Apache HTTPd.

Зчитування з диску та розбір змісту є дуже швидким у сценаріях, коли файл знаходиться на локальній машині. Але перевірку правильності конфігурації треба виконувати самостійно. У випадках, коли коректність налаштувань певного вузла залежить від налаштувань чи стану на інших, треба організовувати власні методи синхронізації. Централізована зміна чи читання даних з декількох вузлів одночасно також при такому підході може бути ускладнена.

2.2.2. Розподілені сховища типу «ключ-значення»

В якості сховища даних в розподілених системах можна використати такі бази даних, як Apache ZooKeeper чи etcd. Це БД, що можуть зберігати деревовидні структури, доступ до елементів яких відбуваєтья за ключем або ланцюжком ключів, як це, наприклад, робиться у файлових системах [26].

До переваг таких сховищ можна віднести те, що вони швидкі, дуже добре масштабуються та беруть на себе відопвідальність за розповсюдження змін по вузлам системи, а отже синхронізувати стан між ними розробнику не

треба, достатньо лише підписатись на оновлення на стороні застосунку та обробляти зміни зручним способом.

При цьому перевірки коректності на стороні цих сховищ немає, що ускладнює розробку, особливо якщо є багато налаштувань, що впливають один на одного.

2.2.3. Реляційні бази даних

Конфігурацію можна також зберігати в реляційних базах даних. На відміну від сховищ типу «ключ-значення», дані в них зберігаються по завчасно створеній схемі, а за допомогою індексів, зовнішніх ключів та обмежень (check constraints) можна забезпечити перевірку коректності налаштувань на стороні СКБД та попередити збереження невалідної конфігурації.

Підписатись на зміни в таких базах даних не можна, тому синхронізовувати конфігурацію треба самостійно. Декілька вузлів кластеру при цьому можуть спільно використовувати один сервер БД.

2.2.4. Порівняння підходів

Результати проведеного вище аналізу наведені у вигляді таблиці 2.1.

Таблиця 2.1 – Порівняння підходів

Критерій	Підхід		
	Текстові файли	Розподілені БД «ключ-значення»	Реляційні бази даних
Швидкість при локальному доступі	Дуже висока	Висока	Висока
Швидкість у розподілених системах	Низька	Висока	Висока
Гнучкість	Дуже висока	Низька	Середня

Зручність ручного редагування	Дуже висока	Низька	Низька

Продовження таблиці 2.1

Зручність синхронізації між вузлами	Дуже низька	Дуже висока	Середня
Зручність перевірки коректності налаштувань	Низька	Дуже низька	Дуже висока
Накладні витрати	Відсутні	Низькі	Низькі, але сервер БД може споживати велику к-сть ресурсів
Зручність синхронізації між вузлами	Дуже низька	Дуже висока	Середня

Оскільки в цьому дипломному проєкті конфігурація містить велику кількість взаємозалежних об’єктів (наявні відношення між IP-адресами та віртуальними машинами, між портами та IP-адресами, між портами та правилами пересилки тощо), які при цьому здебільшого є локальними для кожного вузла, доцільно обрати такий підхід, який може надати високі гарантії коректності, зручністю синхронізації між вузлами можна знехтувати, поки наявна можливість централізовано отримувати актуальну конфігурацію всіх вузлів.

Керуючись таблицею 2.1, можна зробити висновок, що найкращим сховищем конфігурації у цьому випадку буде реляційна база даних. Для виибору СКБД потрібно проаналізувати існуючі СКБД, такі як MySQL, Microsoft SQL Server та Oracle Database.

2.2.5. MySQL

MySQL – вільне програмне забезпечення для керування реляційними базами даних. Здебільшого використовується на shared-гостингах. Ця СКБД була створена як альтернатива комерційним системам, має при цьому задовільний набір функціоналу, хоча поступається пропрієтарним СКБД та деяким вільним як PostgreSQL, зокрема у питаннях типізації. Наявна можливість отримати підтримку від розробника у рамках пропозиції MySQL Enterprise.

2.2.6. Microsoft SQL Server

Microsoft SQL Server – пропрієтарна СКБД, вона є швидшою за MySQL та має більш просунуту систему типізації. Ця база даних комерційна, проте для розробки її можна використовувати безкоштовно, також наявна обмежена версія MS SQL Express, що не потребує покупки ліцензії. Сучасні версії цієї СКБД доступні не тільки під Windows, а ще і під Linux.

2.2.7. Oracle Database

Oracle Database – пропрієтарна СКБД, що має схожі з Microsoft SQL Server характеристики, є кросплатформеною та має офіційні збірки для дистрибутивів на базі Red Hat Enterprise Linux. Окремої версії для розробників ця СКБД немає, проте бази даних обсягом менше 12 ГБ можна зберігати у безкоштовній версії Oracle Database, що обмежена лише за обсягом даних, оперативної пам’яті та процесорних ядер.

2.2.8. Порівняння швидкодії СКБД

У тестах щвидкості СКБД, Oracle Database показує себе найркащим чином у наступних сценаріях [27]:

- Вибірка даних з сортуванням

- Вибірка даних з фільтрацією

Найгіршим чином Oracle Database себе показує у сценаріях, де використовуються підзапити. Запити з об’єднаннями Oracle Database виконує значно швидше ніж MySQL, але сильно потсупається Microsoft SQL Server.

2.2.9. Висновок

Результати порівняльного аналізу різних СКБД наведені у таблиці 2.2.

Таблиця 2.2 – Порівняння СКБД

Критерій	СКБД		
	MySQL	Microsoft SQL Server	Oracle Database
Швидкість виконання запитів з сортуванням	Повільна	Повільна	Дуже швидка
Швидкість виконання запитів з фільтрацією	Повільна	Середня	Дуже швидка
Швидкість виконання запитів з підзапитами	Середня	Швидка	Повільна
Швидкість виконання запитів з об’єднаннями	Дуже повільна	Дуже швидка	Середня
Наявність	Так	Так	Так

безкоштовної версії			

Продовження таблиці 2.2

Обмеження безкоштовної версії	Відсутні	Для розробників обмеження відсутні; розмір БД до 10 ГБ, викорситання пам’яті до 1.4 ГБ, використання до 4 ядер процесору	Розмір БД до 12 ГБ, використання пам'яті до 2 ГБ, використання до 2 ядер процесору
Наявність платної підтримки від розробника	Так	Так	Так

Оскільки в проєкті буде виконуватись багато запитів до бази даних із фільтруванням з метою отримання конфігурації об’єктів, що є локальними для вузла, та будуть наявні запити з об’єднаннями, було прийнято рішення використати СКБД Oracle Database. Ця СКБД поступається швидкістю виконання запитів з об’єднаннями, проте має більше доступної для індексів пам’яті у безкоштовному виданні.

2.3. Обґрунтування вибору гіпервізору

Оскільки основною задачею монітору віртуальних машин є розподіл обчислювальних ресурсів серверу між віртуальними машинами за допомогою гіпервізору, вибір правильного гіпервізору є ще одним важливим технічним рішенням. За технічним завданням, програмний комплекс має автоматично призупиняти та відновлювати роботу віртуальних машин при наявності чи

зникненні мережевого трафіку таким чином, щоб зменшити кількість ситуацій, у яких затримка буде дуже помітною для зовнішнього користувача.

Хоча зменшити швидкість запуску ОС, обмежуючись лише засобами гіпервізору, складно, можна обрати такий гіпервізор, який може забезпечити швидку передачу контролю ядру ОС після запуску та швидке відновлення роботи ВМ після призупинення.

Наразі існують два гіпервізори, що задовольняють таким вимогам, а саме Firecracker та QEMU. В стандартній конфігурації QEMU більш схожий на класичні гіпервізори, але має окремий режим роботи під назвою MicroVM, в якому кількість емульованих пристроїв значно менша, а отже і запуск відбувається віртуальної машини відбувається швидше. В такому режимі роботи відсутня можливість динамічного підключення пристроїв під час роботи ВМ та не підтримуються пристрої, що працюють через шину PCI. Для підключення до мережі використовується virtio-net, а для роботи з диском virtio-blk.

Режим роботи MicroVM багато в чому схожий на те, як працює Firecracker і загалом віртуальні машини на Firecracker мають такі ж самі обмеження. «Холодний» запуск займає приблизно однаковий час на обидвох гіпервізорах. При цьому Firecracker швидше ініціалізує мережу та швидше виконує відновлення роботи ВМ із образу оперативної пам’яті та стану гіпервізору. Причиною такої поведінки є те, що Firecracker використовує механізм ядра Memory Mapping для відображення файлу з образом пам’яті на віртуальний простір пам’яті процесу, таким чином уникаючи завантаження файлу цілком у пам’ять одразу. Сторінки пам’яті підвантажуються у ВМ динамічно, що призводить до більш повільної роботи на початку, але дає можливість відновлювати роботу ВМ швидше, особливо якщо їм виділено багато пам’яті.

Монітор віртуальних машин, що розробляється в рамках цієї дипломної роботи, передбачає що користувач не буде вручну вимикати віртуальні

машини, а отже більшість з них буде знаходитись у стані призупиненої роботи і відновлюватимуться монітором із образів пам’яті. В такому випадку швидкість ініціалізації мережі та відновлення із образу є критичною. Отже, в якості гіпервізору буде використовуватись саме Firecracker.

2.4. Обґрунтування вибору проксі-серверу

За технічним завданням, монітор віртуальних машин має фіксувати не тільки IP-трафік, а ще і HTTP. Для маршрутизації IP-трафіку можна використати netfilter, що дозволяє монітору перехоплювати пакети та аналізувати їх, а також створювати правила пересилки. При цьому netfilter не аналізує протоколи вище 4 рівня за моделлю OSI, тобто для маршрутизації одного netfilter буде недостатньо, потрібен ще і проксі-сервер, який зможе виконувати схожі функції, але для протоколу HTTP.

Для ефективної роботи з монітором віртуальних машин, проксі-сервер має мати наступні властивості:

- Висока швидкість роботи
- Наявність можливості зміни конфігурації без перезавантаження
- Наявність можливості приймати підключення на різних інтерфейсах
- Стабільність при великій кількості запитів на різні символьні адреси
- Можливість запуску процесу серверу вручну з підвантаженням конфігурації із заданої директорії

Були розглянуті такі сервери як Apache Traffic Server, nginx, HAProxy, Envoy та Traefik. Усі з них вміють в перезавантаження конфігурації без скидання існуючих підключень, але Apache Traffic Server може читати конфігурацію лише з каталогу /etc. Також усі сервери, окрім Traefik, виявились стабільними і не губили запити, навіть при високих навантаженнях.

В Envoy використовується гібридна модель, що поєднує багатопоточність та неблокуючий ввід-вивід, яка забезпечує високу швидкодію цього серверу [28]. Результати порівняння наведені в таблиці 2.3.

Таблиця 2.3 – Порівняння проксі-серверів

Критерій	Проксі-сервер				
	HAProxy	nginx	Envoy	Traefik	ATS
Динамічна зміна конфігурації	Так	Так	Так	Так	Так
Завантаження конфігурації з власної директорії	Так	Так	Так	Так	Ні
Швидкість	Висока	Висока	Дуже висока	Висока	Висока
Кешування	Ні	Так	Ні	Ні	Так
Стабільність	Дуже висока	Висока	Дуже висока	Середня	Висока

Краще за все себе показав Envoy, який відповідає всім вимогам та має хороший ступінь стабільності та швидкодії. Отже, в якості проксі серверу буде використано саме його.

2.5. Висновки до розділу 2

У цьому розділі дипломного проєкту було проведено аналіз засобів, необхідних для розробки програмного комплексу розподілу обчислювального часу. На основі вимог технічного завдання та практичних міркувань, було

виведено критерії вибору мови програмування, сховища конфігурації, гіпервізору та проксі-серверу.

Було оцінено доцільність використання низькорівневих мов програмування накшталт C чи C++ та детально розглянуто високорівневі мови. З міркувань безпеки та надійності було обрану мову програмування Java та досліджено яким чином можна підвищити ефективність використання ресурсів інтерпретатором цієї мови. Не зважаючи на це, зручність інтеграції та висока швидкість C не можна було ігнорувати і для розробки деяких компонентів програмного комплексу було вирішено використовувати цю мову також, але в обмеженій кількості.

Також були розглянуті підходи до зберігання конфігурації, зокрема було порівняно зберігання у розподілених базах даних типу «ключ-значення», у реляційних базах даних та у текстових файлах. З огляду на специфічні потреби програмного комплексу, було прийнято рішення зберігати налаштування у реляційній базі даних. Різні реляційні СКБД були порівняні між собою за швидкодією та ліцензійними обмеженнями, в результаті чого для використання було обрано Oracle Database.

Використаний у цьому дипломному проєкті гіпервізор Firecracker було порівняно із альтернативою, що зараз доступна у QEMU. В результаті аналізу було підтверджено, що Firecracker підходить більше через більш швидке відновлення роботи віртуальних машин. Для забезпечення роботи HTTP-маршрутизації був обраний проксі-сервер Envoy, адже він показав себе як стабільне та швидке рішення, що задовольняє вимоги.

Прийняті технологічні рішення дозволяють побудувати надійну основу для функціоналу програмного комплексу, використовуючі перевірені часом інструменти, що відповідають сучасним стандартам та найбільш підходять для використання у системах такого типу.

3. ОПИС РОЗРОБЛЕНОГО МОНІТОРУ ВМ

В рамках цієї дипломної роботи було створено програмний комплекс для забезпечення роботи віртуальних машин, їх створення, видалення та редагування їх налаштувань, зокрема налаштувань мережевого підключення та політики автоматичного призупинення та відновлення їх роботи. Програмний комплекс складається з серверів баз даних, проксі серверу та, власне, розробленого під час дипломного проєктування монітору віртуальних машин, що встановлюється на один або декілька вузлів (див. частину 3.3). Цей розділ присвячений опису архітектури монітору віртуальних машин та принципів роботи його компонентів.

3.1. Загальна архітектура монітору ВМ

3.1.1. Структура проєкту

Монітор віртуальних машин побудований на основі фреймворку Spring Boot, який не вимагає специфічної структури директорій для роботи. Проте в документації наведено типову структуру проєкту [29], що зображена на рисунку 3.1.

```
com
 +- example
     +- myapplication
         +- MyApplication.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

Рисунок 3.1 – Типова структура проєкту Spring Boot

Як можна побачити, головний клас застосунку знаходиться в кореневій директорії свого пакету, а основний функціонал розташований у підпакетах. Розташування по підпакетам відбувається по сутностям: кожна сутність розміщена у одному пакеті із пов'язаними з нею контролером, репозиторієм та службою.

Структура розробленого монітору віртуальних машин наведена на рисунку 3.2 та схожа на типову, проте в кожен підпакет може входити більше однієї сутності і, відповідно, більше пов'язаних об'єктів. На відміну від типової схеми організації проєкту, в цьому випадку розташування відбувається не стільки по сутностям, скільки по групам функціоналу.

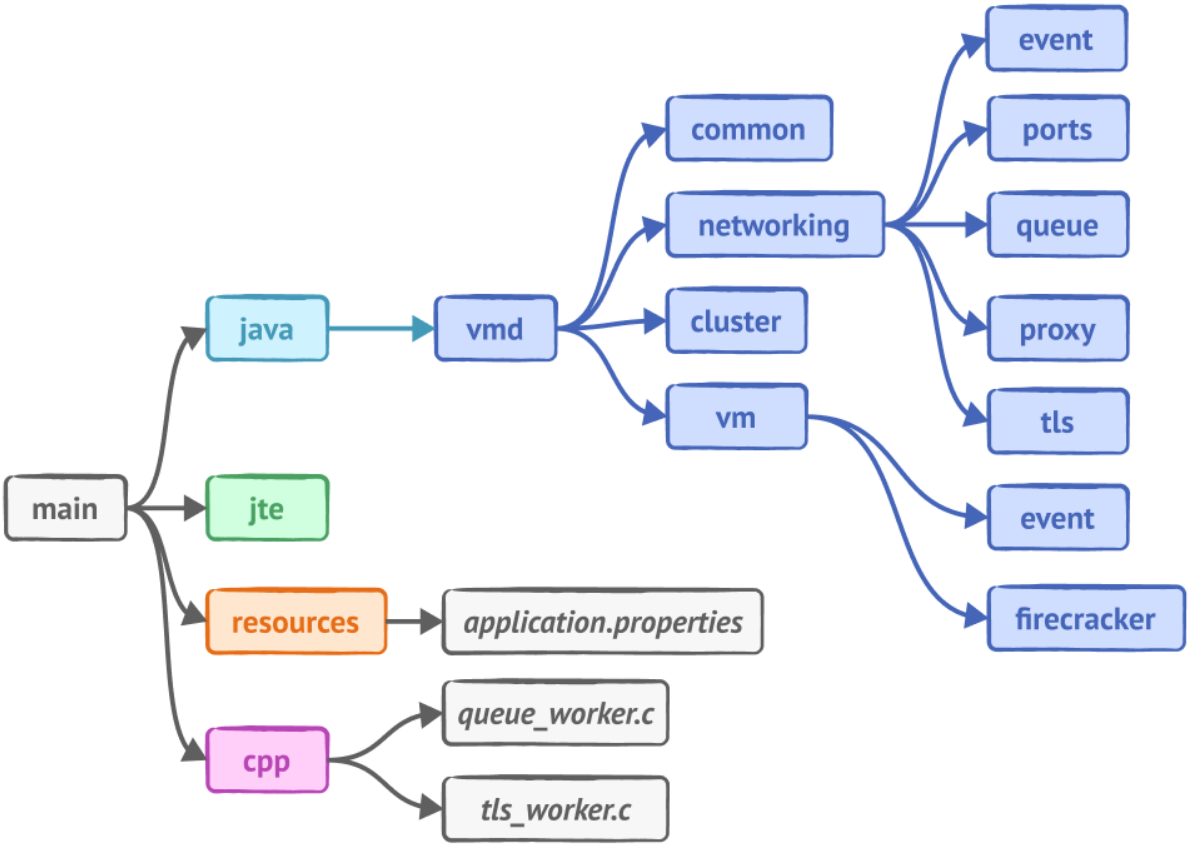


Рисунок 3.2 – Узагальнена структура монітору віртуальних машин

Як і в більшості проєктів, написаних мовою Java, вихідний код застосунку знаходиться у директорії src/main, а саме у піддиректоріях виду

src/language, де language – ім’я мови програмування. Виключенням до цього правила є директорія resources, що зберігає статичні ресурси. У моніторі ВМ безпосередньо використовується три мови програмування: Java, JTE (Java Template Engine) та C, тому піддиректорій теж три: java, jte та cpp. Розглянемо структуру більш детально:

- src/main – Головна директорія з вихідним кодом
  - jte – Директорія з шаблонами, написані мовою програмування JTE
    - envoy-config.jte – Шаблон по якому генерується конфігурація проксі серверу Envoy
    - kea-config.jte – Шаблон по якому генерується конфігурація DHCP-серверу Kea
    - waitingRoom.jte – Шаблон сторінки очікування, що використовується компонентом маршрутизації HTTP-трафіку (див. пункт 3.7.5)
  - cpp – Директорія із кодом, що написаний мовами C/C++
    - queue_worker.c – Код програми, що використовується для перехоплення IP-пакетів (див. частини 3.6, 3.7)
    - tls_worker.c – Код програми, що використовується для маршрутизації TLS-трафіку (див. пункт 3.7.6)
  - java – Директорія із основним кодом монітору ВМ, що написаний мовою Java
  - VmdApplication.java – головний клас застосунку
  - vmd – Директорія кореневого пакету застосунку
    - common – Спільні для деяких компонентів об’єкти та конфігураційні класи
    - cluster – Код компоненту для роботи з вузлами кластеру (див. частину 3.3)

- vm – Код компоненту для роботи з ВМ (див. частини 3.4, 3.5)
    - event – Класи подій, що генеруються компонентом для роботи з ВМ
    - firecracker – Службові об’єкти для роботи з гіпервізором
- networking – Код компоненту для роботи з мережею
    - event – Класи подій, що генеруються компонентом для роботи з мережею
    - ports – Підкомпонент для роботи з переадресацією портів (див. пункт 3.7.3)
    - proxy – Підкомпонент для роботи з HTTP-трафіком (див. пункт 3.7.5)
    - sni_forwarding – Підкомпонент для роботи з TLS-трафіком (див. пункт 3.7.6)
    - queue – Підкомпонент, що відповідає за перехоплення IP-пакетів (див. частини 3.6, 3.7)

Варто зазначити, що наведена вище структура є спрощеною і не містить детальних описів та повного переліку класів та підпакетів. Компоненти та їх повна структура описуються детально в інших пунктах цього розділу.

3.1.2. Типова структура компоненту

Розглянемо типову структуру компоненту монітору ВМ на прикладі компоненту для роботи з переадресацією портів (.vmd.ports), що наведена на рисунку 3.3.

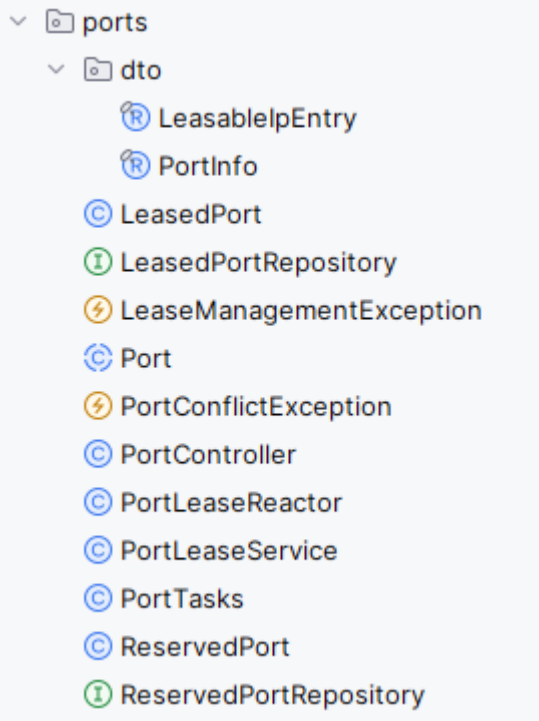


Рисунок 3.3 – Типова структура компоненту

В директорії компоненту зберігаються наступні об’єкти:

- Сутність – модель певного об’єкту, яким оперує компонент, з анотаціями JPA (Java Persistence API) для збереження у БД
- Виключення – спеціальні класи, що являють собою помилку або непередбачений стан
- Служба – клас, що містить у собі бізнес-логіку компоненту
- Репозиторій – інтерфейс, який описує методи для роботи з БД
- Контролер – клас, що містить у собі логіку обробки HTTP-запитів, зазвичай для реалізації методів API

Кожному об’єкту системи відповідає своя сутність та репозиторій для неї, а взаємодія користувачів з об’єктами відбувається за допомогою контролерів. Кожен контролер може відповідати одному або декільком об’єктам, в залежності від складності компоненту: наприклад, у цьому компоненті контролер PortController використовується для API-методів, що

керують резервуванням портів (об’єкт ReservedPort) та правилами переадресації трафіку між портами хоста та ВМ (об’єкт LeasedPort).

При цьому бізнес-логіка компоненту має бути розташована в одній службі. Таке правило дозволяє запобігти створенню зайвих об’єктів та мотивує розділяти код на підкомпоненти на інтуїтивному рівні.

Загалом структура компоненту є реалізацією патерну MVC (Model-View-Controller), але, оскільки контролери використовуються здебільшого для API, шар презентації окремо не представлений, його роль виконується класами сутностей, що автоматично серіалізуються в JSON засобами фреймворку. У випадках коли така серіалазіція неможлива через складність сутності чи наявності в ній приватних даних, або коли в якості результата виконання запиту бажано повернути об’єкт, який містить у собі більше інформації, ніж в одній сутності, використовуються DTO (Data Transfer Object).

У цьому застосунку DTO прийнято зберігати в окремому підпакеті з назвою dto всередені компоненту. DTO являють собою прості записи (Records) та не містять бізнес-логіки, приклад такого запису наведений на рисунку 3.4.

