Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

”Допущено до захисту”
Завідувач кафедри кібербезпеки
та програмного забезпечення
д.т.н., професор
_____ Олексій СМІРНОВ
“____” _____ 2025 р.

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА
за першим (бакалаврським) рівнем вищої освіти
на тему
“Програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища”

Виконав здобувач вищої освіти
IV курсу, групи КІ-22-МБ
ОПП «Комп’ютерна інженерія»
спеціальності 123 «Комп’ютерна інженерія»
_____ Кицюк М.Д.
«____» _____ 2025 р.

Керівник проекту
кандидат технічних наук, доцент
_____ Коваленко А.С.
«____» _____ 2025 р.
Рецензент _____
_____

м. Кропивницький

Центральноукраїнський національний технічний університет

Факультет *Механіко-технологічний*

Кафедра *Кібербезпеки та програмного забезпечення*

Освітній ступінь *бакалавр*

Галузь знань ._12 _____*“*Інформаційні технології*”*_____

Спеціальність *123 “*Комп’ютерна інженерія*”*

Освітньо-професійна (освітньо-наукова) програма _*“*Комп’ютерна інженерія*”*_

ЗАТВЕРДЖУЮ
Завідувач кафедри
д.т.н., проф.
_____ Олексій СМІРНОВ
« 17 » січня 2025 року

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ ЗА ПЕРШИМ (БАКАЛАВРСЬКИМ) РІВНЕМ ВИЩОЇ ОСВІТИ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Кицюку Микиті Дмитровичу
(прізвище, ім’я, по батькові)

1. Тема роботи *Програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища*

2. Керівник роботи *Коваленко Анна Степанівна, канд. техн. наук, доцент*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу № 48-02 від 17.01.2025 року

3. Строк подання студентом роботи до захисту *23.05.2025 р.*

4. Мета та завдання випускної кваліфікаційної роботи*: Метою роботи є розробка програмного забезпечення системи віртуалізації з підтримкою розподіленого сховища*

5. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Призначення та область використання.

2.Перегляд аналогічних існуючих систем.

3. Опис і обґрунтування проектних рішень.

4. Етапи програмування системи.

5. Впровадження системи в промислову експлуатацію.

6. Висновки

6. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

Структурна схема системи	*1 аркуш*
Функціональна схема системи	*1 аркуш*
Діаграма процесів	*1 аркуш*
Блок-схема алгоритму роботи додатку	*2 аркуша*

7. Дата видачі завдання « 17 » січня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Строк виконання етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Примітка
1.	Аналіз існуючих систем	10.03.2025 р.	
2.	Постановка задачі, оформлення ТЗ	15.03.2025 р.	
3.	Розробка моделі компонента	20.03.2025 р.	
4.	Розробка структур даних	25.03.2025 р.	
5.	Розробка алгоритмів зв’язку та відображення	30.03.2025 р.	
6.	Програмування алгоритмів	10.04.2025 р.	
7.	Оформлення ПЗ	17.04.2025 р.	
8.	Попередній захист роботи	23.05.2025 р.	

Дата видачі завдання
« 17 » січня 2025 р.

Підпис керівника
_____ Коваленко А.С.
(прізвище та ініціали)

Завдання прийнято до виконання
« 17 » січня 2025 р.

Підпис здобувача
_____ Кицюк М.Д.
(прізвище та ініціали)

АНОТАЦІЯ

Кицюк М.Д. Програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища. 123 Комп'ютерна інженерія. Центральноукраїнський національний технічний університет. Кропивницький. 2025.

В даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти розроблено програмне забезпечення, яке призначено для системи віртуалізації з підтримкою розподіленого сховища.

Метою розробки є програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища.

Результат роботи – програмна реалізація системи віртуалізації з підтримкою розподіленого сховища.

В процесі роботи над програмною моделлю виконано аналіз існуючих апаратних та програмних засобів. В повній мірі описані всі компоненти розробленого програмного забезпечення.

Розроблено зручний інтерфейс користувача. Наведені інструкції по роботі з програмними засобами.

Програма може використовуватися на ПЕОМ з ОС Windows 10/11.

Програму розроблено в середовищі Python.

Ключові слова: комп'ютерна інженерія, віртуалізація

ABSTRACT

Kytsyuk M.D. Software for a virtualization system with distributed storage support. 123 Computer Engineering. Central Ukrainian National Technical University. Kropyvnytskyi. 2025.