```
public record PortInfo(int number, ReservedPort.Type reservedType, PortForwardInfo forwardRule) {
    public record PortForwardInfo(String vmId, int vmPort, LeasedPort.Protocol protocol) {}
}
```

Рисунок 3.4 – DTO з інформацією про резервацію або правило переадресації порту

3.1.3. Використання системи подій

Фреймворк Spring Boot містить реалізацію патерну Observer у вигляді системи подій, що дозволяє організувати слабко зв’язану взаємодію між компонентами. Для використання цієї системи достатньо створити клас події, що може наслідуватись від класу ApplicationEvent та «опублікувати» подію через клас ApplicationEventPublisher, підписка на події відбувається за допомогою аннотації @EventListener [30].

Основна цінність системи подій полягає у тому, що реєстрація обробника відбувається незалежно від джерела. Отже, обробник та джерело не мають залежати один від одного, що у свою чергу дає змогу спростити код та запобігти виникненню проблеми циклічної залежності.

Ця система фреймворку активно використовується у розробленому застосунку, адже в ньому наявна велика кількість компонентів, які мають швидко реагувати на зміни в стані системи. Наприклад, службовий поток переадресації трафіку між портами має коригувати правила netfilter у випадку зміни внутрішньої IP-адреси ВМ та видаляти їх, якщо ВМ була призупинена чи деактивована.

Події, що можуть бути створені компонентом визначені як публічні класи та зберігаються у підпакеті event. На рисунку 3.5 наведено перелік подій, що можуть генеруватися компонентом роботи з мережею та його підкомпонентами.

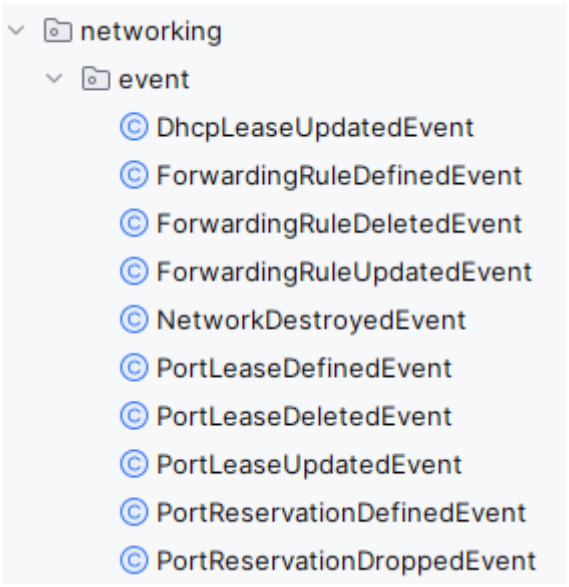


Рисунок 3.5 – Події компоненту роботи з мережею

3.2. Перелік сутностей та схема БД

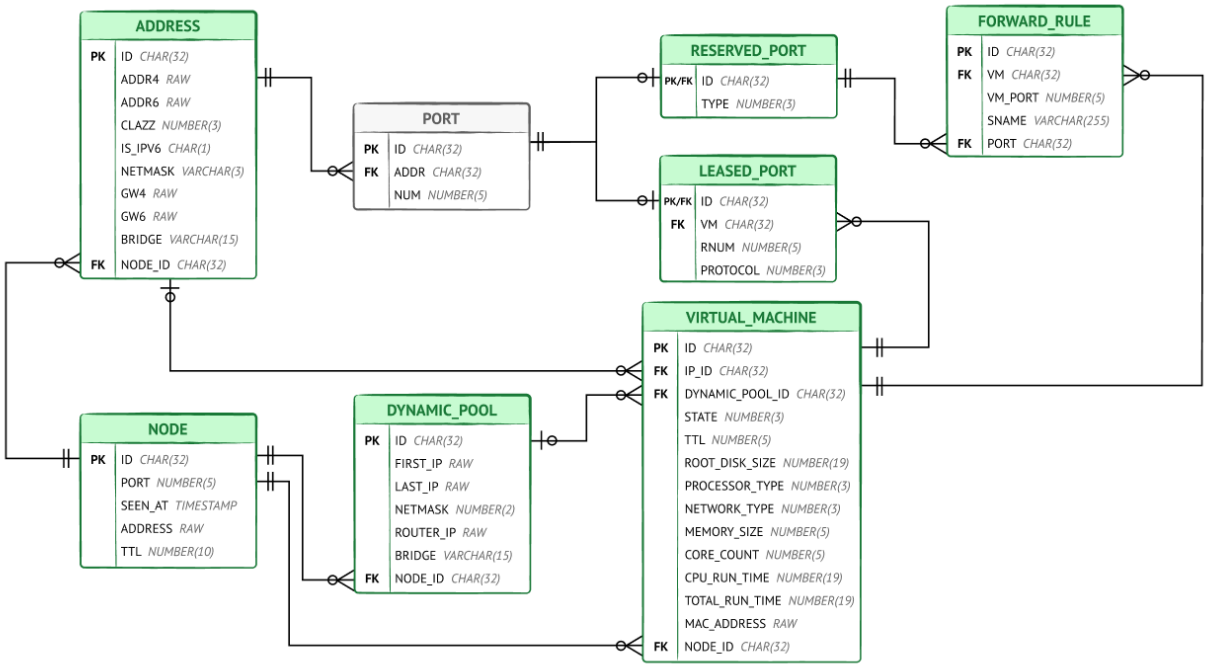


Рисунок 3.6 – Діаграма взаємозв’язків

Монітор ВМ зберігає свій стан та налаштування в базі даних, структуру якої можна побачити на рисунку 3.6. База даних складається із 8 таблиць, одна з яких є службовою:

- NODE – таблиця вузлів кластеру
  - ID – унікальний ідентифікатор вузлу
  - PORT – порт, на якому монітор ВМ приймає API-запити
  - ADDRESS – IP-адреса, на якій монітор ВМ приймає API-запити
  - TTL – інтервал оновленя SEEN_AT у секундах
  - SEEN_AT – часова мітка, що оновлюється монітором ВМ раз на TTL секунд
- ADDRESS – таблиця IP-адрес
  - ID – унікальний ідентифікатор IP-адреси

- IS_IPV6 – версія протоколу IP, приймає значення ‘Y’, якщо шоста
- ADDR4, ADDR6 – IP-адреса
- GW4, GW6 – IP-адреса шлюзу
- NETMASK – мережева маска
- CLAZZ – клас IP-адреси:
  - 0 – адреса, яку можна використати для правил переадресації портів, HTTP та TLS маршрутизації
  - 1 – адреса, яку можна призначити ВМ
  - 2–255 – зарезервовано
- BRIDGE – назва мосту, який підключено до інтерфейсу, на який маршрутизуються IP-пакети для цієї адреси
- NODE_ID – унікальний ідентифікатор вузла

- DYNAMIC_POOL – таблиця локальних мереж, в які можна об’єднати ВМ без виділеної IP-адреси
  - ID – унікальний ідентифікатор мережі
  - FIRST_IP – перша IP-адреса, яка доступна для ВМ
  - LAST_IP – остання IP-адреса, яка доступна для ВМ
  - ROUTER_IP – IP-адреса, яка буде призначена хосту віртуалізації
  - BRIDGE – назва мосту, через який буде організовано роботу мережі
  - NODE_ID – унікальний ідентифікатор вузла
- PORT – службова таблиця, містить спільні властивості сутностей ReservedPort та LeasedPort
  - ID – унікальний ідентифікатор порту
  - ADDR – унікальний ідентифікатор IP-адреси
  - NUM – номер порту
- RESERVED_PORT

- ID – унікальний ідентифікатор порту
- TYPE – вид порту:
  - 0 – зарезервований для використання на хості, недоступний для створення правил переадресації
  - 1 – доступний лише для правил маршрутизації TLS-трафіку
  - 2 – доступний лище для правил маршрутизації HTTP-трафіку
  - 3–255 – зарезервовано
- LEASED_PORT – таблиця правил переадресації портів
  - ID – унікальний ідентифікатор порту
  - VM – унікальний ідентифікатор ВМ
  - RNUM – порт ВМ, на який буде переадресовано трафік
  - PROTOCOL – протокол:
    - 0 – TCP
    - 1 – UDP
    - 2 – SCTP
    - 3 – DCTP
    - 4–255 – зарезервовано
- FORWARD_RULE – таблиця правил маршрутизації TLS/HTTP трафіку
  - ID – унікальний ідентифікатор правила
  - PORT – унікальний ідентифікатор порту
  - VM – унікальний ідентифікатор ВМ
  - VM_PORT – порт ВМ, на який буде переадресовано трафік
  - SNAME – символьна адреса
- VIRTUAL_MACHINE – таблиця віртуальних машин

- ID – унікальний ідентифікатор ВМ
- ROOT_DISK_SIZE – розмір завантажувального диску у МБ
- MEMORY_SIZE – розмір оперативної пам’яті у МБ
- CORE_COUNT – кількість ядер процесору, доступних ВМ
- PROCESSOR_TYPE – тип емульованого процесору:
    - 0 – AMD EPYC (Milan)
    - 1 – Intel Skylake
    - 2 – такий самий, як на хості
    - 3–255 – зарезервовано
- NETWORK_TYPE – тип підключення до мережі:
    - 0 – через виділену IP-адресу
    - 1 – через локальну мережу
    - 2–255 – зарезервовано
- MAC_ADDRESS – MAC-адреса
- IP_ID – унікальний ідентифікатор виділеної IP-адреси
- DYNAMIC_POOL_ID – унікальний ідентифікатор локальної мережі
- TTL – час (у секундах), після якого ВМ вважається неактивною за умови відсутності мережевого трафіку та стає в чергу на призупинення
- STATE – стан віртуальної машини:
    - 0 – працює
    - 1 – призупинена
    - 2 – призупинена, образ пам’яті відсутній
    - 3 – призупинена внаслідок помилки гіпервізору чи монітору

- 4 – створюється
- 5 – виникла помилка при створенні
- 6 – конфігурація оновлюється
- 7 – призупинена та недоступна для автоматичного запуску
- 8–255 – зарезервовано

- TOTAL_RUN_TIME – час роботи ВМ у секундах
- CPU_RUN_TIME – процесорний час, витрачений на виконання ВМ, у секундах
- NODE_ID – унікальний ідентифікатор вузлу, на якому розміщена ВМ

Як було зазначено в другому розділі цієї роботи, можливість перевірки коректності конфігурації на стороні серверу була важливою причиною вибору саме реляційної БД в якості сховища, а отже в схемі БД також були створені обмеження, перелій яких наведено в таблиці 3.1.

Таблиця 3.1 – Перелік обмежень

Таблиця	Вид обмеження	Стовпчики	Опис
NODE	Первинний ключ	ID	–
VIRTUAL_MACHINE	Первинний ключ	ID	–
	Зовнішній ключ	NODE_ID	Ідентифікатор вузла має бути валідним
	Зовнішній ключ	DYNAMIC_POOL_ID	Ідентифікатор локальної

			мережі

Продовження таблиці 3.1

VIRTUAL_MACHINE	Зовнішній ключ	IP_ID	Ідентифікатор IP-адреси має бути валідним
	Унікальний ключ	IP_ID	Кожну виділену IP-адресу можна призначити тільки одній ВМ
	Перевірка	CORE_COUNT	Кількість ядер має бути від 1 до 16 включно
	Перевірка	MEMORY_SIZE	Розмір пам’яті має бути більше 256 МБ
	Перевірка	NETWORK_TYPE	Тип підключення має існувати
	Перевірка	PROCRSSOR_TYPE	Тип прцоесору має існувати
	Перевірка	ROOT_DISK_SIZE	Розмір диску має бути більше 256

			МБ

Продовження таблиці 3.1

VIRTUAL_MACHINE	Перевірка	STATE	Стан ВМ має бути валідним
	Індекс	IP_ID	–
PORT	Первинний ключ	ID	–
	Зовнішній ключ	ADDR	Ідентифікатор IP-адреси має бути валідним
	Індекс	ADDR	–
		NUM	
	Унікальний ключ	ADDR	Для кожного порту має бути тільки один запис в БД
		NUM	
RESERVED_PORT	Первинний ключ	ID	Ідентифікатор порту має бути валідним
	Зовнішній ключ		
	Перевірка	TYPE	Тип резервування має бути валідний

LEASED_PORT	Первинний ключ	ID	-

Продовження таблиці 3.1

LEASED_PORT	Зовнішній ключ	ID	Ідентифікатор порту має бути валідним
	Зовнішній ключ	VM	Ідентифікатор ВМ має бути валідним
FORWARD_RULE	Первинний ключ	ID	–
	Зовнішній ключ	PORT	Ідентифікатор порту має бути валідним
	Зовнішній ключ	VM	Ідентифікатор ВМ має бути валідним
	Унікальний ключ	PORT	Символьні адреси мають бути унікальними в межах одного порту
		SNAME	
ADDRESS	Первинний ключ	ID	–
	Зовнішній	NODE_ID	Ідентифікатор

	ключ		вузла має бути валідним

Продовження таблиці 3.1

ADDRESS	Перевірка	CLAZZ	Клас IP-адреси має бути валідним
	Перевірка	IS_IPV6	Маркер версії протоколу має бути валідним (Y або N)
	Перевірка	IS_IPV6	Адреси мають бути встановлені у відповідності із маркером протоколу
		ADDR4	
		ADDR6	
		GW4	
		GW6	

3.3. Забезпечення роботи системи в кластері

Для роботи монітору ВМ потрібно розгорнути його разом із СКБД Oracle Database та аналітичною базою даних ClickHouse. Ці програми мають властивість забирати ресурси серверу, іноді у великих об’ємах, що не може не вплинути на ефективність роботи вузла віртуалізації. З метою вирішення цієї проблеми була передбачена можливість підключення декількох вузлів кластеру до віддаленного серверу, на якому будуть працювати СКБД та аналітична БД.

Для реалізації цієї можливості в схему БД було додано стовпчики NODE_ID, за якими монітори ВМ можуть розрізняти сутності за принципом

приналежності до вузла. Підключення вузла до кластеру відбувається за допомогою налаштувань, що можуть передаватись за допомогою параметрів командного рядка, перелік яких наведено у таблиці 3.2.

Таблиця 3.2 – Перелік параметрів, що контролюють налаштування кластеризації

Параметр	Формат	Опис
-D vmd.cluster.node-id	32-символьний рядок	Унікальний ідентифікатор кластеру
-D vmd.cluster.ip-addr	IP-адреса	IP-адреса, на якій вузол здатний приймати API-запити
-Dvmd.cluster.port	Число від 1 до 65536	Порт, на якому приймаються API-запити

Передавати достатньо лише перші два параметри, останній за умовчанням дорівнює 32032. Варто зазначити, що параметр -Dvmd.cluster.port не контролює безпосередньо порт, який буде прослуховувати монітор ВМ, а лише визначає яке число буде покладено у БД. Для того щоб змінити порт прослуховування треба використати параметр -Dserver.port. Якщо передати параметр -Dserver.port без -Dvmd.cluster.port, то в БД буде покладено порт прослуховування замість порту за умовчанням. Подібна гнучкість дозволяє застосувати монітор ВМ у середовищах, де потрібно, щоб локальний та зовнішній порт відрізнялись.

За роботу вузла в кластері відповідає компонент кластерізації, що розміщений у пакеті vmd.cluster, структура якого наведена на рисунку 3.7.

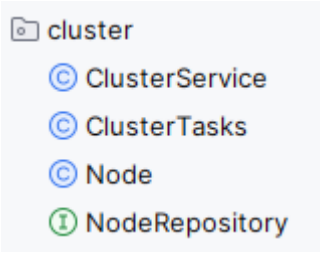


Рисунок 3.7 – Структура компоненту кластеризації

Цей компонент складається із служби ClusterService, що містить у собі наступні методи:

- registerSelf – повідомити БД про те, що локальний вузол готов до роботи
- getLocalNode – отримати сутність вузла у вигляді об’єкту класу Node
- getLocalNodeId –отримати ідентифікатор локального вузла

Метод registerSelf викликається автоматично кожні 3500 секунд засобами фреймворку, визначення цієї задачі знаходиться у класі ClusterTasks (див. рисунок 3.8).

```
@Scheduled(fixedDelay = 3500000)
public void registerSelf() {
    this.clusterService.registerSelf();
    logger.info("Registered node within cluster");
}
```

Рисунок 3.8 – Код задачі registerSelf

Більшість служб інших компонентів залежать від служби ClusterService та використовують її для визначення приналежності певного ресурсу до вузла, приклад такої перевірки в компоненті роботи з мережею наведено на рисунку 3.9.

```
public LeasedPort leasePort(@NotNull VirtualMachine vm, String ipId, char hostPort, char vmPort,
                            LeasedPort.Protocol protocol)
        throws IllegalArgumentException, ForeignResourceException, PortConflictException {
    var ip = this.addressRepository.findById(ipId).orElseThrow(() ->
            new IllegalArgumentException("Cannot find address with id " + ipId));
    if (!ip.getNodeId().equals(this.clusterService.getLocalNodeId()))
        throw new ForeignResourceException("IP " + ipId + " is not owned by this node");
```

Рисунок 3.9 – Перевірка приналежності IP-адреси локальному вузлу

При виникненні ситуації, коли дію над ресурсом неможливо виконати через те, що він не створений на локальному вузлі, викидується виключення ForeignResourceException, що визначено у пакеті vmd.common (див. рисунок 3.10).

```
public class ForeignResourceException extends RuntimeException {
    public ForeignResourceException(String message) { super(message); }
}
```

Рисунок 3.10 – Визначення виключення ForeignResourceException

Відповідальність за визначення до якого вузла треба сформувати API-запит для зміни певного ресурсу покладається на користувача, тому у випадках, коли дія мала виконатись по запиту користувача через API, у відповідь повертається помилка із HTTP-кодом 400 Bad Request. Пошук адреси вузла пропонується виконувати за допомогою JOIN-запиту до БД.

3.4. Механізм створення ВМ

У розробленому програмному комплексі монітор ВМ має можливість створювати нові ВМ за API-запитом користувача. За створення ВМ відповідає служба VirtualizationService, а саме метод createVirtualMachine. Щоб розпочати створення, у цей метод треба передати конфігурацію ВМ, ім’я хосту ВМ, назву шаблону ОС та пароль. Приклад такого об’єкту наведено у вигляді JSON на рисунку 3.11.

```
{
  "config": {
    "coreCount": 4,
    "memorySize": 2048,
    "rootDiskSize": 2048,
    "networkType": "DEDICATED_IP",
    "ipAddressId": "3639CEBC096D9BA4E065000000000001",
    "sensitivity": "IN_OUT_PACKETS",
    "ttl": 999
  },
  "template": "fedora-42",
  "hostname": "test-vm-2.local",
  "password": "enanan"
}
```


Рисунок 3.11 – Конфігурація нової ВМ

Після отримання цієї інформації метод createVirtualMachine виконує наступні дії:

- Перевіряє налаштування мережі (рисунок 3.12)
- Перевіряє валідність імені хоста (рисунок 3.13)
- Перевіряє валідність паролю (рисунок 3.13)
- Перевіряє наявність шаблону ОС (рисунок 3.13)
- Зберігає налаштування в БД (рисунок 3.14)
- Розпочинає процес створення ВМ та відправляє події (рисунок 3.14)

```
if (config.networkType == VirtualMachine.NetworkType.SHARED_IP) {
    if (config.dynamicPoolId == null)
        throw new MalformedConfigException("Dynamic pool is required for this configuration");

    var pool = networkingService.getPool(config.dynamicPoolId);
    if (pool == null)
        throw new MalformedConfigException("Dynamic pool " + config.dynamicPoolId + " not found");
    else if (!pool.getNode().getId().equals(this.clusterService.getLocalNodeId()))
        throw new MalformedConfigException("Dynamic pool " + config.dynamicPoolId + " is foreign");

    vm.setDynamicPool(pool);
} else if (config.networkType == VirtualMachine.NetworkType.DEDICATED_IP) {
    if (config.ipAddressId == null)
        throw new MalformedConfigException("IP address is required for this configuration");

    var address = networkingService.getAddress(config.ipAddressId);
    if (address == null)
        throw new MalformedConfigException("IP address " + config.ipAddressId + " not found");
    else if (!address.getNode().getId().equals(this.clusterService.getLocalNodeId()))
        throw new MalformedConfigException("IP address " + config.ipAddressId + " is foreign");
    else if (address.getClazz() != Address.AddressClass.ADDR_ASSIGNABLE)
        throw new MalformedConfigException("IP address " + config.ipAddressId + " is not assignable as dedicated");
    else if (this.vms.existsByIp(address))
        throw new MalformedConfigException("IP address " + config.ipAddressId + " is already assigned");

    vm.setIp(address);
}
```

Рисунок 3.12 – Перевірка налаштувань мережі

```
if (!this.FQDN_PATTERN.matcher(hostname).matches())
    throw new MalformedConfigException("Hostname " + hostname + " is not valid");

if (password.contains("\n") || password.contains("\r") || password.contains("\0"))
    throw new MalformedConfigException("Password " + password + " contains line break or terminator");

File templateRecipe = new File( pathname: this.home + "/templates/" + template + "/recipe.rb");
if (!templateRecipe.exists())
    throw new MalformedConfigException("Template " + template + " not found");
```

Рисунок 3.13 – Перевірка імені хоста, паролю та шаблону ОС

```
var mac = new byte[3];
new Random().nextBytes(mac);
vm.setMacAddress(mac);

vm.setTtl(config.ttl);
vm.setCoreCount(config.coreCount);
vm.setMemorySize(config.memorySize);
vm.setSensitivity(config.sensitivity);
vm.setRootDiskSize(config.rootDiskSize);
vm.setProcessorType(config.processorType);
vm.setState(VirtualMachine.State.CREATING);
vm.setNode(this.clusterService.getLocalNode());
this.vms.save(vm);

this.prepareVmHome(vm, template, hostname, password);
this.eventPublisher.publishEvent(new VirtualMachineCreatedEvent(vm));
this.eventPublisher.publishEvent(new VirtualMachineDefinedEvent(vm));
```

Рисунок 3.14 – Початок створення ВМ та збереженя конфігурації в БД

Процес створення ВМ відбувається у методі prepareVmHome, який викликає сценарій створення ВМ із шаблону ОС. Створення шаблону ОС та його розповсюдження по вузлам – відповідальність адміністратора, монітор ВМ лише перевіряє наявність скрипта recipe.rb у директорії з шаблоном та передає йому керування. Адміністратор може організувати структуру шаблону згідно зі своїми вподобаннями, єдиною вимогою є наявність скрипта, що визначає функцію, сигнатура якої наведена на рисунку 3.15.

```
def prepareRootFs(tpl_home, id, hostname, password,
                  disk_size, use_dhcp, use_ipv6, gw, addr, nmask)
```

Рисунок 3.15 – Сигнатура функції prepareRootFs

Скрипт пишеться мовою програмування Ruby, що часто використовується у DevOps-інструментах через свою гнучкість та простоту. Монітор ВМ створює директорію ВМ та викликає функцію prepareRootFs, яка має повернути значення true. Якщо ця функція повертає false чи створює виключення, монітор ВМ переводить ВМ у стан CREATION_FAILED.

Результатом роботи цієї функції має бути створення в директорії ВМ файлу root.vhd з образом диску, на якому встановлена ОС з відповідними налаштуваннями, файлу kernel.elf з ядром ОС та файлу initrd.img з образом тимчасової файлової системи для ініціалізації ОС. Для двох останніх файлів

допускається створення символічних посилань на файли у директорії шаблону ОС.

На рисунках 3.16, 3.17 та 3.18 наведено приклад скрипту створення ВМ зі встановленою ОС Fedora Linux 42.

```
def prepareRootFs(tpl_home, id, hostname, password, disk_size, use_dhcp, use_ipv6, gw, addr, nmask)
    vm_home = "/opt/castorice/vms/#{id}"

    Dir.mkdir(vm_home + "/rootfs")
    rootfs_extracted = system("tar -xJf #{tpl_home}/rootfs.tar.xz -C #{vm_home}")
    unless rootfs_extracted
        puts "Error while extracting rootfs"
        return false
    end

    password_changed = system("printf \"#{password}\\n#{password}\\n\" | chroot #{vm_home}/rootfs passwd")
    unless password_changed
        puts "Error while changing password"
        return false
    end

    File.write("#{vm_home}/rootfs/etc/hostname", hostname)

    if use_dhcp
        prepareDhcpNetplan(vm_home)
    else
        prepareIpNetplan(vm_home, use_ipv6, gw, addr, nmask)
    end

    disk_file = vm_home + "/root.vhd"
    disk_size *= 1024 * 1024
    chunk_size = 1024 * 1024 * 64

    File.open(disk_file, "wb") do |f|
        (disk_size / chunk_size).times { f.write("\x00" * chunk_size) }
    end

    fs_created = system("mkfs.btrfs -r #{vm_home}/rootfs #{vm_home}/root.vhd")
    unless fs_created
        puts "Error while creating filesystem"
        return false
    end

    FileUtils.rm_rf("#{vm_home}/rootfs")
    system("ln -s #{tpl_home}/kernel.elf #{vm_home}/kernel.elf")
    system("ln -s #{tpl_home}/initrd.elf #{vm_home}/initrd.elf")

    return true
end
```

Рисунок 3.16 – Функція prepareRootFs

```
def prepareDhcpNetplan(vm_home)
    File.open("#{vm_home}/rootfs/etc/netplan/00-castorice.yaml", "w") do |plan|
        plan << "network:\n"
        plan << "    version: 2\n"
        plan << "    renderer: NetworkManager\n"
        plan << "    ethernets:\n"
        plan << "        eth0:\n"
        plan << "            dhcp4: yes\n"
        plan << "            dhcp6: no\n"
    end
end
```

Рисунок 3.17 – Функція prepareDhcpNetplan для налаштування мережі

```
def prepareIpNetplan(vm_home, use_ipv6, gw, addr, nmask)
    File.open("#{vm_home}/rootfs/etc/netplan/00-castorice.yaml", "w") do |plan|
        plan << "network:\n"
        plan << "    version: 2\n"
        plan << "    renderer: NetworkManager\n"
        plan << "    ethernets:\n"
        plan << "        eth0:\n"
        plan << "            dhcp4: no\n"
        plan << "            dhcp6: no\n\n"

        plan << "            addresses:\n"
        plan << "                - #{addr}/#{nmask}\n"
        plan << "            routes:\n"
        plan << "                - to: default\n"
        plan << "                  via: #{gw}\n"
        plan << "            nameservers:\n"
        plan << "                addresses:\n"
        plan << "                    - 9.9.9.9\n"
        plan << "                    - 149.112.112.112\n"
    end
end
```

Рисунок 3.18 – Функція prepareIpNetplan для налаштування мережі

Як можна побачити, цей скрипт розпаковує заздалегіть підготовлений архів зі змістом кореневої файлової системи, змінює пароль користувача за допомогою утиліт chroot та passwd, генерує конфігурацію Netplan, створює образ диску із файловою системою BTRFS та переносить файли ОС зі зміненою конфігурацією на нього. Після чого виконується видалення тимчасових файлів та створюються символьні посилання на файли ядра та тимчасової ФС.

Архів зі змістом ФС ОС Fedora Linux 42 було підготовлено в рамках дипломного проєктування за допомогою утілити mkosi, що дозволяє створювати образи ФС через функціонал пакетних менеджерів ОС. Для потреб

цієї роботи було створено архів, що містить в собі стандартні пакети Fedora Server (роль server) та програму Netplan для можливості визначення налаштувань мережі через YAML-файли, що дозволило спростити скрипт recipe.rb.

Також було взяте звичайне ядро Linux, що знаходилось у репозиторіях Fedora Linux. Ініціалізуюча файлова система була створена в контейнері Fedora за допомогою утілити dracut, до якої було передано флаг --no-host-only, що дозволило включити в ІФС додаткові модулі, необхідні для запуска ОС на інших комп'ютерах (в даному випадку – всередині ВМ).

3.5. Механізм моніторінгу ВМ

Функції моніторінгу ВМ в реалізованому застосунку можна умовно поділити наступним чином:

- Спостереження за мережевим трафіком
- Спостереження за станом ВМ та керування процесами гіпервізора

Спостереження за мережевим трафіком виконується відповідним компонентом та залежить від налаштувань конкретної ВМ, цей механізм буде детально розглянутий в інших частинах цього розділу. Ця ж частина присвячена функціям спостереження за станом віртуальних машин та тим, як відбувається керування процесами.

3.5.1. Взаємодія із менеджером гіпервізора

Менеджер прцоесів гіпервізора реалізований в класі FireCrackerServiceThread, що знаходиться у пакеті vmd.vm.firecracker. Цей клас реалізує інтерфейс Runnable і призначений для запуска в окремому потоці програми. Взаємодія із менеджером відбувається асинхронно, за допомогою записів AsyncFireCrackerRequest, визначення яких наведено на рисунку 3.19.

```
public record AsyncFireCrackerRequest(Type type, String instanceId, CompletableFuture<Result> result, Object data) {
    public enum Type {  10 usages
        START_INSTANCE,  2 usages
        STOP_INSTANCE,  2 usages
        SUSPEND_INSTANCE,  2 usages
        COLD_START_INSTANCE,  2 usages
        RESET_TIMER  3 usages
    }

    public enum Result {  11 usages
        OK,  3 usages
        DATA_AVAILABLE,  no usages
        DATA_INCOMPLETE,  no usages
    }
}
```

Рисунок 3.19 – Запис AsyncFireCrackerRequest

Розглянемо поля цього запису:

- Type – тип запиту:
    - START_INSTANCE – запустити ВМ
    - STOP_INSTANCE – зупинити ВМ
    - SUSPEND_INSTANCE – призупинити ВМ зі збереженням стану
    - COLD_START_INSTANCE – запустити ВМ, ігноруючи збережений стан
    - RESET_TIMET – скинути таймер автоматичного призупинення
- instanceId – унікальний ідентифікатор ВМ
- result – CompletableFuture через який буде повернено результат
- data – додаткові дані

Для того щоб виконати певну дію потрібно створити об’єкт класу CompletableFuture та відповідний запис і передати його у менеджер через метод sendRequest, що кладе ці записи у чергу. Приклад створення та відправки запиту наведено на риснку 3.20.

```
private CompletableFuture<AsyncFireCrackerRequest.Result> sendServiceThreadRequest(AsyncFireCrackerRequest.Type type,
                                                                                    String instanceId) {
    return this.sendServiceThreadRequest(type, instanceId,  data: null);
}

@SneakyThrows  3 usages
public AsyncFireCrackerRequest.Result startVirtualMachine(String id, boolean discardSnapshot) {
    AsyncFireCrackerRequest.Type type = discardSnapshot
            ? AsyncFireCrackerRequest.Type.COLD_START_INSTANCE
            : AsyncFireCrackerRequest.Type.START_INSTANCE;

    try {
        return this.sendServiceThreadRequest(type, id).join();
    } catch (CompletionException ex) {
        throw ex.getCause();
    }
}
```

Рисунок 3.20 – Створення запиту до менеджеру гіпервізора

Обробка запиту виконується у головному методі run класу FireCrackerServiceThread, де відбувається очікування надходження запиту. Менеджер здійснює обробку запитів по одному за раз, у часових вікнах розміром по 30 секунд, після чого переходить до інших задач (див. пункт 3.5.4).

Менеджер гіпервізора також дозволяє виконувати наступні дії без асинхронних запитів:

- Отримання агенту вводу-виводу (див. пункт 3.5.3)
- Отримання статусу процесу гіпервізора по ідентифікатору ВМ

Процеси зберігаються у потоко-безпечних геш-таблицях, тому додаткова синхронізація для цих дій не потрібна. Остання дія також використовується в компоненті роботи з мережею і має високі вимоги до швидкості відпрацювання.

3.5.2. Об’єкт процесу гіпервізора

На кожну запущену ВМ створюється об’єкт процесу гіпервізора, що відповідає за збір статистики по спожитому ВМ процесорному часу, пересилку команд до гіпервізора, спостереженням за статусом процесу ОС в якому виконується гіпервізор та координацію роботи агенту вводу-виводу.

До переліку дій, які можна виконати з об’єктом відносяться:

- Запуск ВМ
- Зупинка ВМ без збереження стану
- Зупинка ВМ зі збереженням стану
- Отримання агенту вводу-виводу
- Зупника процесу
- Отримання статистики споживання ресурсів

Таймер об’єкту процеса являє собою звичайне число, яке всередині самого процесу не змінюється саме по собі. Зовнішній код має викликати функцію декременту таймеру та передати кількість секунд, що пройшла. Коли значення таймеру опускається нижче 0, об’єкт процесу автоматично виконає послідовність дій для зупинки ВМ зі збереженням стану та завершить процес гіпервізору. Цей таймер можна скинути.

Зовнішній код також має змогу отримати інформацію про спожитий гіпервізором процесорний час, як за весь період роботи процесу, так і за період між останнім та поточним запитом.

3.5.3. Агент вводу-виводу

Об’єкт процесу гіпервізора також підключає до процесу ОС агент ввода-виводу. Агент вводу-виводу реалізований в класі FireCrackerIoAgent, що також призначений для запуску в окремому потоці. Цей об’єкт забирає інформацію, що гіпервізор виводить через стандартний вивід (STDOUT) та дозволяє передати інформацію до стандартного вводу (STDIN).

Всередині об’єкта створюється спеціальний буфер, що зберігає останні 8 КБ даних. Зовнішній код має змогу звернутись до цього буферу, а також підписатись на отримання нових даних чи відправити свої.

Цей об’єкт є необхідним для реалізації функціоналу консолі, до якої можна підключитись у разі пошкодження мережевих налаштувань ВМ. Оскільки гіпервізор перенаправляє все, що виводить ВМ у послідовну консоль

до свого стандартного виводу, таким механізмом можна надати користувачу можливість взаємодіяти з ВМ навіть якщо вона перестане буде доступною ззовні.

Агент вводу-виводу використовується у класі WebSocketConsoleHandler, який обробляє підключення по протоколу WebSockets для взаємодії з консоллю із браузера. Для цього був розроблений невеликий власний протокол, що складається із повідомлень, перелій яких наведено у таблиці 3.3.

Таблиця 3.3 – Повідомлення протоколу консолі

Код	Повідомлення	Напрямок	Дані	
			Розмір	Опис
0x0A	Запит буферу	Клієнт -> Сервер	Відсутні	
0x0B	Зміст буферу	Сервер -> Клієнт	2 байти	Розмір буферу
			Довільний	Зміст буферу
0x0E	Надіслати клавішу	Клієнт -> Сервер	2 байти	Номер знаку
0x0F	Оновлення консолі	Сервер -> Клієнт	2 байти	Номер знаку

Знаки в протоколі кодуються за допомогою кодування UTF-8. Для підключення до консолі по WebSockets потрібно виконати HTTP-запит за URL /vm/tty, а в рядку параметрів передати ідентифікатор ВМ та підпис. Підключитись до консолі можна лише через вузол, на якому вона запущена. Зміст повідомлення передається після коду повідомлення, що займає два байти.

3.5.4. Інші задачі менеджеру гіпервізора

Як було сказано раніше, менеджер гіпервізора отримує та виконує запити в циклі протягом 30 секунд. Кожні 30 секунд він переходить до виконання інших задач, в число яких входить декремент таймерів процесів, опитування їх стану та статистики.

Після отримання стану від процесу менеджер може прийняти рішення звільнити пам’ять, виділену під об’єкт цього процесу. Також стан процесу синхронізується із БД, таким чином стан ВМ актуалізується раз на 30 секунд.

Отримана статистика також використовуєтсья для зміни лічильників спожитого процесорного часу в БД. Стан лічильників також надсилається до аналітичної бази даних ClickHouse з часовою міткою, це дозволяє користувачеві провести аналіз споживання ресурсів вузла або побудувати графік навантаженності ВМ.

3.6. ВМ з виділеною IP-адресою

ВМ з виділеними зовнішніми адресами є доволі розповсюдженими і зазвичай провайдери, за винятком дуже маленьких, пропонують ВМ з однією виділеною адресою без додаткових витрат. Отже, в моніторі ВМ передбачено створення та керування ВМ, що підключаються до мережі з виділеною статичною IP-адресою.

3.6.1. Схема підключення

Припустимо, що у нас є підмережа, яку ми нам видав вищестоящий інтернет-провайдер. За умови наявності можливості розділити цю підмережу на декілька ми можемо створити окрему підмережу, в якій будуть знаходитись IP-адреси для ВМ. Після цього можна налаштувати роутер таким чином, щоб пакети, призначені для цих IP-адрес йшли на окремий фізичний інтерфейс нашого вузла віртуалізації, який ми будемо використовувати для надання доступу віртуальним машинам до мережі Інтернет

На рисунку 3.21 показана приблизна схема підключення. Насправді розбивати підмережі не обов’язково, якщо на роутері дозволяється використання протоколу ARP.

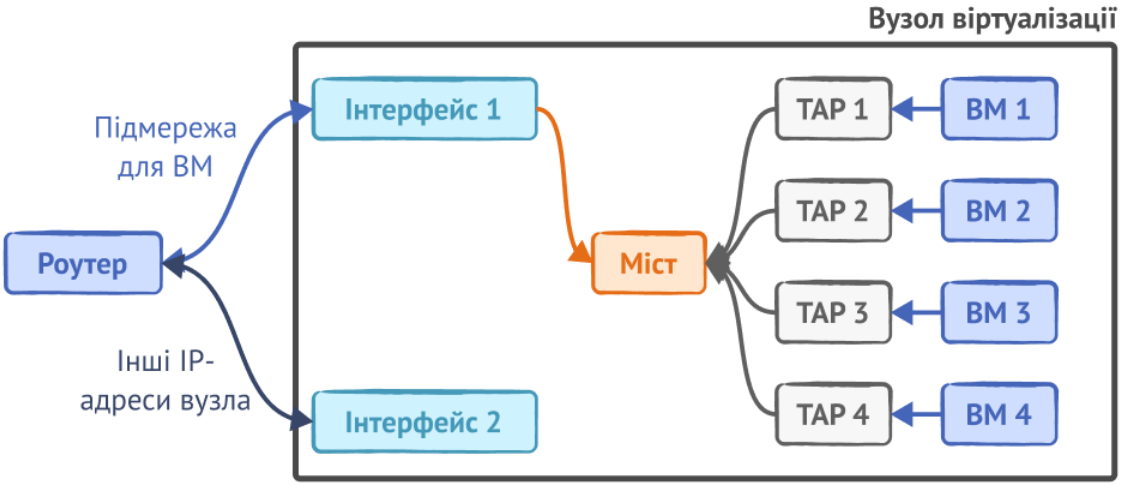


Рисунок 3.21 – Схема підключення віртуальних машин з виділеною адресою

У наданій схемі використовується така сутність як міст (в термінології Linux – bridge). Такі мости являють собою віртуальні комутатори, тобто пристрої, що можуть використовуватись для маршрутизації мережевих пакетів на канальному рівні. До моста можна підключити реальні інтерфейси та віртуальні TAP-пристрої.

Оскільки гіпервізор Firecracker підтримує підключення ВМ до мережі за допомогою TAP-пристроїв, ми можемо розмістити на одному вузлі декілька ВМ з різними IP-адресами, якщо створимо міст, до якого підключемо мережевий інтерфейс та віртуальні машини одночасно.

В такому випадку при отриманні Ethernet-кадру, ядро Linux буде пересилати їх до ВМ через міст, якщо в них в якості отримувача вказана широкомовна MAC-адреса або MAC-адреса, що міститься в базі даних пересилки мосту.

Ядро Linux слідкує за кадрами, що пересилаються по інтерфейсам і автоматично заносить MAC-адреси в базу даних, якщо вони були помічені у кадрах, що проходять через міст. Отже, якщо ми налаштуємо ВМ на відправку якогось пакету під час запуску та роботи, наприклад GARP (Gratuitous ARP), то її MAC-адресу буде додано до БД мосту. Через це збірка Fedora Linux, що

використовується в цьому дипломному проєкті на віртуальних машинах була налаштована на періодичне надсилання GARP-запитів.

Також можна додати запис в БД моста через API чи командний рядок, або взагалі нічого не налаштовувати і покластися на те, що широкомовний ARP-запит спровокує відправку відповіді з ВМ через міст, проте такі рішення мають свої недоліки. Загалом при додаванні вручну ми втрачаємо можливість визначити готовність мережі на ВМ через БД, а широкомовні запити від роутера можуть надходити не завжди. Ця можливість є критичною для забезпечення коректної роботи компоненту, відповідального за автоматичне відновлення роботи ВМ при наявності трафіку.

3.6.2. Детектування мережевого трафіку

Щоб реалізувати технічне завдання, а саме частину про автоматичне відновлення роботи ВМ при наявності мережевого трафіку, потрібно якимось чином реагувати на вхідні пакети.

Для цього можна створити свій драйвер або модуль ядра, але в Linux 2.6.14 з’явилась можливість створювати черги, в які потраплятимуть пакети і на які можна підписатись зі сторони програми. Черги створюються автоматично, якщо в правилах iptables наявні посилання на них, щоб створити таке правило, достатньо використати дію NFQUEUE та передати номер черги через --queue-num. Програми мають змогу знищити пакет в черзі, пропустити його далі або змусити пройти через ланцюжки netfilter ще один раз.

Для наших цілей цього більше ніж достатньо. Конкретно у випадку ВМ з виділеними IP-адресами достатньо створити правило на кожну IP-адресу, що буде ставити вхідні пакети в чергу, яку ми будемо прослуховувати у моніторі ВМ.

Дія NFQUEUE доступна для використання у таблицях filter, mangle, security та raw. Тут доречно буде розмістити правило у ланцюжку PREROUTING таблиці raw, адже відновити роботу ВМ треба до того, як пакет

дійде до етапу маршрутизації, на якому він буде знищений, якщо система не буде знати, куди його направити.

3.6.3. Практична реалізація

Розробку варто почати із обробки черг. Оскільки бібліотеки для роботи з цими об’єктами для Java немає, процедура налаштування прослуховування черги є багатоетапною, а вимоги до швидкодії високі, має сенс розробити цю частину компоненту мовою C.

Обробник черг має бути універсальним та простим, адже його функціонал зводиться до відкриття сокету черги та затримки пакетів, поки не буде ясно, що їх можна або пропустити, або ліквідувати. Для цього варто реалізувати наступні функції:

- net_queue_setup – виконати налаштування черги
- net_queue_destroy – знищити чергу та звільнити пам’ять
- net_queue_run – основна процедура, що починає обробку пакетів

У функцію net_queue_setup можна одразу передати вказівники на функції логування та прийняття рішення, що будуть знаходитись в основній частині програми на мові Java. Функціонал Project Panama як раз дозволяє згенерувати функції, що сумісні із стандартною угодою про виклики і здатні викликати методи, прив’язані до певного об’єкта. Визначення інтерфейсу об’єкту, що містить потрібні методи, надано на рисунку 3.22, а код для передачі вказівників на рисунку 3.23.