In this final qualification work for the first (bachelor's) level of higher education, software has been developed, which is intended for a virtualization system with distributed storage support.

The purpose of the development is software for a virtualization system with distributed storage support.

The result of the work is a software implementation of a virtualization system with distributed storage support.

In the process of working on the software model, an analysis of existing hardware and software was performed. All components of the developed software are fully described.

A convenient user interface has been developed. Instructions for working with software are provided.

The program can be used on PCs with Windows 10/11.

The program is developed in Python.

Keywords: computer engineering, virtualization

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

ІПМ	–	імітаційна програмна модель
ЛОМ	–	локальні обчислювальні мережі
ПрЗд	–	пропускна здатність
ПМ	–	програмна модель
ТПП	–	таймер повторної передачі
ARTCP	–	удосконалений транспортний протокол
BER	–	імовірність бітової помилки
CR	–	запит на з'єднання
CBR	–	протокол передачі даних без підтверджень
FIFO	–	принцип first input fipst output
IP	–	адресний протокол
OSI	–	еталона модель мережної архітектури
RTT	–	час затрачуваємий підтвердженням
TCP	–	протокол транспортного рівня
TCP/IP	–	протокол передачі даних
TPDU	–	повідомлення транспортного протоколу
UDP	–	протокол користувальницьких датаграмм
WAN	–	територіальні мережі

ВСТУП

Актуальність теми. У даній роботі представлено рішення для керування даними, яке забезпечує швидке створення екземплярів віртуальної машини (VM) і ефективне виконання під час виконання для підтримки віртуальних машин як середовища виконання в Grid-обчисленнях. Робота заснована на нових методах віртуалізації розподіленої файлової системи та є унікальним у тому, що:

– рішення забезпечує міждоменний доступ на вимогу до стану віртуальної машини для немодифікованих моніторів віртуальної машини;

– рішення дає змогу використовувати приватні канали файлової системи для створення екземплярів віртуальної машини за допомогою безпечного тунелювання та автентифікації на основі сеансового ключа;

– рішення підтримує дискові кеші на рівні користувача та дискові кеш-пам’яті із зворотним записом, політики кешування для кожного додатка та моделі узгодженості, керовані проміжним програмним забезпеченням;

– рішення використовує метадані програми, пов’язані з файлами, для прискорення передачі даних.

У роботі повідомляється про його продуктивність у глобальних налаштуваннях з використанням віртуальних машин на базі VMware. Результати показують, що це рішення забезпечує більш ніж на 30% кращу продуктивність, ніж власна NFS, а завдяки теплим кеш-пам’ятам воно може зменшити накладні витрати, що сприймаються програмою, нижче 10% порівняно з налаштуванням локального диска. Рішення також дозволяє клонувати віртуальну машину з 1,6 ГБ віртуального диска та 320 МБ віртуальної пам’яті протягом 160 секунд для першого клонування та протягом 25 секунд для наступних клонів.

Мета й завдання дослідження. Метою роботи є програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища.

Для досягнення поставленої мети визначена програма дослідження, що складається з наступних завдань:

– Огляд існуючих систем віртуалізації з підтримкою розподіленого сховища.

– Дослідження системи віртуалізації з підтримкою розподіленого сховища.

– Програмна реалізація системи віртуалізації з підтримкою розподіленого сховища.

Практична цінність отриманих результатів полягає в тому, що розроблені алгоритми дозволяють успішно вирішувати задачі віртуалізації з підтримкою розподіленого сховища.

Таким чином, виходячи з вищеперерахованого, програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища, є актуальною задачею, яка потребує вирішення у даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1 Призначення системи

Віртуалізація є основоположною технологією, застосовною як для впровадження хмарних обчислень, так і для великих даних. Він забезпечує основу для багатьох атрибутів платформи, необхідних для доступу, зберігання, аналізу та керування розподіленими обчислювальними компонентами в середовищах великих даних. Віртуалізація – процес використання комп'ютерних ресурсів для імітації інших ресурсів – цінується за її здатність підвищувати використання IT-ресурсів, ефективність і масштабованість. Одним із основних застосувань віртуалізації є консолідація серверів, яка допомагає організаціям збільшити використання фізичних серверів і потенційно заощадити витрати на інфраструктуру. Однак віртуалізація має багато переваг. Компанії, які спочатку зосереджувалися виключно на віртуалізації серверів, тепер усвідомлюють, що її можна застосувати в усій IT-інфраструктурі, включаючи програмне забезпечення, сховища та мережі.

1.2 Область застосування

Віртуалізація відокремлює ресурси та служби від основного фізичного середовища доставки, що дозволяє створювати багато віртуальних систем в одній фізичній системі. На малюнку 5–1 показано типове середовище віртуалізації.

Однією з головних причин, чому компанії впроваджують віртуалізацію, є підвищення продуктивності та ефективності обробки різноманітних робочих навантажень. Замість того, щоб призначати виділений набір фізичних ресурсів для кожного набору завдань, об'єднаний набір віртуальних ресурсів можна швидко розподілити за потреби для всіх робочих навантажень. Покладення на

пул віртуальних ресурсів дозволяє компаніям зменшити затримку. Це підвищення швидкості та ефективності надання послуг є функцією розподіленого характеру віртуалізованих середовищ і допомагає покращити загальний час до окупності.

Використання розподіленого набору фізичних ресурсів, таких як сервери, більш гнучким і ефективним способом забезпечує значні переваги з точки зору економії коштів і підвищення продуктивності. Ця практика має кілька переваг, зокрема:

– Віртуалізація фізичних ресурсів (таких як сервери, сховища та мережі) дозволяє істотно покращити використання цих ресурсів.

– Віртуалізація дозволяє покращити контроль над використанням і продуктивністю ваших IT-ресурсів.

– Віртуалізація може забезпечити певний рівень автоматизації та стандартизації для оптимізації вашого обчислювального середовища.

– Віртуалізація забезпечує основу для хмарних обчислень.

Незважаючи на те, що можливість віртуалізації ресурсів додає величезну кількість ефективності, це не без витрат. Віртуальними ресурсами потрібно керувати так, щоб вони були безпечними. Зображення може стати технікою для зловмисника, щоб отримати прямий доступ до критичних систем. Крім того, якщо компанії не мають процесу видалення невикористаних зображень, системи більше не працюватимуть ефективно.

Таким чином, виходячи з вищеперерахованого, програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища, є актуальною задачею, яка потребує вирішення у даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти.

2 ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ

2.1 Огляд існуючих систем, технологій, архітектур, програмних рішень за профілем теми випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти

VMware vSphere для віртуалізації серверів

Продукт VMware vSphere – це набір продуктів для віртуалізації серверів. У нього входять гіпервізор ESX(i) і інструмент керування VMware vCenter server, на основі яких формується той або інший ліцензійний набір технологій. Ліцензії VMware vSphere можна порадити здобувати після покупки одного зі стартових пакетів Essentials або Acceleration kit.

VMware vSphere – продукт компанії VMware, що випускається в чотирьох основних видах ліцензій: Standard, Advanced, Enterprise, Enterprise Plus. Кожна ліцензія містить у собі свій набір технологій:

– **ESX** – автономний гіпервізор від компанії VMWare, що являє собою операційну систему (з консоллю керування), створену для запуску й керування віртуальними машинами. Входить тільки до складу платних ліцензій.

– **ESXi** – автономний гіпервізор від компанії VMWare, що являє собою операційну систему (без консолі керування), створену для запуску й керування віртуальними машинами. ESXi вирішує більше вузький клас завдань, чим ESX. Доступний для завантаження на сайті виробника в безкоштовному виданні.

– **vCenter Server** – сервер керування віртуальною інфраструктурою vSphere: серверами ESX(i), віртуальними мережами й сховищами. Установлюється в Windows сервер у вигляді декількох служб. Для підключення й керування vCenter server необхідний клієнт VMware vSphere Client.

– **VMware vSphere Client** – безкоштовний клієнт, що дозволяє управляти як окремим гіпервізором ESXi, так і vCenter server.

– **Thin provisioning** – технологія «тонких дисків», що дозволяє заощаджувати фізичний дисковий простір. Віртуальна машина займає рівно стільки місця на жорсткому диску, скільки в цей момент потрібно встановленої в неї операційній системі.

– **Update manager** – продукт для автоматичного відновлення встановлених гіпервізорів ESX(i), гостьових ОС, додатків у гостьових ОС, Virtual Appliance і компонентів ESX(i) (напр. віртуальних Комутаторів від Cisco).