```java
public interface QueueWorkerCb {
    void handleLogMessage(@NotNull MemorySegment rawMessage);
    boolean getVerdict();
}
```

Рисунок 3.22 – Інтерфейс об’єкту

```java
public MemorySegment setup(char queue_id, @NotNull QueueWorkerCb cb, Arena arena) {
    var loggerHandle = MethodHandles.lookup().findVirtual(cb.getClass(), name: "handleLogMessage",
            MethodType.methodType(void.class, MemorySegment.class)).bindTo(cb);
    var governorHandle = MethodHandles.lookup().findVirtual(cb.getClass(), name: "getVerdict",
            MethodType.methodType(boolean.class)).bindTo(cb);

    var loggerStub = this.linker.upcallStub(loggerHandle, FunctionDescriptor.ofVoid(ValueLayout.ADDRESS),
            arena);
    var governorStub = this.linker.upcallStub(governorHandle, FunctionDescriptor.of(ValueLayout.JAVA_BOOLEAN),
            arena);

    return (MemorySegment) this.setupHandle.invoke(queue_id, loggerStub, governorStub);
}
```

Рисунок 3.23 – Код методу, що викликає net_queue_setup

Цей метод повертає вказівник на створенний функцією контекст, який можна буде потім передати до net_queue_destroy чи net_queue_run. За допомогою Project Panama виконується і зворотня операція по створенню методів Java, шо викликають нативний код, як це показано на рисунку 3.24.

```java
public NativeQueueLib(Path libPath) {  1 usage
    this.linker = Linker.nativeLinker();
    var lib = SymbolLookup.libraryLookup(libPath, Arena.global());

    var setupDescr = FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_CHAR,
            ValueLayout.ADDRESS, ValueLayout.ADDRESS);
    var dtorDescr = FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);
    var runDescr = FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);

    this.setupHandle = this.linker.downcallHandle(lib.find( name: "net_queue_setup").get(), setupDescr);
    this.dtorHandle = this.linker.downcallHandle(lib.find( name: "net_queue_destroy").get(), dtorDescr);
    this.runHandle = this.linker.downcallHandle(lib.find( name: "net_queue_run").get(), runDescr);
}
```

Рисунок 3.24 – Створення об’єктів-посилань на нативні функції

Для покращення сприйняття коду було створено інші допоміжні методи, що схожі на метод setup(), вони показані на рисунку 3.35.

```
@SneakyThrows  1 usage
public void run(MemorySegment context) { this.runHandle.invoke(context); }

@SneakyThrows  2 usages
public void cleanup(MemorySegment context) { this.dtorHandle.invoke(context); }
```

Рисунок 3.25 – Допоміжні методи-викликачі

Як видно на рисунку 3.34, клас-обгортка NativeQueueLib завантажує бібліотеку по шляху, що передається у параметр конструктора libPath. При запуску монітору ВМ виконується копіювання файлу з бібліотекою із JAR-архіву у директорію тимчасових файлів. На жаль, напряму завантажити бібліотеку у пам’ять з JAR-ресурсу наразі не можна.

Для компіляції програми використовується компілятор clang, який викликається із сценарію збірки Gradle. Релевантні частини коду показані на рисунку 3.26.

```
val compileNative = tasks.register<Exec>( name: "compileNative") {
    inputs.file(nativeSourceDir.resolve( relative: "queue_worker.c"))
    outputs.file(nativeBuildDir.resolve( relative: "queue_worker.so"))

    doFirst {
        nativeBuildDir.mkdirs()
    }

    commandLine( ...arguments:
        "clang",
        "-shared",
        "-fPIC",
        "-Ofast",
        "-lnetfilter_queue",
        "-std=c2y",
        "-o", nativeBuildDir.resolve( relative: "queue_worker.so").absolutePath,
        nativeSourceDir.resolve( relative: "queue_worker.c").absolutePath
    )
}

val copyNativeToResources = tasks.register<Copy>( name: "copyNativeToResources") {
    dependsOn(compileNative)
    from(nativeBuildDir)
    include( ...includes: "queue_worker.so")
    into( destDir: "src/main/resources/native")
}

tasks.processResources {
    dependsOn(compileNative)
    dependsOn(copyNativeToResources)
}
```

Рисунок 3.26 – Фрагмент сценарію збірки

Сам код підпрограми на С є простим, тому не буде наведений цілком. Із особливостей реалізації можна відмітити, що застосовано режим копіювання пакетів NFQNL_COPY_META з метою покращення швидкодії, адже для роботи цього компоненту нема потреби копіювати пакет разом з його змістом. Це витратна операція, а зміст при цьому не аналізується, як можна побачити на рисунку 3.27.

```
static int onPacketReceived(struct nfq_q_handle *qh,
                            struct nfgenmsg *nfmsg, struct nfq_data *nfa,
                            void *ctx_ptr) {
    net_queue_ctx_t *ctx = ctx_ptr;
    int verdict = ctx->get_verdict() ? NF_ACCEPT : NF_DROP;
    struct nfqnl_msg_packet_hdr *ph = nfq_get_msg_packet_hdr(nfa);
    if (!ph)
        return nfq_set_verdict(ctx->qh, 0, NF_ACCEPT, 0, NULL);

    uint32_t id = ntohl(ph->packet_id);
    return nfq_set_verdict(ctx->qh, id, verdict, 0, NULL);
}
```

Рисунок 3.27 – Код функції-обробника пакетів

Реалізація методів handleLogMessage та getVerdict є у класі QueueWorker. Об’єкти цього класу створюються на кожну ВМ та оброблюють пакети із черг, що призначені для неї. Оскільки функція net_queue_run містить нескінечний цикл, то варто працювати з нею в окремому потоці. Тому QueueWorker також реалізує інтерфейс Runnable, де і виконується ця функція. Лістинг реалізованих методів наведено на рисунку 3.28.

```java
@Override 1 usage
public void handleLogMessage(@NotNull MemorySegment rawMessage) {
    String log = rawMessage.reinterpret( newSize: 255).getUtf8String( offset: 0);
    this.logger.debug("Message from worker: {}", log);
}

@Override 1 usage
public boolean getVerdict() {
    if (this.virtualizationService.getVmmProcessStatus(vmId) == ProcessStatus.RUNNING) {
        this.virtualizationService.resetTimerAsync(vmId);
        return true;
    }

    try {
        this.logger.trace("{} down, need to boot", vmId);
        var quick = this.virtualizationService.isQuickStartAvailable(vmId);
        this.virtualizationService.startVirtualMachine(vmId, discardSnapshot: false);

        // If quick start was not available, we should give VM some time to boot
        if (!quick)
            this.networkingService.waitForNetworkAvailability(vmId);

        this.logger.trace("VM {} up", vmId);
    } catch (ForeignResourceException | UnknownVmException | ResourceLockedException | TapCreationException
             | IllegalStateException ignored) {
        return false;
    }

    return true;
}

@Override
public void run() {
    this.queueCtx = this.queueLib.setup(this.queueId, cb: this, Arena.ofAuto());
    if (this.queueCtx.address() == 0)
        throw new QueueManagementException("Queue context not set");

    try {
        this.queueLib.run(this.queueCtx);
    } finally {
        this.queueLib.cleanup(this.queueCtx);
    }
}
```

Рисунок 3.28 – Лістинг методів handleLogMessage, getVerdict та run

Як можна побачити, при виконанні функції run, об’єкт класу QueueWorker прив’язує себе до контексту нативної бібліотеки та починає обробляти пакети із черги, пропускаючи їх у випадку, коли ВМ запущена. У випадку коли ВМ вимкнена, її робота відновлюється, а якщо ВМ потрібно буде увімкнути «з нуля», пакет затримується доки не ВМ не з’явиться в мережі. Такий підхід дозволяє запобігти втраті пакетів у час, коли ВМ запущена, але ще не готова їх приймати.

Для віртуальних машин з виділеною IP-адресою очікування появи в мережі виконується в службі NetworkingService за допомогою циклічного опитування БД мосту, як показано на рисунку 3.29.

```
var mac = vm.getMacAddressStr();
var br = vm.getIp().getBridge();
var cmd = STR."bridge fdb show br \{br} | grep -i \{mac}";
var proc = Runtime.getRuntime().exec(new String[]{"bash", "-c", cmd});
if (proc.waitFor() == 0)
    break;

this.logger.debug("{} for {}'s MAC ({}) to appear in {}'s FDB", i < 2 ? "Waiting" : "Still waiting",
        vm.getId(), mac, br);
Thread.sleep( millis: 100);
```

Рисунок 3.29 – Опитування БД мосту

Створення правил iptables відбувається у класі QueueManager, там же і виконується диспетчерізація обробників черг QueueWorker. Правила створюються для кожної активованої ВМ при запуску монітору, а також динамічно у разі появи нових віртуальних машин. Коли ВМ видаляється, асоційований з нею QueueWorker знищується. Номери черг починаються з 1000, спеціальний boolean-масив використовується для збереження інформації про те, які номери черг вільні для використання (див. рисунок 3.30).

```
private char getAvailableQueue() throws QueueManagementException {  2 usages
    for (char i = 0; i < queuePool.length; i++)
        if (!queuePool[i])
            return (char) (1000 + i);

    throw new QueueManagementException("No available queues left!");
}
```

Рисунок 3.40 – Функція, що повертає вільний номер черги

Створення TAP-інтерфейсів відбувається у службі NetworkingService, а саме у методі bringUpNetwork. Оскільки максимальна довжина імені інтерфейсу в Linux не може перевищувати 15 байт, а ідентифікатори ВМ більші за це значення, то імена TAP інтерфейсів формуються за схожим із чергами принципом. Створений TAP інтерфейс автоматично під'єднується до мосту, що було зазначено при створенні IP-адреси (див. рисунок 3.31).

```
String bridge;
if (vm.getNetworkType() == VirtualMachine.NetworkType.SHARED_IP) {
    this.ensureKeaRunning();
    bridge = vm.getDynamicPool().getBridge();
} else if (vm.getNetworkType() == VirtualMachine.NetworkType.DEDICATED_IP) {
    bridge = vm.getIp().getBridge();
} else {
    this.logger.error("Unknown network type: " + vm.getNetworkType() + ", falling back to master bridge CMBR");
    bridge = "cmbr";
}

proc = Runtime.getRuntime().exec(new String[]{"ip", "link", "set", "ctap" + tap, "master", bridge});
rc = proc.waitFor();
if (rc != 0)
    throw new TapCreationException("{ip link set ctap" + tap + " master " + bridge + "} failed with rc=" + rc);
```

Рисунок 3.31 – Підключення TAP-інтерфейсу до мосту

3.7. ВМ без виділеної IP-адреси

У протоколі IPv4 адреси являють собою 32-бітне число, що сильно обмежує максимально можливу кількість адрес, що можуть бути використані для маршрутизації в Інтернеті. Хоча зараз існує протокол IPv6, адреси в якому значно більше (128 замість 32 біт), його використання не завжди є доречним: наприклад, у деяких країнах Африки та східної Європи впровадження цієї версії протоколу залишається невеликим [31]. Через це попит на IP-адреси 4 версії продовжує бути великим, а в умовах їх дефіциту зростає і ціна їх аренди.

Зазвичай гостинг-провайдери разом із ВМ видають клієнтам також і виділену IP-адресу для неї і це негативно впливає на ціну послуг, для маленьких ВМ вартість адреси може становити більше третини повної вартості. Проте не у всіх є потреба у виділеній IP-адресі, в деяких випадках наявність певної кількості відкритих портів, що доступні по IPv4 буде достатньо, а іноді можна обійтись і без виділеного порту, якщо доступна маршрутизація на прикладному рівні, наприклад, за символьною адресою, яку можна визначити по HTTP-заголовкам чи SNI (Server Name Indication) з пакету ClientHello.

З метою покрити ці сценарії використання, у розробленому моніторі ВМ передбачена можливість створювати ВМ без виділеної IP-адреси. Таким ВМ можна призначити окремі порти на спільних IP-адресах, а також налаштувати правила переадресації трафіку за заголовками HTTP та пакетам ClientHello. Такі ВМ можна об'єднувати у локальні мережі.

3.7.1. Схема підключення

На рисунку 3.32 наведено узагальнену схему підключення ВМ без виділеної адреси. Об'єднання ВМ у локальні мережі відбувається за допомогою мостів: TAP-інтерфейси машин підключаються до мосту, на кожну локальну мережу створюється один міст. На відміну від схеми, що використовується для ВМ з виділеною адресою, ці мости мають також і власну адресу, що знаходиться в одній підмережі з локальними адресами ВМ. Локальні адреси призначає DHCP-сервер, що прослуховує запити на мостах.

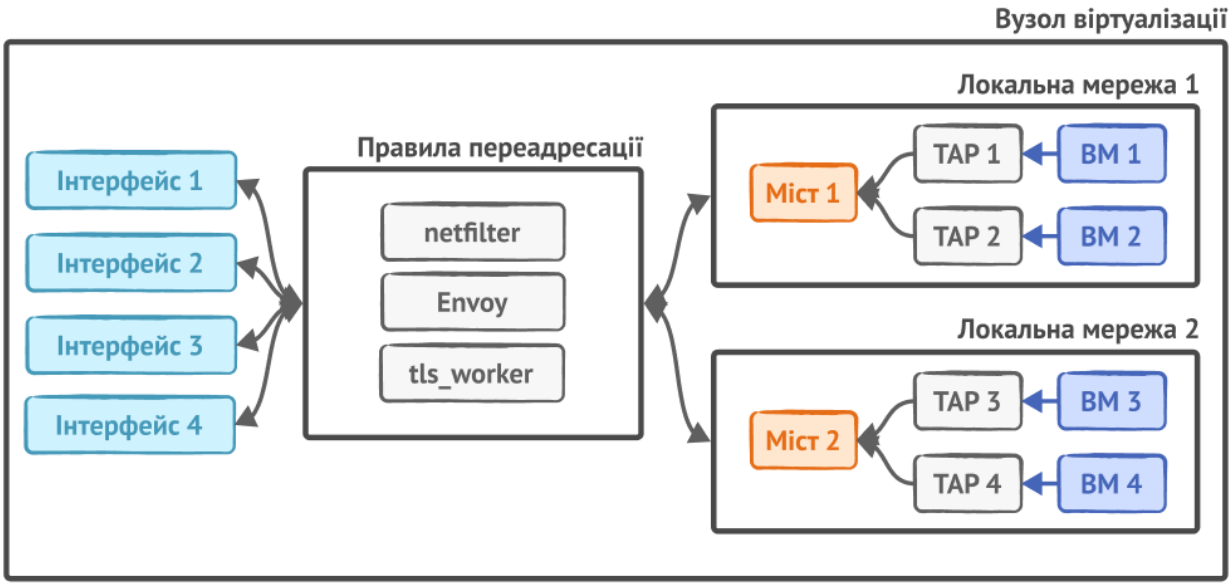


Рисунок 3.32 – Схема підключення віртуальних машин без виділеної адреси

Також на відміну від попередньої схеми, у цій до мостів не підключаються зовнішні інтерфейси, адже ці ВМ не мають безпосередньо взаємодіяти з зовнішньою мережею, натомість трафік до них маршрутизується

всередені вузла віртуалізації за допомогою механізму netfilter, проксі-серверу Envoy або компонента tls_worker.

При запуску ВМ без наявного образу оперативної пам’яті, для перевірки доступності ВМ в мережі, додатково використовується опитування БД DHCP-серверу.

3.7.2. Реалізація роботи з DHCP-сервером

У цій дипломній роботі було використано DHCP-сервер Kea, налаштований на зберігання БД адрес у вигляді CSV-файлу. При створенні або редагуванні локальної мережі, служба NetworkingService використовує JTE-шаблон kea-config.jte для створення JSON-конфігурації DHCP-серверу та перезавантажує його.

Оскільки для створення правил та конфігурації проксі-серверу треба знати локальну IP-адресу ВМ, було також реалізовано спеціальний клас DhcpObserver, який зчитує БД адрес при зміні, зберігає її кеш в пам’яті програми та генерує події, на які компоненти можуть реагувати. Спрощена схема роботи представлена на рисунку 3.33.

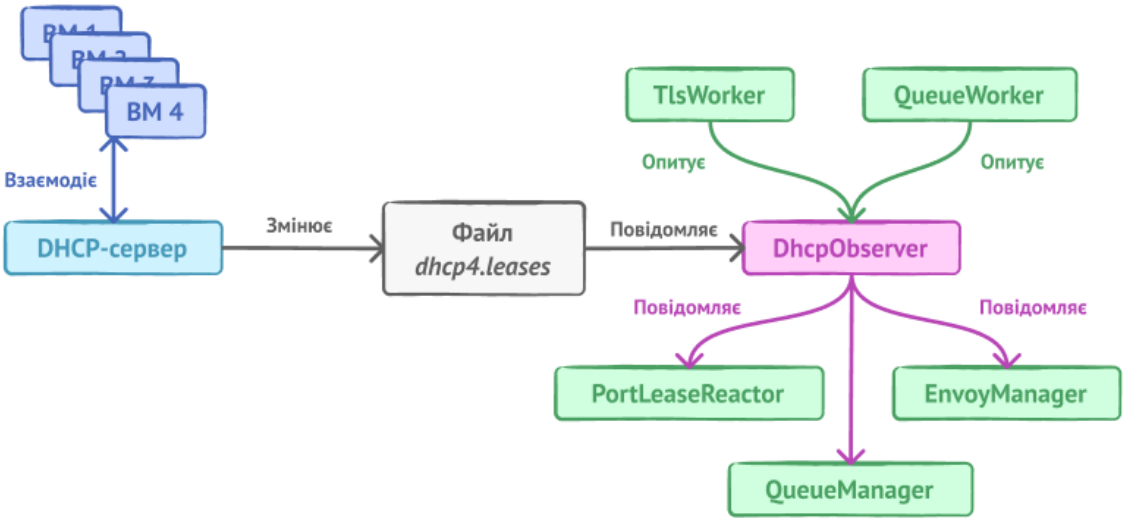


Рисунок 3.33 – Схема роботи DhcpObserver

Окрім подій, взаємодіяти з DhcpObserver можна також за допомогою наступних методів:

- activeLeaseExists – перевірити, чи видана зараз IP-адреса ВМ за певним ідентифікатором
- waitForNewLease – повертає CompletableFuture, що завершується коли DHCP-сервер видасть нову IP-адресу певній ВМ

Обидва методи є потокобезпечними. DhcpObserver також реалізує інтерфейс Runnable і запускається в окремому потоці службою NetworkingService.

3.7.3. Робота з виділеними портами

В розробленому застосунку передбачено можливість призначення портів спільних зовнішніх IP-адрес індивідуальним віртуальним машинам, при цьому номер порту на ВМ може не співпадати із номером зовнішнього порту. Щоб закріпити порт за ВМ, користувачеві треба вказати протокол (підтримуються протоколи TCP, UDP, SCTP, DCTP) та пару портів, зовнішній та внутрішній. Можна створити декілька правил, що пересилатимуть трафік на один внутрішній порт ВМ, але для кожного зовнішнього порту одночасно може існувати лише одне правило. Маніпулювати таким чином можна лише такими портами, що знаходяться на IP-адресах вузла, що визначені як спільні, та які не є зарезервованими.

Клас QueueManager динамічно змінює конфігурацію netfilter та додає правила коли порт закріплюють за ВМ або відкріплюють. Черга при цьому залишається одна на кожну ВМ, але замість одного правила може бути декілька, що посилаються на одну і ту ж саму чергу. На відміну від ВМ з виділеною IP-адресою, в цьому випадку до предикатів правил належить не лише IP-адреса, а ще й порт.

Механізм обробки пакетів залишається тим самим, використовується той самий QueueWorker, що і для ВМ з виділеною адресою. Різниця полягає у тому, що виклик NetworkingService::waitForNetworkAvailability для ВМ такого

типу викличе DhcpObserver::waitForNewLease, якщо за ВМ ще не закріплена ніяка адреса.

DhcpObserver::waitForNewLease завершить виконання, коли DHCP-сервер призначить IP-адресу ВМ, але не до того, як виконаються всі обробники події DhcpLeaseUpdatedEvent, зокрема обробник PortLeaseReactor::onDhcpLeaseUpdated.

Клас PortLeaseReactor відповідає за налаштування правил переадресації трафіку, а його метод onDhcpLeaseUpdated оновлює правила, щоб вони містили актуальну IP-адресу ВМ. Оскільки подія DhcpLeaseUpdatedEvent синхронна, то на момент виходу із методу NetworkingService::waitForNetworkAvailability в QueueWorker гарантовано буде працювати пересилка трафіку між портами.

PortLeaseReactor організовує правила в ланцюжки, які додаються до ланцюжку FORWARD таблиці mangle, що дозволяє швидко їх видаляти у разі деактивації або знищення ВМ користувачем.

3.7.4. Зарезервовані порти

Порти в спільних IP-адресах можна зарезервувати. Такі порти не можна буде призначити віртуальним машинам. При цьому можна дозволити спільне використання таких портів сервісами, що працюють за протоколом HTTP або TLS.

3.7.5. HTTP-маршрутизація

Для спільного використання порту, який було зарезервовано для HTTP-маршрутизації, передбачено окремий функціонал, що дозволяє створити правило переадресації HTTP-запитів за заголовком Host на певний порт ВМ.

Маршрутизацію виконує проксі-сервер Envoy, робота з яким реалізована в класі EnvoyManager, що спостерігає за процесом та реагує на події, зокрема на події про оновлення IP-адреси ВМ. Цей реагує на зміни в системі та генерує конфігурацію проксі-серверу в залежності від них.

У випадку коли ВМ вимкнена, усі запити до закріпленої за нею символьною адресою переадресуються до монітору ВМ. При отриманні такого запиту, монітор вмикає або відновлює роботу ВМ та відправляє відповідь із HTTP-перенаправленням на сторінку очікування. Код на сторінці очікування опитує монітор ВМ через спеціальний API та повертається до початкової адреси після того, як дізнається, що ВМ активна.

Сторінка очікування та спеціальний API реалізовані у контролері ShoreKeeperController, лістинг якого наведено на рисунку 3.34.

```
@RequestMapping({"/ShoreKeeperCapture", "/ShoreKeeperCapture/**"})
public String handleCapture() { return "redirect:/shorekeeper-cgi/WaitingRoom.pl"; }

@GetMapping("/ShoreKeeper/{rayId}/WaitingRoom.pl")
public String waitingRoom(@RequestParam String rayId) {
    var rule = this.forwardingService.getForwardRuleById(rayId).orElseThrow(() ->
            new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Misdicrected request (invalid ray)"));

    var vmId = rule.getVirtualMachine().getId();
    if (this.dhcpObserver.activeLeaseExists(vmId))
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Oops!");

    try {
        if (this.virtualizationService.getVmmProcessStatus(vmId) == ProcessStatus.NOT_LOADED)
            this.virtualizationService.startVirtualMachine(vmId, discardSnapshot: false);
    } catch (UnknownVmException | ResourceLockedException | ForeignResourceException ex) {
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Hypervisor error: " + ex.getMessage());
    }

    return "shoreKeeperWaitingRoom";
}

@GetMapping("/ShoreKeeper/{rayId}/VmAvailable.pl")
public @ResponseBody Boolean vmStatus(@PathVariable String rayId) {
    var rule = this.forwardingService.getForwardRuleById(rayId).orElseThrow(() ->
            new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Misdicrected request (invalid ray)"));

    return this.dhcpObserver.activeLeaseExists(rule.getVirtualMachine().getId());
}
```

Рисунок 3.34 – Лістинг контролеру сторінки очікування

Для реалізації такої схеми роботи в згенерованій конфігурації Envoy завжди залишається виключення для URL, що починаються з /shorekeeper-cgi/, запити на них внутрішньо переадресовуються до монітору ВМ. Це необхідно, адже API має проінформувати код на сторінці очікування про те, що ВМ активна вже після того, як внаслідок цього буде перезавантажена

конфігурація. Релевантний фрагмент шаблону конфігурації Envoy наведено на рисунку 3.35.

```
                virtual_hosts:
@for (var mappingId : mappings.keySet())
!{var mapping = mappings.get(mappingId);}
!{if (!mapping.inIp().equals(bind.getValue0())) continue;}
            - name: vh_${mappingId}
              domains: ["${mapping.sName()}"]
              routes:
              - match: { prefix: "/shorekeeper-cgi/" }
                route:
                  cluster: shorekeeper
                  prefix_rewrite: "/ShoreKeeper/${mappingId}/"
              - match: { prefix: "/" }
                route:
@if (mapping.outIp() != null)
                  cluster: fwd_${mappingId}
@else
                  cluster: shorekeeper
                  prefix_rewrite: "/ShoreKeeperCapture/${mappingId}/"
@endif
@endfor
```

Рисунок 3.35 – Фрагмент конфігурації проксі-серверу Envoy

Скидання таймеру автоматичного призупинення роботи виконується через правила netfilter за механізмом, схожим на той, що був описаний раніше. Проте в цьому випадку обробник черги не вмикає ВМ, це робить монітор самостійно по HTTP-запиту. Обробник черги лише скидує таймер і дозволяє пакету пройти далі по системі. Правила ставлять пакети у відповідну чергу, якщо в них адреса призначення співпадає з внутрішньою адресою ВМ.

3.7.6. TLS-маршрутизація

Задля можливості використання спільних портів для протоколів, що працюють поверх TLS, зокрема HTTPS без необхідності передавати до монітора ВМ приватні ключі сертифікатів, було передбачено функціональність по маршрутизації трафіку за символьною адресою, яку можна отримати з поля SNI пакету ClientHello, який відправляється під час TLS-рукостискання.

Для реалізації цього було розроблено нативну програму, що використовує бібліотеку libuv для створення обробників TCP-підключень до спільних портів. При отриманні пакету ClientHello, ця програма відкриває

підключення до ВМ та починає перенаправлення усього трафіку між відкритими сокетами.

Скидання таймерів виконується програмою самостійно за допомогою виклику відповідного методу через трамплін, згенерований на стороні Java за допомогою Project Panama. Отже, в цьому випадку правила netfilter не використовуються.

Геш-таблиця, що співставляє символьні адреси із IP-адресами ВМ реалізована через об’єкт GHashTable, доступний у бібліотеці GLib, та поповнюється по запитам через функції, що теж викликаються через Project Panama, тільки вже зі сторони Java-коду. Для синхронізації доступу використовується GMutex.

3.8. Висновки до розділу 3

У рамках дипломного проєкту було розроблено програмний комплекс для моніторингу та керування віртуальними машинами, що дозволяє автоматизувати їх створення, зупинку, та запуск залежно від мережевої активності. Архітектура комплексу реалізована на базі фреймворку Spring Boot із використанням Java та C для високопродуктивних компонентів, що взаємодіють із мережею.

Монітор підтримує роботу в кластерному середовищі, має модульну структуру з чітким поділом компонентів за функціональним призначенням та реалізує шаблон MVC з використанням DTO для API. Застосунок інтегрує систему подій Spring для слабко зв’язаної взаємодії між компонентами, що забезпечує ефективну реакцію на зміни стану ВМ та мережевих інтерфейсів.

Було реалізовано зберігання конфігурації в реляційній базі даних із чіткою схемою та системою обмежень, що підвищує надійність і унеможливлює збереження неконсистентних даних. У схемі бази враховано складні взаємозв’язки між ВМ, IP-адресами, портами, правилами маршрутизації тощо.

Таким чином, створено комплексне рішення, здатне забезпечити надійне управління ресурсами у віртуалізованому середовищі з високим рівнем автоматизації, контролю та адаптивності до змін навантаження.

4. ТЕСТУВАННЯ

Тестування є завершальним етапом розробки програмного продукту, який відіграє ключову роль у забезпеченні його надійності, стабільності та відповідності поставленим вимогам. Цей процес є не лише важливим, а й необхідним, оскільки дозволяє виявити та усунути помилки або недоліки ще до впровадження системи в експлуатацію.

У цьому розділі розглядається процес тестування монітору ВМ, що здійснюється шляхом взаємодії з його програмним інтерфейсом (API) та створеними ВМ у межах кількох практичних сценаріїв використання.

Зокрема, ці сценарії включають в себе тестування ВМ із виділеною IP-адресою, а також ВМ, що використовують спільну IP-адресу. Такий підхід дозволяє перевірити здатність розробленого програмного продукту коректно керувати гіпервізором та налаштуваннями системи у різних мережевих конфігураціях ВМ.

4.1. Опис середовища та засобів тестування

Для тестування було використано сервер, на якому було створено ВМ з наступними характеристиками:

- Процесор: 8 ядер, з підтримкою AMD-V через вкладену віртуалізацію
- Оперативна пам’ять: 8 ГБ
- Накопичувач: 128 ГБ NVMe-диск
- ОС: Oracle Linux 9

На сервері було створено міст з IP-адресою 192.168.122.1/24, до якого було підключено цю ВМ, після чого на неї було встановлено інтерпретатор мови програмування Java IBM Semeru та інше необхідне програмне забезпечення. Схему підключеня надано на рисунку 4.1.

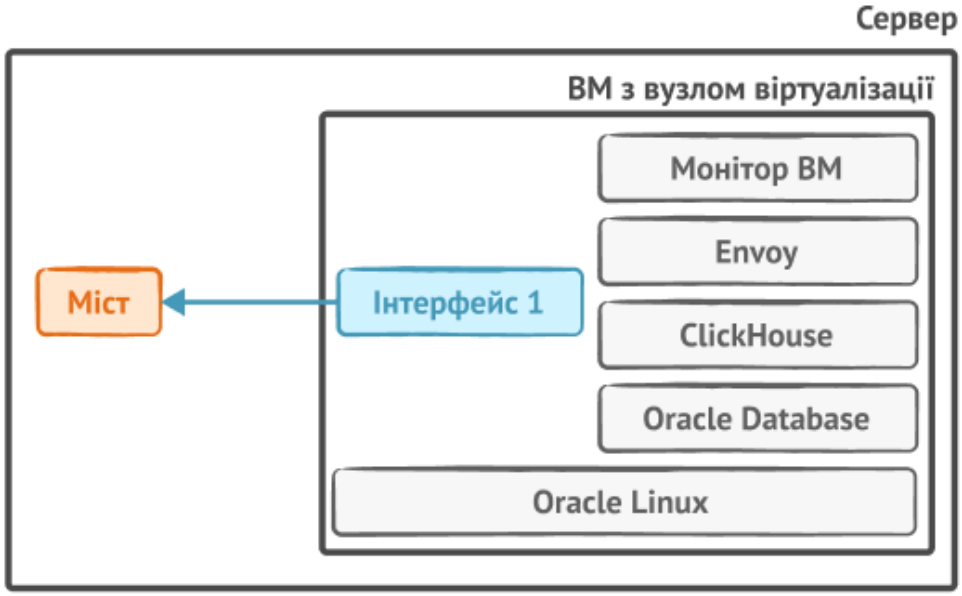


Рисунок 4.1 – Схема підключення

Тестування проводилось за допомогою утілити для тестування та документації API Yaak [34], власних скриптів JS (JavaScript), спостереження за логами монітору ВМ та поведінкою створених ним віртуальних машин. Підключення до монітору ВМ виконувалось через SSH-тунелювання, в якості роутеру було використано сервер, тому взаємодія зі створеними ВМ проводилась на ньому.

Такий підхід дозволив зекономити ресурси під час тестування, особливо IP-адреси, але не до кінця схожий на реальний сценарій використання. Проте для потреб цієї дипломної роботи, тестування таким чином не має завадити покриттю тестами значної частини функціоналу розробленого монітору ВМ.

4.2. Тестування ВМ з виділеною IP-адресою

Для початку налаштуємо мережу на вузлі віртуалізації, для цього можна одразу створити на ньому міст br_master, до якого будуть підключатись ВМ, та призначити йому IP-адресу 192.168.122.138. Ця IP-адреса буде належати вузлу та використовуватись для підключення до нього, зокрема для роботи з API монітору ВМ.

```
[root@localhost ~]# ip link add name br_master type bridge
[root@localhost ~]# ip a a 192.168.122.138/24 dev br_master
[root@localhost ~]# ip link set dev ens8 master br_master
[root@localhost ~]# ip r a 192.168.122.1 dev br_master
[root@localhost ~]# ip r a default via 192.168.122.1 dev br_master
```

Рисунок 4.2 – Створення мосту br_master

На рисунку 4.2 зображена послідовність команд, яка була виконана для створення мосту, присвоєння йому IP-адреси та підключення його до мережевого інтерфейсу, також було налаштовано IP-адресу шлюзу.

Після цього запустимо сервер БД та монітор ВМ. Коли монітор ВМ завершить ініціалізацію, потрібно буде додати адресу, яка буде використана для тестової ВМ. Зробити це можна за допомогою API-запиту, який показано на рисунку 4.3.

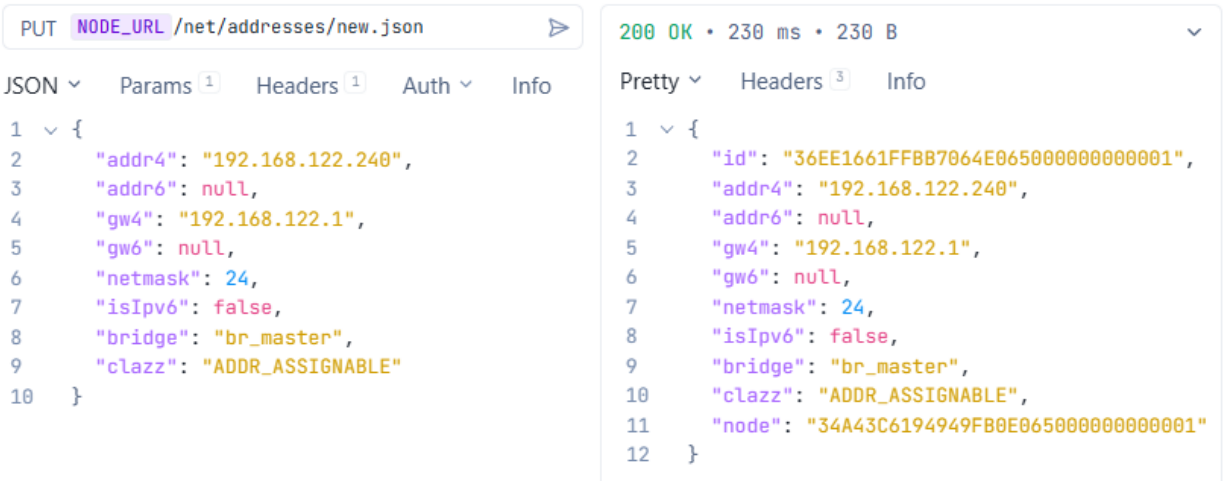


Рисунок 4.3 – Результат виконання запиту створення IP-адреси

В тіло запиту передаються налаштування IP-адреси. В цьому випадку було встановлено клас IP-адреси у ADDR_ASSIGNABLE, бо цю адресу ми будемо призначувати ВМ. Також було передано ім’я мосту, який було створено раніше, маску підмережі та адресу шлюзу. В результаті було отримано копію налаштувань разом із унікальним ідентифікатором адреси. Перевіримо, що ця адреса справді збереглась, виконавши API-запит на отримання списку адрес (див. рисунок 4.4).

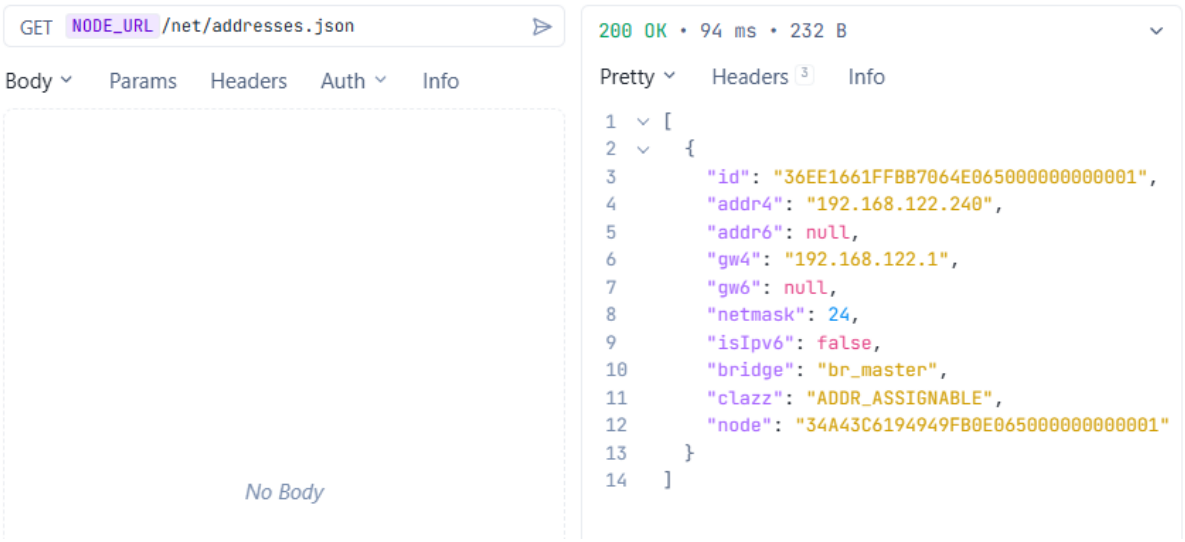


Рисунок 4.4 – Результат отримання запису IP-адреси по API

Як можна побачити, IP-адресу було збережено. Тепер можна перейти до створення тестової ВМ. На рисунку 4.5 зображено API-запит, що створює ВМ.

Рисунок 4.5 – Створення віртуальної машини

В результаті було отримано унікальний ідентифікатор ВМ, якій було виділено 2 ГБ оперативної пам’яті, 2 ГБ місця на диску та 4 ядра процесору. В моніторі ВМ також передбачена можливість зміни конфігурації вже створеної

ВМ, перевіримо це, продублювавши цей запит, але на шлях до конфігурації цієї ВМ (див. рисунок 4.6) та з іншим значенням TTL.

PUT NODE_URL /vm/36EE1661FFBC7064E065000000000001/config.json

JSON Params Headers 1 Auth Info

```
{
  "config": {
    "coreCount": 4,
    "memorySize": 2048,
    "rootDiskSize": 2048,
    "processorType": "MATCH_HOST",
    "networkType": "DEDICATED_IP",
    "ipAddressId": "36EE1661FFBB7064E065000000000001",
    "sensitivity": "IN_OUT_PACKETS",
    "ttl": 60
  },
  "password": "test"
}
```

200 OK • 493 ms • 629 B

Pretty Headers 3 Info

```
{
  "id": "36EE1661FFBC7064E065000000000001",
  "state": "HYPERVISOR_FAULT",
  "processorType": "MATCH_HOST",
  "coreCount": 4,
  "memorySize": 2048,
  "rootDiskSize": 2048,
  "networkType": "DEDICATED_IP",
  "ip": {
    "id": "36EE1661FFBB7064E065000000000001",
    "addr4": "192.168.122.240",
    "addr6": null,
    "gw4": "192.168.122.1",
    "gw6": null,
    "netmask": 24,
    "isIpv6": false,
    "bridge": "br_master",
    "clazz": "ADDR_ASSIGNABLE",
    "node": "34A43C6194949FB0E065000000000001"
  },
  "dynamicPool": null,
  "macAddress": "FIWk",
  "ttl": 60,
  "sensitivity": "IN_OUT_PACKETS",
  "totalRunTime": 0,
  "cpuRunTime": 0,
  "macAddressStr": "00:c0:4c:14:85:a4",
  "processorTypeStr": "None",
  "node": "34A43C6194949FB0E065000000000001"
}
```

Рисунок 4.6 – Зміна конфігурації ВМ

Як можна побачити за відповіддю від серверу, конфігурацію було успішно змінено. Зараз ВМ залишається у стані STOPPED, який означає, що ВМ не запущена, але може бути автоматично увімкнена. Спробуємо запустити її за допомогою ping-запиту на її IP-адресу.

```
[root@ne41763 ~]# ping 192.168.122.240
PING 192.168.122.240 (192.168.122.240) 56(84) bytes of data.
64 bytes from 192.168.122.240: icmp_seq=1 ttl=64 time=14690 ms
64 bytes from 192.168.122.240: icmp_seq=2 ttl=64 time=13666 ms
64 bytes from 192.168.122.240: icmp_seq=3 ttl=64 time=12642 ms
64 bytes from 192.168.122.240: icmp_seq=4 ttl=64 time=11620 ms
64 bytes from 192.168.122.240: icmp_seq=5 ttl=64 time=10596 ms
64 bytes from 192.168.122.240: icmp_seq=6 ttl=64 time=9573 ms
64 bytes from 192.168.122.240: icmp_seq=7 ttl=64 time=8549 ms
64 bytes from 192.168.122.240: icmp_seq=8 ttl=64 time=7536 ms
64 bytes from 192.168.122.240: icmp_seq=9 ttl=64 time=6513 ms
64 bytes from 192.168.122.240: icmp_seq=10 ttl=64 time=5489 ms
64 bytes from 192.168.122.240: icmp_seq=11 ttl=64 time=4465 ms
64 bytes from 192.168.122.240: icmp_seq=12 ttl=64 time=3441 ms
64 bytes from 192.168.122.240: icmp_seq=13 ttl=64 time=2418 ms
64 bytes from 192.168.122.240: icmp_seq=14 ttl=64 time=1394 ms
64 bytes from 192.168.122.240: icmp_seq=15 ttl=64 time=370 ms
64 bytes from 192.168.122.240: icmp_seq=16 ttl=64 time=1.51 ms
64 bytes from 192.168.122.240: icmp_seq=17 ttl=64 time=1.87 ms
64 bytes from 192.168.122.240: icmp_seq=18 ttl=64 time=1.03 ms
64 bytes from 192.168.122.240: icmp_seq=19 ttl=64 time=1.37 ms
```

Рисунок 4.7 – Запуск ВМ через відправку пакету на неї

Монітор ВМ успішно перехопив мережевий пакет та запустив ВМ, для якої цей пакет був призначений. Втрат пакетів не було, адже відповіді починаються з icmp_seq=1, а отже, монітор ВМ коректно визначає момент доступності мережі на ВМ, принаймні для цього шаблону ОС. Як можна побачити за виводом утиліти ping, запуск зайняв приблизно 15 секунд.

Тепер можна перевірити можливість підключення до ВМ за прикладним протоколом, наприклад SSH (див. рисунок 4.8).

```
[root@ne41763 ~]# ssh 192.168.122.240
root@192.168.122.240's password:
Last login: Fri Jun  6 23:26:30 2025 from 192.168.122.1
root@test-vm-2:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:c0:4c:14:85:a4 brd ff:ff:ff:ff:ff:ff
    altname enx00c04c1485a4
    inet 192.168.122.240/24 brd 192.168.122.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::2c0:4cff:fe14:85a4/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
root@test-vm-2:~# screenfetch
           /:-------------:\          root@test-vm-2.local
        :-------------------::        OS: Fedora
      :-----------/shhOHbmp---:\      Kernel: x86_64 Linux 6.14.5-300.fc42.x86_64
    /-----------omMMMNNNMMD  ---:     Uptime: 5m
   :-----------sMMMMNMNMP.    ---:    Packages: 614
  :-----------:MMMdP-------    ---\   Shell: bash 5.2.37
 ,------------:MMMd--------    ---:   Disk: 1.7G / 3.7G (51%)
 :------------:MMMd-------    .---:   CPU: AMD EPYC @ 4x 3.593GHz
 :----    oNMMMMMMMMMNho     .----:   RAM: 338MiB / 1956MiB
 :--     .+shhhMMMmhhy++   .------/
 :-    -------:MMMd--------------:
 :-   --------/MMMd-------------;
 :-    ------/hMMMy------------:
 :-- :dMNdhhdNMMNo------------;
 :---:sdNMMMMNds:------------:
 :------::://:-------------::
 :---------------------://

root@test-vm-2:~# ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=60 time=3.11 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=60 time=2.93 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=60 time=3.50 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=60 time=3.13 ms
64 bytes from 1.1.1.1: icmp_seq=5 ttl=60 time=3.00 ms
64 bytes from 1.1.1.1: icmp_seq=6 ttl=60 time=2.64 ms
64 bytes from 1.1.1.1: icmp_seq=7 ttl=60 time=2.73 ms
64 bytes from 1.1.1.1: icmp_seq=8 ttl=60 time=2.87 ms
64 bytes from 1.1.1.1: icmp_seq=9 ttl=60 time=3.35 ms
64 bytes from 1.1.1.1: icmp_seq=10 ttl=60 time=2.91 ms
```

Рисунок 4.8 – Підключення до ВМ по SSH

Як можна побачити, ВМ була створена коректно, їй призначено 4 ядра та 2 ГБ оперативної пам’яті. На інтерфейсі eth0 наявна коректна IP-адреса, з ВМ доступна зовнішня мережа.

Перевіримо збір статистики про спожиті ресурси, для цього достатньо прослідкувати за логом монітору ВМ. Приблизно раз на 30 секунд має виконуватись опитування, відповідно, в логах має бути видно записи про це. Оскільки ми не виконуємо на ВМ ніяких інтенсивних операцій, будемо очікувати що за кожен період ВМ буде спожито менше 5 секунд процесорного часу.

```
[vmd] [      Thread-1] i.c.v.v.f.FireCrackerServiceThread      : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30033, cpuTime=0]
[vmd] [      Thread-1] org.hibernate.SQL                      : update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[vmd] [      Thread-1] i.c.v.v.f.FireCrackerServiceThread      : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30040, cpuTime=1]
[vmd] [      Thread-1] org.hibernate.SQL                      : update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[vmd] [      Thread-1] i.c.v.v.f.FireCrackerServiceThread      : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30040, cpuTime=0]
[vmd] [      Thread-1] org.hibernate.SQL                      : update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[vmd] [      Thread-1] i.c.v.v.f.FireCrackerServiceThread      : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30035, cpuTime=0]
[vmd] [      Thread-1] org.hibernate.SQL                      : update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[vmd] [      Thread-1] i.c.v.v.f.FireCrackerServiceThread      : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30043, cpuTime=0]
```

Рисунок 4.9 – Записи про зібрану статистику

На рисунку 4.9 видно, що ВМ дійсно не споживає більше 5 секунд процесорного часу. Додатково перевіримо коректність роботи шляхом створення штучного навантаження всередині ВМ утілітою sysbench (див. рисунки 4.10 та 4.11).

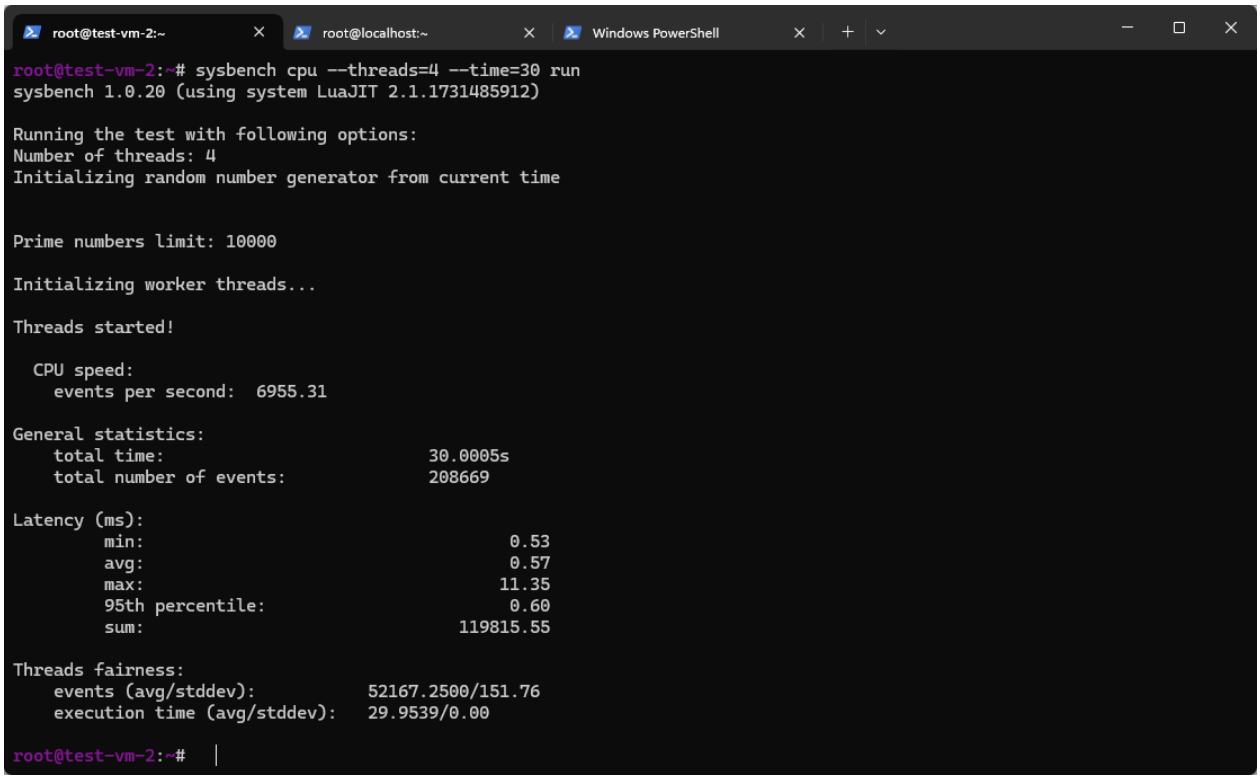


Рисунок 4.10 – Створення навантаження на процесор утілітою sysbench

```
update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30196, cpuTime=1]
update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30050, cpuTime=1]
update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=34778, cpuTime=118]
update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
[36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=30267, cpuTime=3]
```

Рисунок 4.11 – Запис зі статистикою

Оскільки ми навантажували 4 ядра протягом 30 секунд, монітор ВМ мав би зафіксувати споживання 120 секунд процесорного часу. За рисунком 4.11 видно, що було зафіксовано споживання 118 секунд. Розбіжність між теоретично розрахованим значенням і тим, яке було отримано на практиці, можна пояснити тим, що виконання sysbench не було закінчено на момент збору статистики, і припустити, що решта часу була врахована в наступному періоді, де як раз зафіксовано на 2 секунди більше часу ніж зазвичай.

Після цього можна перевірити автоматичне призупинення роботи ВМ. Для цього від'єднаємось від неї та почекаємо як мінімум 60 секунд.

```
i.c.v.v.f.FireCrackerServiceThread       : Suspending VM 36EE1661FFBC7064E065000000000001: timer ran out
PropertyNamingStrategy$SnakeCaseStrategy : PropertyNamingStrategy.SnakeCaseStrategy is used but it has been deprecated due to risk of deadlock. Consider
i.c.vmd.networking.NetworkingService     : TAP ctap0 assigned to interface primary of VM 36EE1661FFBC7064E065000000000001 got deleted via tearDownNetwor
```

Рисунок 4.12 – Запис про призупинення ВМ

За рисунком 4.12 бачимо, що ВМ була призупинена. Спробуємо створити ping-запит до неї та подивимось на записи в логах та час відповіді. Оскільки монітор ВМ мав зберігти стан ВМ перед призупиненням, відновлення роботи має відбутися швидко.

```
[root@ne41763 ~]# ping 192.168.122.240
PING 192.168.122.240 (192.168.122.240) 56(84) bytes of data.
64 bytes from 192.168.122.240: icmp_seq=1 ttl=64 time=172 ms
64 bytes from 192.168.122.240: icmp_seq=2 ttl=64 time=1.29 ms
64 bytes from 192.168.122.240: icmp_seq=3 ttl=64 time=1.93 ms
64 bytes from 192.168.122.240: icmp_seq=4 ttl=64 time=1.17 ms
64 bytes from 192.168.122.240: icmp_seq=5 ttl=64 time=0.902 ms
64 bytes from 192.168.122.240: icmp_seq=6 ttl=64 time=1.29 ms
```

Рисунок 4.13 – Результат виконання ping-запитів до призупиненої ВМ

На рисунку 4.13 видно, що у випадку коли ВМ відновлює роботу зі збереженого стану, відповідь на перший запит приходить значно швидше. Використання стану підтверджується записами з логів, що зображені на рисунку 4.14.

```
i.c.v.v.f.FireCrackerServiceThread       : Start request received for VM 36EE1661FFBC7064E065000000000001 (cold=false)
org.hibernate.SQL                        : select vm1_0.id,vm1_0.core_count,vm1_0.cpu_run_time,dp1_0.id,dp1_0.bridge,dp1_0.first_ip,dp1_0.last_ip,dp1_0
i.c.vmd.networking.NetworkingService     : Interface primary of VM 36EE1661FFBC7064E065000000000001 got this TAP: ctap0
PropertyNamingStrategy$SnakeCaseStrategy : PropertyNamingStrategy.SnakeCaseStrategy is used but it has been deprecated due to risk of deadlock. Conside
org.hibernate.SQL                        : select vm1_0.id,vm1_0.core_count,vm1_0.cpu_run_time,dp1_0.id,dp1_0.bridge,dp1_0.first_ip,dp1_0.last_ip,dp1_0
i.c.vmd.networking.queue.QueueWorker     : VM 36EE1661FFBC7064E065000000000001 up
org.hibernate.SQL                        : update virtual_machine vm1_0 set vm1_0.state=? where vm1_0.id=?
i.c.v.v.f.FireCrackerServiceThread       : [36EE1661FFBC7064E065000000000001] Process usage: Counters[clockTime=41953, cpuTime=6]
```

Рисунок 4.14 – Запис про «гарячий» запуск ВМ

Також варто перевірити доступ до консолі. Для цього було написано сценарій на мові JavaScript, що підключається до консолі ВМ через монітор засобами протоколу WebSockets. Частковий лістинг цього сценарію наведено на рисунку 4.15, а результати на 4.16.

```
function runTtyLoop(sock) {
    sock.onmessage = (event) => {
        let pkt = new Uint8Array(event.data);
        let op = pkt[1];
        switch (op) {
            case 0x0F:
                process.stdout.write(String.fromCharCode(pkt[2]));
                break;
            case 0x0B:
                let len = (pkt[2] << 8) | pkt[3];
                for (let i = 0; i < len; i++)
                    process.stdout.write(String.fromCharCode(pkt[3 + 1 + i]));
                break;
        }
    };

    sock.send(Uint8Array.from([0, 0x0A]));
    process.stdin.on('data', chunk => {
        if (chunk[0] == 0x03)
            process.exit(0);

        chunk.forEach(b => sock.send(Uint8Array.from([0, 0x0E, b])));
    })
}
```

Рисунок 4.15 – Частковий лістинг сценарію для підключення до консолі ВМ

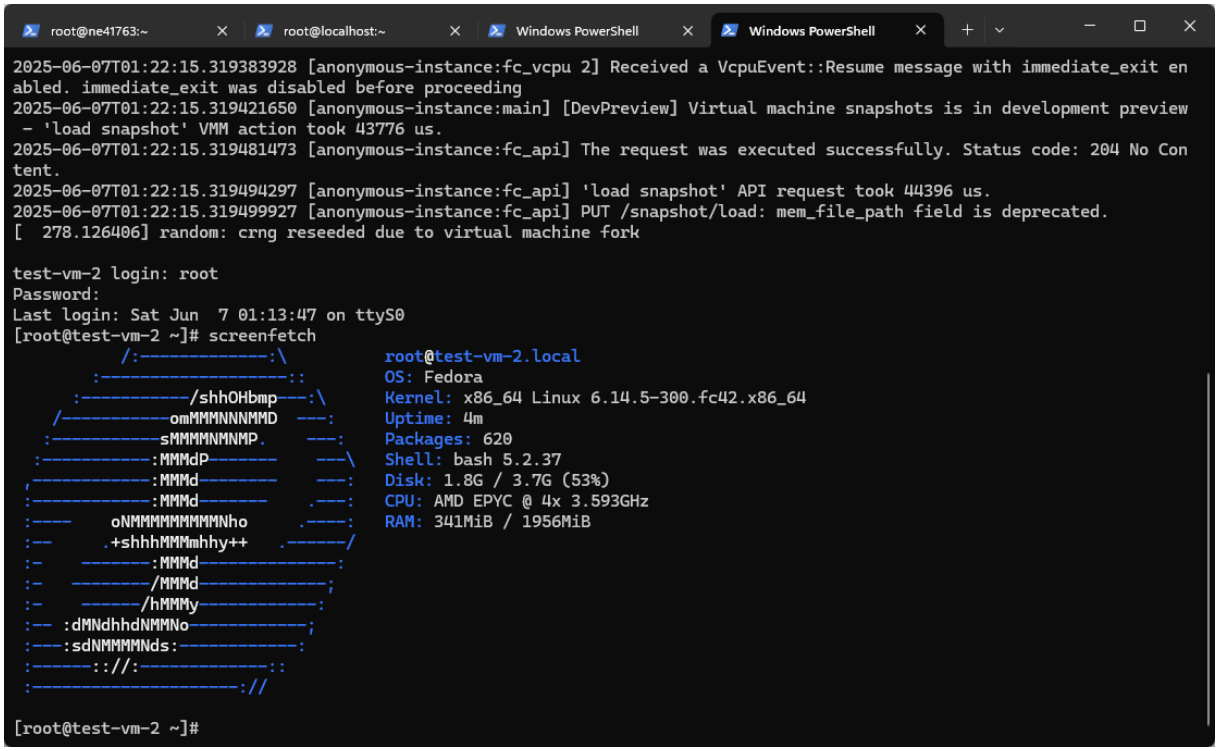


Рисунок 4.16 – Підключення до консолі ВМ через WebSockets

За результатами проведеного тестування можна зробити висновок, що монітор ВМ здатний коректно працювати з ВМ з виділеною IP-адресою, перехоплення пакетів та їх затримка до готовності ВМ виконується коректно. Також було перевірено збір статистики спожитих ресурсів, відновлення роботи зі збереженого стану та функціонал віртуальної консолі.

4.3. Тестування ВМ без виділеної IP-адреси

В розробленому програмному забезпечені реалізована можливість створення ВМ зі спільною IP-адресою, що працює через пересилку трафіку від мережевого інтерфейсу до мосту, а звідти до ВМ. Вручну налаштовувати міст не треба, монітор ВМ створює його автоматично для кожної визначеної локальної мережі. Отже, створимо цю мережу API-запитом, як це показано на рисунку 4.17, в якості IP-адрес будемо використовувати 10.10.10.0/24.

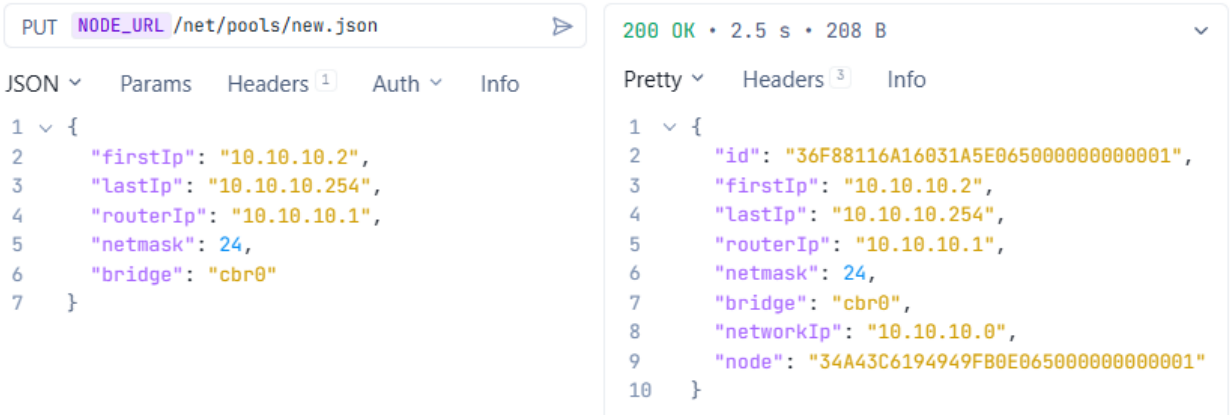


Рисунок 4.17 – Створення локальної мережі

Потім створимо ВМ в цій локальній мережі. API-запит для цього використовується такий самий, як і для ВМ з виділеною адресою, тільки замість ідентифікатору адреси потрібно передати ідентифікатор мережі, як це показано на рисунку 4.18.

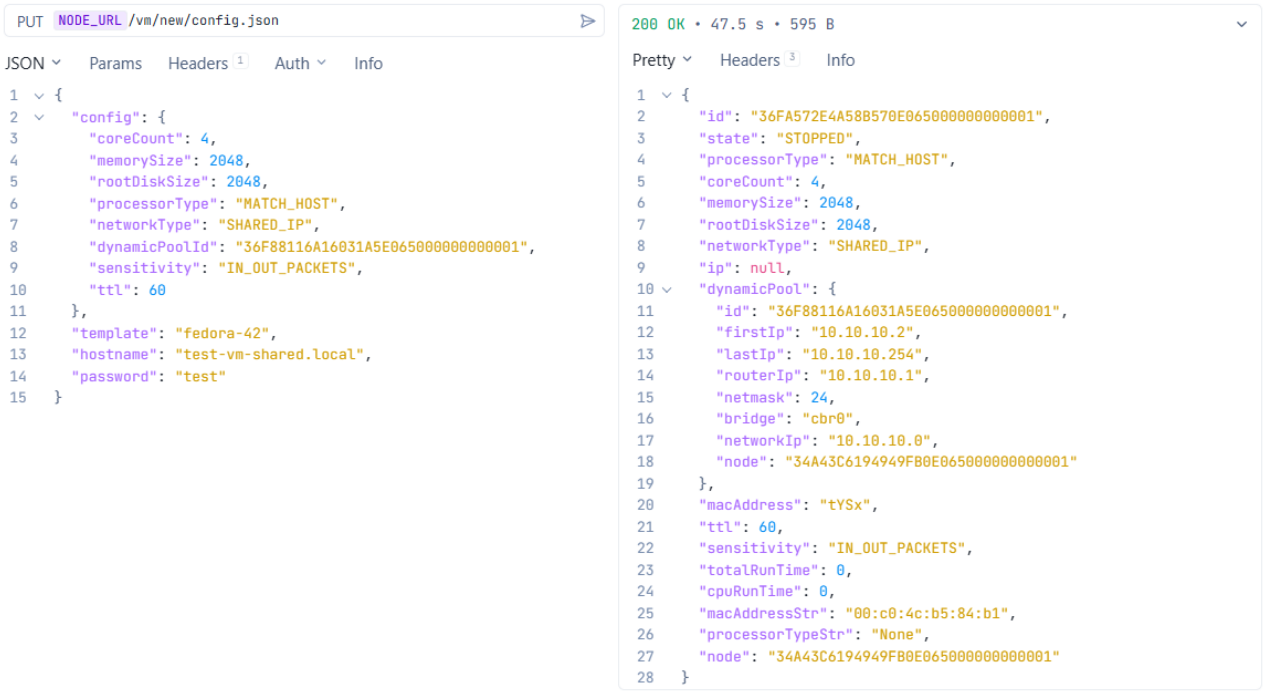


Рисунок 4.18 – Створення ВМ без виділеної адреси

Тепер визначимо спільну IP-адресу, порт з якої ми призначимо цій ВМ. Запит показано на рисунку 4.19.

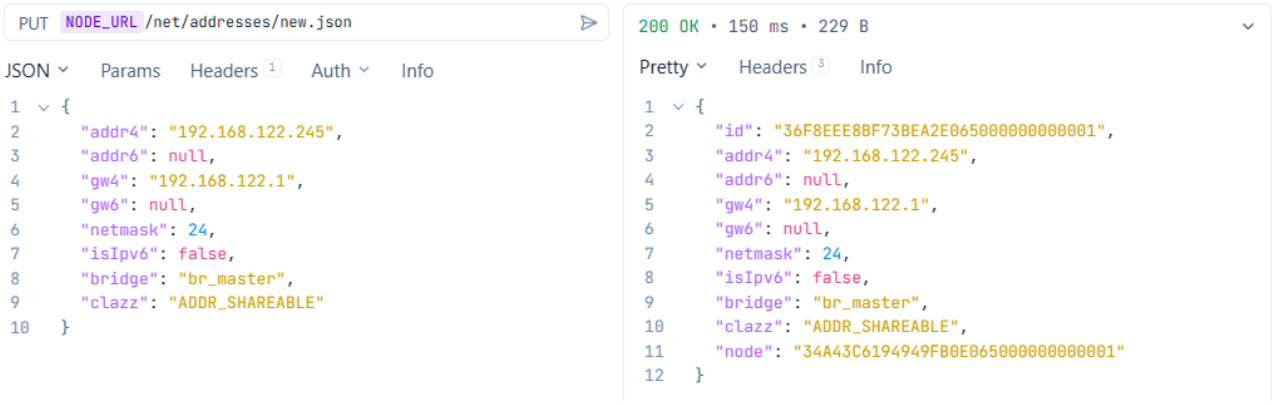


Рисунок 4.19 – Створення спільної IP-адреси

З метою тестування призначимо порт 2525 створеної IP-адреси ВМ. Переадресація трафіку буде відбуватись на порт 22, де працює SSH-сервер. Запит для призначення порту показано на рисунку 4.20.

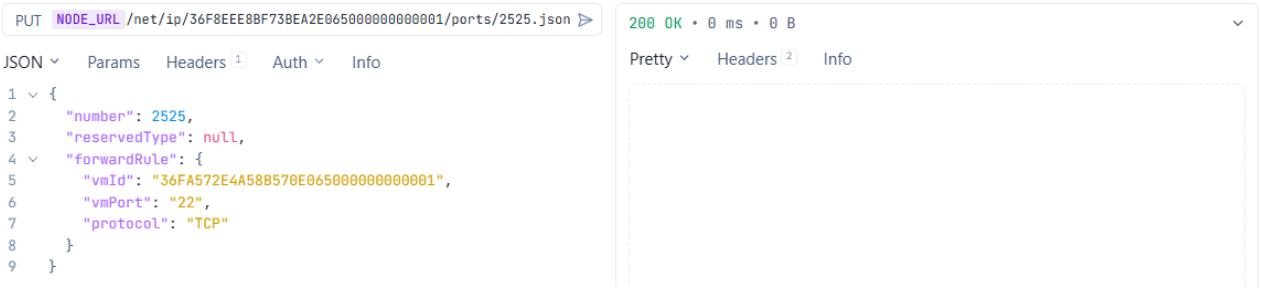


Рисунок 4.20 – Призначеня порту ВМ

Перевіримо, що пересилка трафіку та перехоплення пакетів працює належним чином, виконавши спробу під’єднання до порту (див. рисунок 4.21).

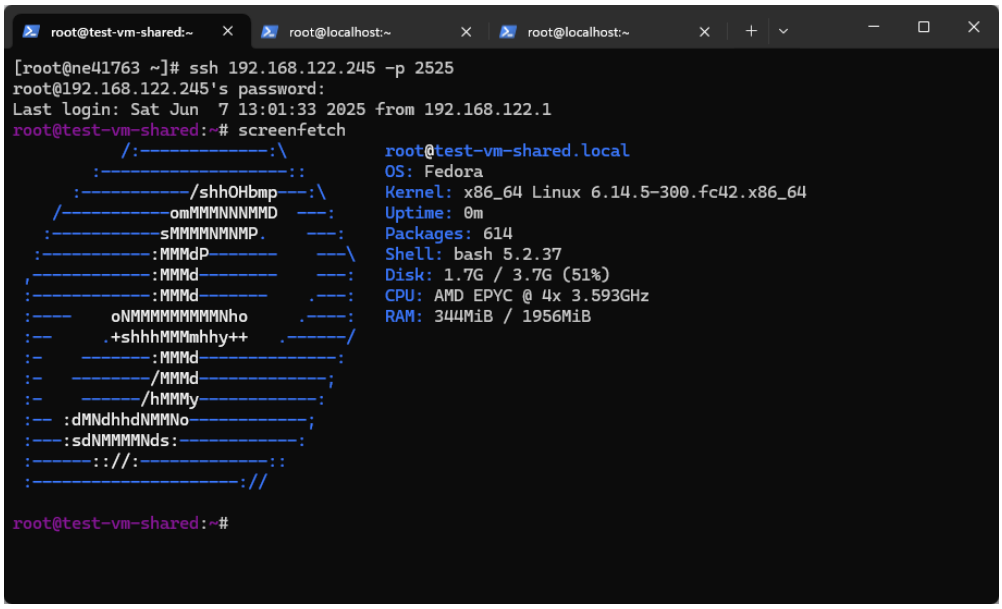


Рисунок 4.21 – Підключення до ВМ через окремий порт

В логах монітору видно записи про присвоєння IP-адреси (див. рисунок 4.22).

```
.vmd.networking.DhcpObserver  : DHCP DB updated, reloading state
.vmd.networking.DhcpObserver  : IP changed for VM 36FA572E4A58B570E065000000000001: new IP is 10.10.10.2
```

Рисунок 4.22 – Запис про присвоєння ВМ внутрішньої адреси

Отже, монітор ВМ здатний запускати ВМ, що не мають виділеної зовнішньої адреси, пересилання трафіку та перехоплення пакетів працює коректно.

Тепер перевіримо коректність маршрутизації трафіку прикладного рівня, закріпивши за ВМ символьну адресу test-vm-shared.local на порті 80 (HTTP). Спочатку створимо на спільній IP-адресі порт 80 та позначимо його як такий, що використовується для пересилання HTTP-трафіку, як це показано на рисунку 4.23.

Рисунок 4.23 – Визначення порта 80

Після цього створимо правило пересилки (див. рисунок 4.24).

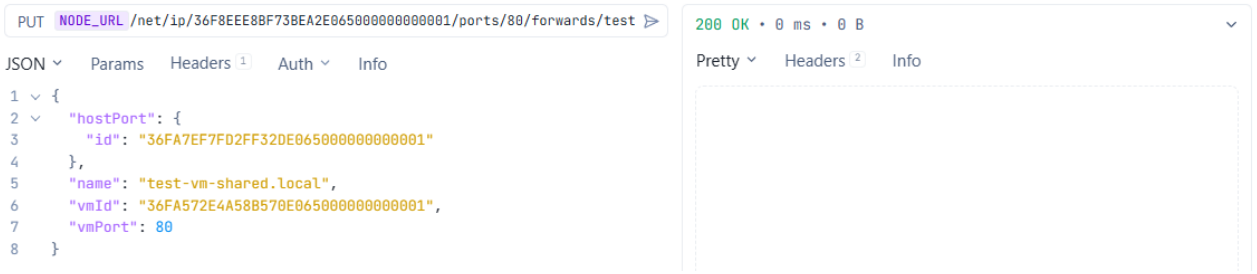


Рисунок 4.24 – Створення правила пресилки

Тепер виконаємо запит до цього порту, вказавши символьну адресу. Оскільки ВМ вимкнена, маємо отримати у відповідь перенаправлення на сторінку очікування.

```
[root@ne41763 ~]# curl --head -H "Host: test-vm-shared.local" http://192.168.122.245/
HTTP/1.1 302 Found
location: http://test-vm-shared.local/shorekeeper-cgi/WaitingRoom.pl
content-language: en-US
date: Sat, 07 Jun 2025 11:49:06 GMT
x-envoy-upstream-service-time: 24
server: envoy
transfer-encoding: chunked
```

Рисунок 4.25 – Отримане перенаправлення

Перейдемо за адресою, яка була отримана (див. рисунок 4.25). Після переходу на сторінку очікування має з’явитись запис в логах монітору ВМ про запуск віртуальної машини та перезавантаження конфігурації проксі-серверу.

```
[root@ne41763 ~]# curl --head -H "Host: test-vm-shared.local" http://192.168.122.245/shorekeeper-cgi/WaitingRoom.pl
HTTP/1.1 200 OK
content-type: text/html;charset=UTF-8
content-language: en-US
content-length: 346
date: Sat, 07 Jun 2025 12:16:09 GMT
x-envoy-upstream-service-time: 1381
server: envoy
```

Рисунок 4.26 – Перехід на сторінку очікування

```
: DHCP DB updated, reloading state
: IP changed for VM 36FA572E4A58B570E065000000000001: new IP is 10.10.10.2
: Envoy restarted
```

Рисунок 4.27 – Записи в логах про перезавантаження проксі-сереру

Як видно за рисунками 4.26 та 4.27, монітор ВМ запустив машину після переходу на сторінку очікування. Тепер спробуємо отримати статус ВМ (див. рисунок 4.28).

```
[root@ne41763 ~]# curl -H "Host: test-vm-shared.local" http://192.168.122.245/shorekeeper-cgi/VmAvailable.pl && printf "\n"
true
```

Рисунок 4.28 – Отримання статусу ВМ

Перевіримо, що пересилка трафіку працює. Перенаправлення на сторінку очікування відбуватись не має, адже ВМ активна. Як бачимо на рисунку 4.29, запит було передано проксі сервером до ВМ. Отже, налаштування правил пересилки працює, монітор ВМ здатний коректно керувати проксі-сервером, логіка роботи сторінки очікування відповідає вимогам.

```
[root@ne41763 ~]# curl --head -H "Host: test-vm-shared.local" http://192.168.122.245/
HTTP/1.1 200 OK
server: envoy
date: Sat, 07 Jun 2025 12:31:08 GMT
content-type: text/html
content-length: 8484
last-modified: Thu, 20 Mar 2025 00:00:00 GMT
etag: "67db5a80-2124"
accept-ranges: bytes
x-envoy-upstream-service-time: 1
```

Рисунок 4.29 – Виконання запиту до ВМ

4.4. Висновки до розділу 4

Четвертий розділ дипломного проєкту був присвячений тестуванню розробленого монітору ВМ. Було продемонстровано роботу програмного комплексу та взаємодію з монітором ВМ через його API. Тестування проводилось для ВМ з виділеною IP-адресою та зі спільною, було перевірено коректність роботи правил переадресації трафіку, віртуальної консолі, збору статистики, призупинення та відновлення роботи ВМ, а також логіку роботи сторінки очікування. Монітор ВМ продемонстрував здатність виконувати функції, що були передбачені технічним завданням.

ВИСНОВКИ

У результаті виконання дипломного проєкту було створено програмний комплекс моніторингу віртуальних машин, який дозволяє динамічно керувати обчислювальними ресурсами серверів з використанням гіпервізора Firecracker. Система автоматично призупиняє та відновлює роботу віртуальних машин залежно від наявності мережевого трафіку, що дозволяє зменшити навантаження на інфраструктуру без погіршення якості сервісу.

Під час проєктування було обґрунтовано вибір мови програмування Java для реалізації основного функціоналу, а також мови C — для створення високошвидкісних компонентів, що взаємодіють із ядром ОС. Для зберігання конфігурації використано реляційну базу даних Oracle Database, яка забезпечує високу швидкодію запитів з фільтрацією та підтримує цілісність даних завдяки наявності обмежень на рівні СКБД.

Було реалізовано багатокомпонентну архітектуру, що складається з кластеризованого монітору, проксі-серверу Envoy для обробки HTTP-запитів, а також допоміжних служб для аналізу мережевого трафіку, маршрутизації та динамічного керування IP-адресами. Завдяки використанню системи подій та принципів MVC, забезпечено слабке зв’язування між компонентами та можливість гнучкої модифікації системи.

Розроблена система може ефективно працювати в умовах змінного навантаження, має високу швидкодію відновлення ВМ із образу пам’яті, а також дозволяє централізовано керувати конфігураціями вузлів. Всі технічні рішення були підібрані з урахуванням вимог до безпеки, масштабованості та стабільності, що дозволяє розглядати систему як перспективну для використання в хмарній інфраструктурі загального призначення.

Таким чином, поставлені завдання дипломного проєкту були повністю виконані. Розроблена система демонструє високу ефективність у задачах

керування обчислювальними ресурсами та має потенціал для подальшого розвитку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Cloud computing — The business perspective // [Електронний ресурс]. — Режим доступу: https://doi.org/10.1016/j.dss.2010.12.— Дата доступу: червень 2025.
2. Best Web Hosting Control Panel Software in 2025 //. — Режим доступу: https://6sense.com/tech/web-hosting-control-panel. — Дата доступу: червень 2025.
3. DirectAdmin Web Control Panel Features // [[Електронний ресурс]]. — Режим доступу: https://directadmin.com/features_list.php. — Дата доступу: червень 2025.
4. User Cgroups // [Електронний ресурс]. — Режим доступу: https://docs.directadmin.com/operation-system-level/os-general/user_cgroups.html. — Дата доступу: червень 2025.
5. Filesystem and quotas // [Електронний ресурс]. — Режим доступу: https://docs.directadmin.com/operation-system-level/os-general/filesystems-and-quotas.html. — Дата доступу: червень 2025.
6. General [PHP] // [Електронний ресурс]. — Режим доступу: https://docs.directadmin.com/webservices/php/general.html. — Дата доступу: червень 2025.
7. Features (CloudLinux) // [Електронний ресурс]. — Режим доступу: https://docs.cloudlinux.com/ubuntu/features/. — Дата доступу: червень 2025.
8. How to install MySQL Governor // [Електронний ресурс]. — Режим доступу: https://support.cpanel.net/hc/en-us/articles/360053353033-How-to-install-MySQL-Governor. — Дата доступу: червень 2025.

9. Using the balloon device with Firecracker // [Електронний ресурс]. — Режим доступу: https://github.com/firecracker-microvm/firecracker/blob/main/docs/ballooning.md. — Дата доступу: червень 2025.
10. Preemptible VMs // [Електронний ресурс]. — Режим доступу: https://cloud.google.com/compute/docs/instances/preemptible. — Дата доступу: червень 2025.
11. Red Hat OpenShift Container Platform // [Електронний ресурс]. — Режим доступу: https://www.redhat.com/en/technologies/cloud-computing/openshift/container-platform. — Дата доступу: червень 2025.
12. Using quotas and limit ranges // [Електронний ресурс]. — Режим доступу: https://docs.redhat.com/en/documentation/openshift_container_platform/4.17/html/scalability_and_performance/compute-resource-quotas. — Дата доступу: червень 2025.
13. Resource quotas per project // [Електронний ресурс]. — Режим доступу: https://docs.redhat.com/en/documentation/openshift_container_platform/4.10/html/building_applications/quotas#quotas-setting-per-project. — Дата доступу: червень 2025.
14. What Is Knative? // [Електронний ресурс]. — Режим доступу: https://www.ibm.com/think/topics/knative. — Дата доступу: червень 2025.
15. CRI-O // [Електронний ресурс]. — Режим доступу: https://cri-o.io/. — Дата доступу: червень 2025.
16. Kata Containers // [Електронний ресурс]. — Режим доступу: https://katacontainers.io/. — Дата доступу: червень 2025.

17. Limitations (Kata Containers) // [Електронний ресурс]. — Режим доступу: https://github.com/kata-containers/kata-containers/blob/main/docs/Limitations.md. — Дата доступу: червень 2025.
18. Kerrisk, M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. — No Starch Press, — 1552 с.
19. An Empirical Study of OSS-Fuzz Bugs // [Електронний ресурс]. — Режим доступу: https://ar5iv.labs.arxiv.org/html/2103.11518v— Дата доступу: червень 2025.
20. Product Security Bad Practices (CISA) // [Електронний ресурс]. — Режим доступу: https://www.cisa.gov/resources-tools/resources/product-security-bad-practices. — Дата доступу: червень 2025.
21. Fannkuchredux Benchmark — The Computer Language Benchmarks Game // [Електронний ресурс]. — Режим доступу: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fannkuchredux.html. — Дата доступу: листопад 2022.
22. Don’t Believe The Lies: PHP Isn’t Thread-Safe Yet // [Електронний ресурс]. — Режим доступу: https://neosmart.net/blog/dont-believe-the-lies-php-isnt-thread-safe-yet/. — Дата доступу: червень 2025.
23. .NET documentation // [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/dotnet/. — Дата доступу: червень 2025.
24. GC policies — Eclipse OpenJ9 // [Електронний ресурс]. — Режим доступу: https://eclipse.dev/openj9/docs/gc. — Дата доступу: червень 2025.
25. Oracle Database // [Електронний ресурс]. — Режим доступу: https://www.oracle.com/ua/database. — Дата доступу: червень 2025.
26. etcd — Distributed reliable key-value store // [Електронний ресурс]. — Режим доступу: https://etcd.io/. — Дата доступу: червень 2025.

27. Comparison of MySQL, MSSQL, PostgreSQL, Oracle databases performance, including virtualization // [Електронний ресурс]. — Режим доступу: https://ph.pollub.pl/index.php/jcsi/article/view/2026/— Дата доступу: червень 2025.
28. Benchmarking 5 Popular Load Balancers: Nginx, HAProxy, Envoy, Traefik and ALB // [Електронний ресурс]. — Режим доступу: https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/. — Дата доступу: червень 2025.
29. Structuring Your Code — Spring Boot Reference // [Електронний ресурс]. — Режим доступу: https://docs.spring.io/spring-boot/reference/using/structuring-your-code.html. — Дата доступу: червень 2025.
30. Spring Events // [Електронний ресурс]. — Режим доступу: https://www.baeldung.com/spring-events. — Дата доступу: червень 2025.
31. IPv6 Statistics // [Електронний ресурс]. — Режим доступу: https://www.google.com/intl/en/ipv6/statistics.html. — Дата доступу: червень 2025.
32. Benvenuti C. Understanding Linux Network Internals. — O'Reilly Media, — 1035 с.
33. Envoy Proxy Configuration — Envoy docs // [Електронний ресурс]. — Режим доступу: https://www.envoyproxy.io/docs/envoy/latest/configuration/configuration. — Дата доступу: червень 2025.
34. Yaak // [Електронний ресурс]. — Режим доступу: https://yaak.app/. — Дата доступу: червень 2025.

ДОДАТОК А

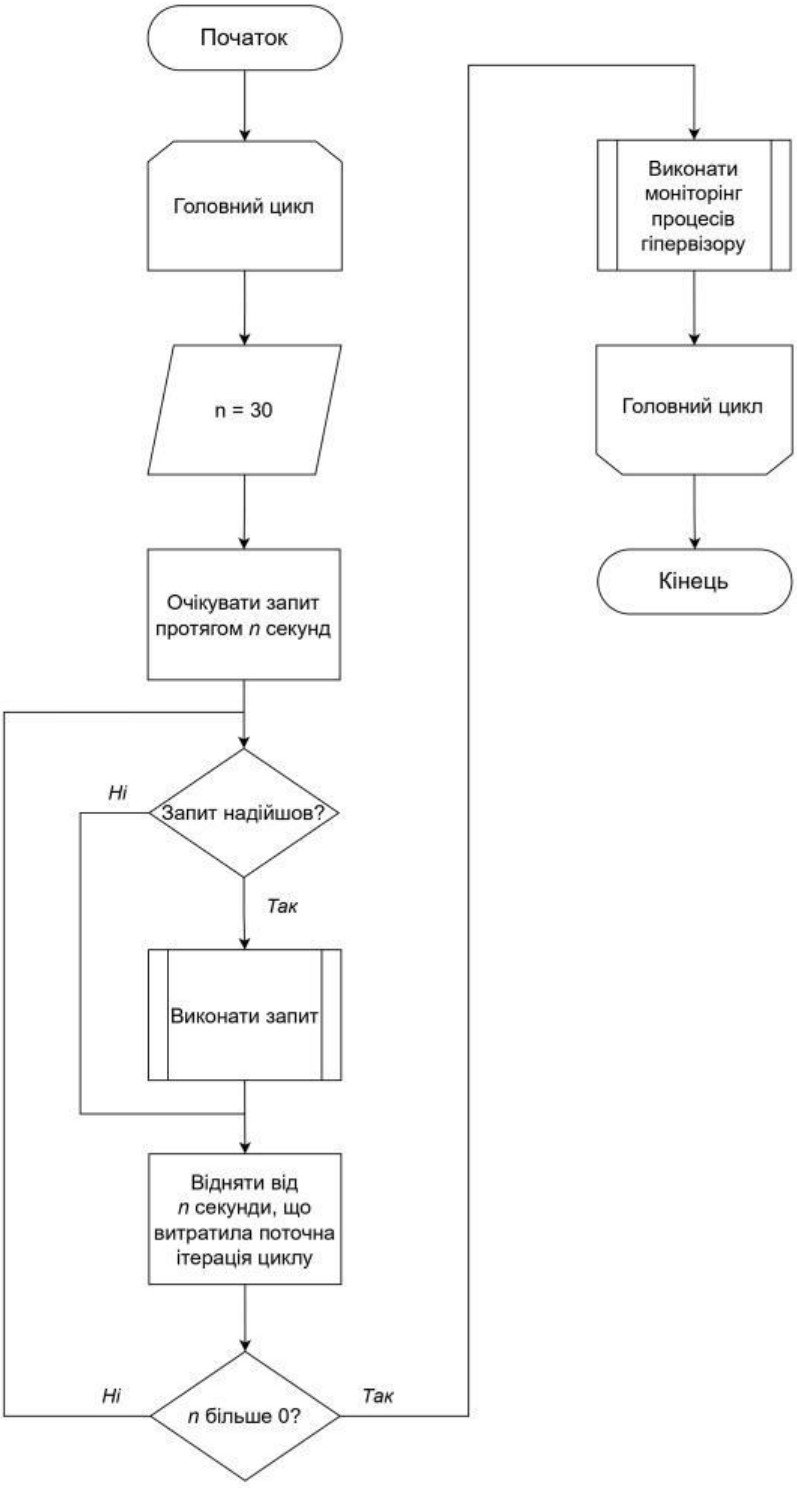
Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker

Алгоритм роботи менеджеру гіпервізора

ІАЛЦ.045490.005 Д1.

Аркушів 1

Київ 2025



					ІАЛЦ.045490.005 Д1	
Зм.	Арк.	№ докум.	Підпис	Дата	Алгоритм роботи менеджеру гіпервізора. Схема алгоритму.	Літ. / Маса / Масштаб
Розробив		Брюханов О.				
Перевірив		Петрашенко А.В.				Арк. / Аркушів
Н. контр.		Клятченко Я.М.				КПІ ім. Ігоря Сікорського, ФПМ, КВ-11
Затв.		Романкевич В.О.				

ДОДАТОК Б

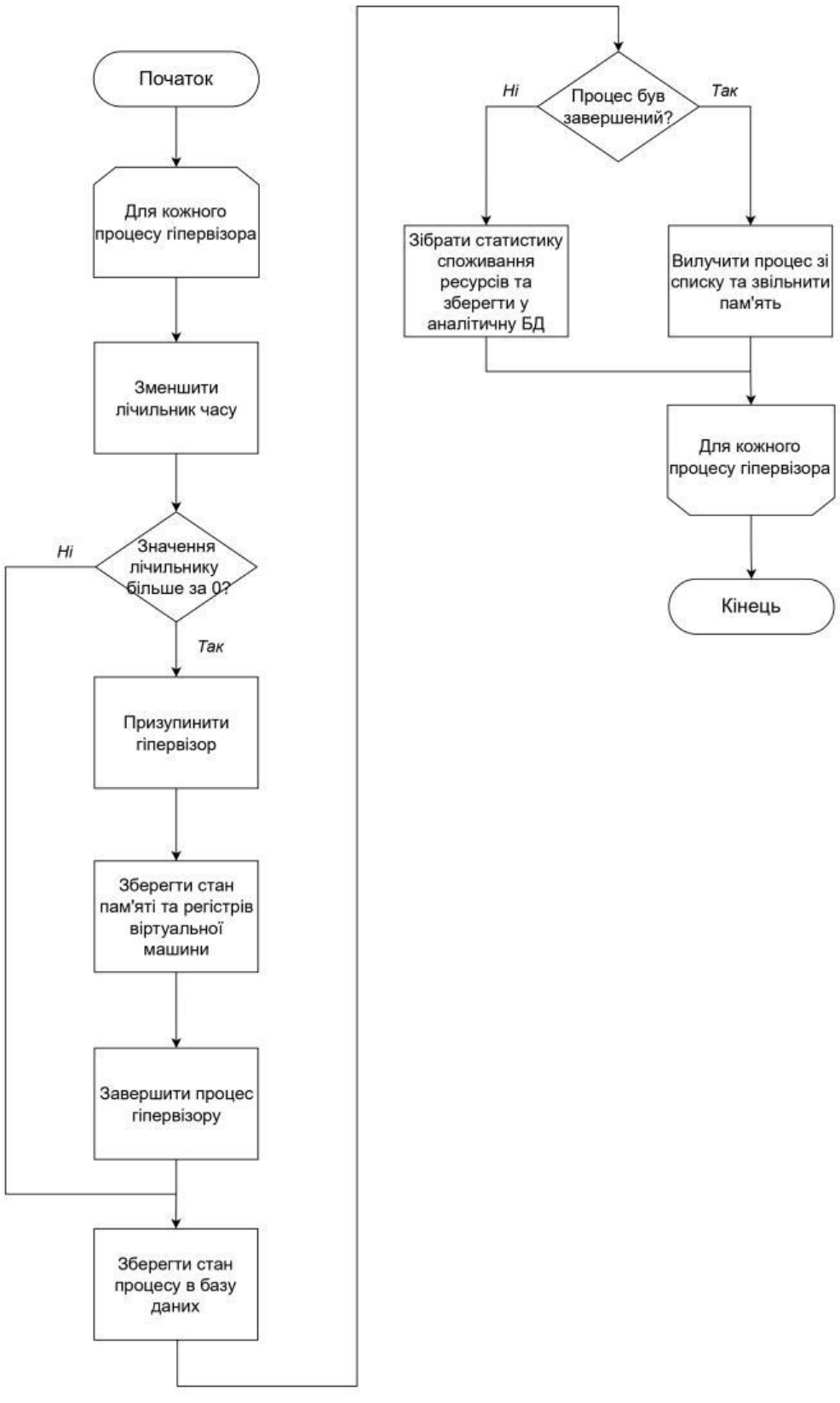
Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker

Алгоритм монітору процесів гіпервізора

ІАЛЦ.045490.006 Д2.

Аркушів 1

Київ 2025



Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.006 Д2			
Розробив		Брюханов О.			Алгоритм монітору процесів гіпервізора. Схема алгоритму.	Літ.	Маса	Масштаб
Перевірив		Петрашенко А.В.				Арк.	Аркушів	
Н. контр.		Клятченко Я.М.				КПІ ім. Ігоря Сікорського, ФПМ, КВ-11		
Затв.		Романкевич В.О.						

ДОДАТОК В

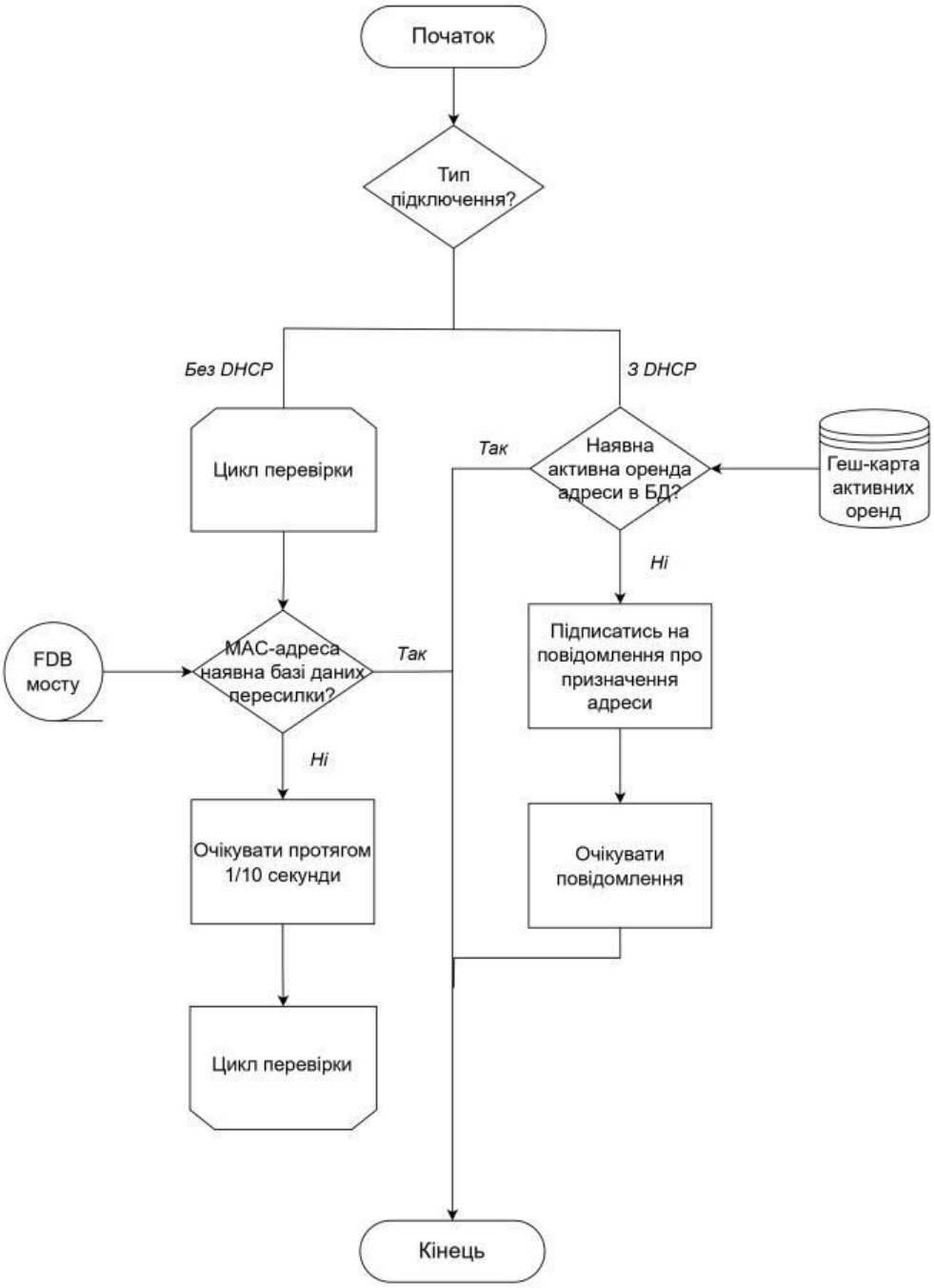
Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker

Алгоритм очікування готовності мережі віртуальної машини

ІАЛЦ.045490.007 ДЗ.

Аркушів 1

Київ 2025



					ІАЛЦ.045490.007 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата	Алгоритм очікування готовності мережі віртуальної машини. Схема алгоритму.	Літ. / Маса / Масштаб
Розробив		Брюханов О.				
Перевірив		Петрашенко А.В.				Арк. / Аркушів
Н. контр.		Клятченко Я.М.				КПІ ім. Ігоря Сікорського, ФПМ, КВ-11
Затв.		Романкевич В.О.				

ДОДАТОК Г

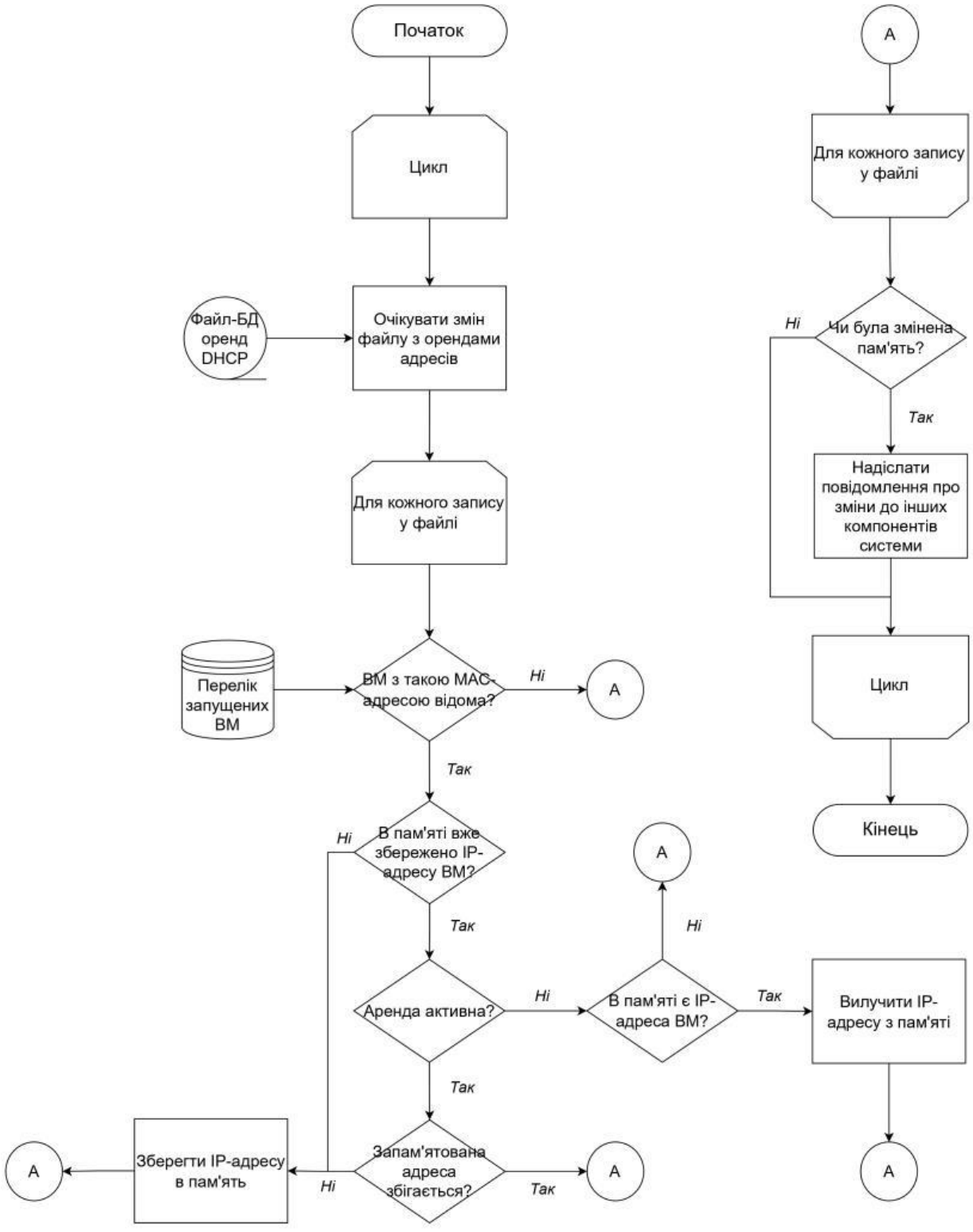
Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker

Алгоритм монітору оренди IP-адресів

ІАЛЦ.045490.008 Д4.

Аркушів 1

Київ 2025



					ІАЛЦ.045490.008 Д4	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Брюханов О.			Алгоритм монітору оренди IP-адресів. Схема алгоритму.	Літ. / Маса / Масштаб
Перевірив		Петрашенко А.В.				Арк. Аркушів
Н. контр.		Клятченко Я.М.				КПІ ім. Ігоря Сікорського, ФПМ, КВ-11
Затв.		Романкевич В.О.				

ДОДАТОК Д

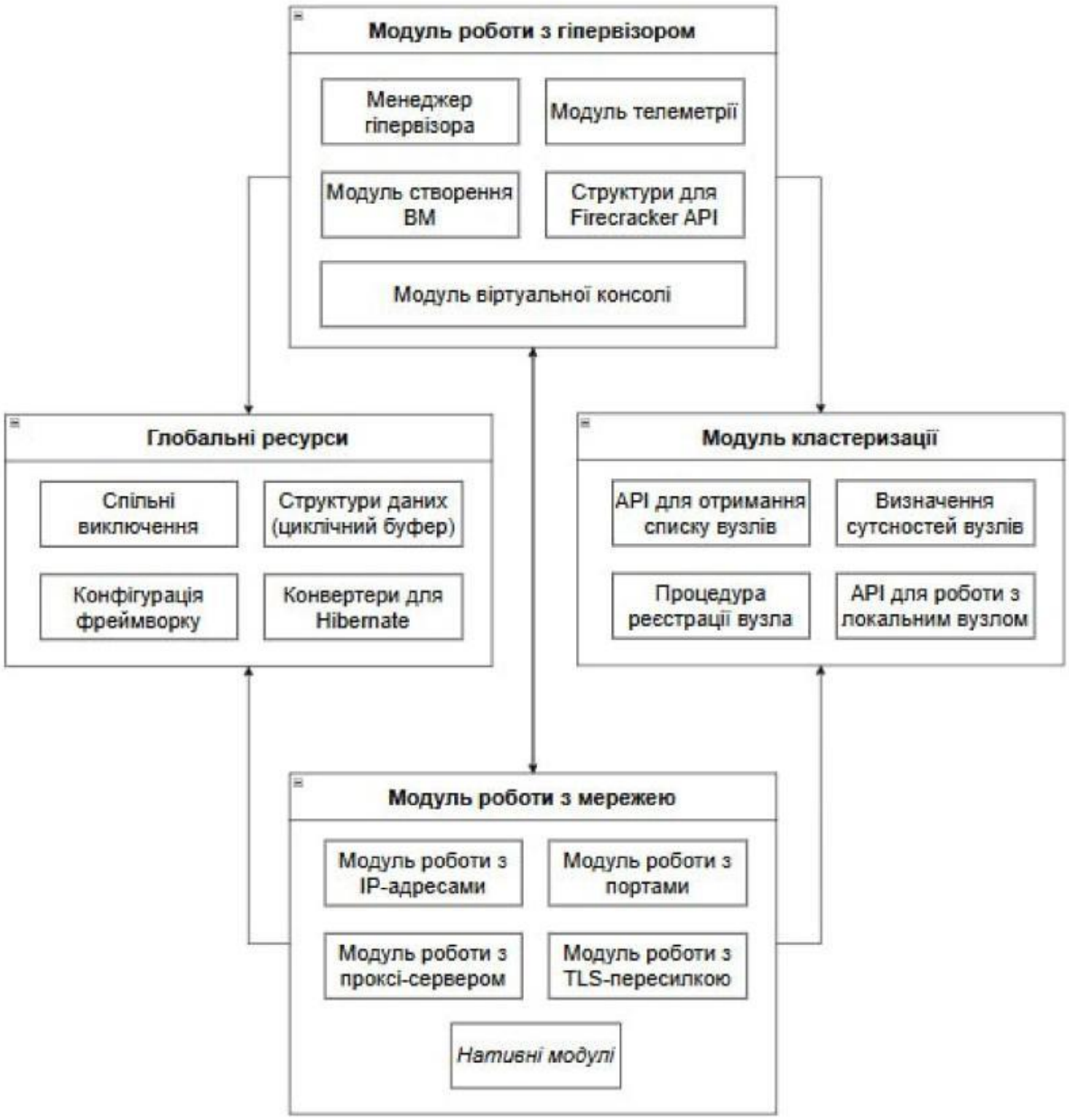
Програмний комплекс організації розподілу потужностей хмарних обчислень на основі гіпервізору Firecracker

Структура монітору віртуальних машин

ІАЛЦ.045490.009 Д5.

Аркушів 1

Київ 2025



Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045490.009 Д5			
Розробив		Брюханов О.			Структура монітору віртуальних машин. Схема структурна.	Літ.	Маса	Масштаб
Перевірив		Петрашенко А.В.						
						Арк.	Аркушів	
Н. контр.		Клятченко Я.М.				КПІ ім. Ігоря Сікорського, ФІПМ, КВ-11		
Затв.		Романкевич В.О.						