– **VMSafe** – набір програмних інтерфейсів (API), що дозволяють стороннім партнерам VMware розробляти ефективні засоби забезпечення безпеки віртуального середовища.

– **vStorage API for Data Protection** – набір програмних інтерфейсів (API), що дозволяють стороннім партнерам VMware засобів резервного копіювання розробляти продукти для віртуальних інфраструктур.

– **High Availability (HA)** – компонент vSphere, що при виході з ладу якого-небудь хосту ESX, перезапускає віртуальні машини, що перебували на ньому, на іншому, працездатному хості. Тобто, навіть якщо vCenter вийде з ладу, HA буде продовжувати працювати.

– **Data recovery** – рішення від VMWare для резервного копіювання віртуальних машин.

– **Hot Add** – технологія, що дозволяє змінювати конфігурацію працюючих віртуальних машин без перезавантаження.

– **Fault Tolerance** – технологія безперервної доступності від VMWare, що досягається шляхом запуску дубліката віртуальної машини на ще одному хості ESX(i). При падінні хосту основної машини, дублікат миттєво стає оригіналом, у такий спосіб простою не відбувається.

– **vShield Zones** – зовнішній брендмауер для віртуальних машин на ESX(i).

– **vMotion** – “живаючи”, без простою роботи, міграція віртуальних машин з одного ESX(i) хосту на іншій.

– **Storage Vmotion** – “живаючи”,без простою роботи, міграція віртуальної машини з одного сховища (Storage) в інше.

– **DRS (Distributed Resource Scheduler)** – автоматичний розподіл віртуальних машин між серверами ESX(i) з метою балансування навантаження. Декілька хостів ESX(i) можна об'єднати в кластер, у ньому можна задавати пріоритети розподілу ресурсів між віртуальними машинами. Правильне налаштування DRS дозволить завжди забезпечувати бізнес-критичні віртуальні машини необхідними для роботи ресурсами.

– **vNetwork Distributed Switch** – рішення для ефективного централізованого адміністрування віртуальної мережі на великій кількості серверів ESX(i). Так само можливе використання комутаторів від Cisco (Nexus 1000V) прямо "усередині" ESX(i) для підключення до них віртуальних машин.

– **Host profiles** – профілі налаштувань – засіб для централізованого налаштування хостів ESX(i), і для автоматичного відстеження відповідності налаштувань профілю.

– **Third Party Multipathing** – можливість стороннім постачальникам розробляти модулі multipathing для ESX(i).

VMware vCenter server 5

У таблиці технологій VMware vSphere без vCenter server буде працювати тільки одна – Thin Provisioning, а щоб задіяти весь функціонал, обов'язкова установка цього платного сервера централізованого керування.

Модель керування безкоштовним ESXi

В одному вікні vSphere client адміністратор може управляти тільки одним хостом ESXi. Є можливість запустити кілька вікон vSphere client на одному комп'ютері й бачити на екрані декілька хостів відразу, але принцип керування не зміниться. Кожний хост буде існувати окремо від іншого, віртуальна машина буде «замкнена» у рамках свого ESXi.

Навіть якщо у вас є СЗД і диски віртуальних машин зберігаються на ній, у випадку аварії одного із серверів, необхідно буде вручну запускати кожну ВМ,

щоб вона продовжила працювати на іншому сервері. Це вкрай незручно й не схоже на відказостійкість навіть віддалено.

Ліцензію на безкоштовний ESXi уводити потрібно однаково, щоб відключити тріальний період в 60 днів. Вона видається на сайті www.vmware.com при завантажуванні дистрибутива.

Модель керування з vCenter server

vCenter server не може управляти сервером, на якому активована безкоштовна ліцензія ESXi. Необхідно буде, так чи інакше, купити процесорні ліцензії Standard, Enterprise або Enterprise Plus окремо або в складі пакетів Acceleration Kit, Essentials Kit.

На vCenter server теж потрібна одна із трьох видів ліцензій:

– vCenter server Essentials, входить до складу Essentials Kit.

– vCenter server Foundation, може управляти тільки 3-мя хостами ESXi, продається окремо, ціна 1943,50$ за ліцензію + 708,50$ за технічну підтримку.

– vCenter server Standard, може управляти необмеженим числом хостів ESXi, продається як окремо (6493$ за ліцензію + 1363,70$ за технічну підтримку) так і в складі пакетів Acceleration Kit.

Зверніть увагу, що на сайті зазначені ціни із прайс-аркуша VMware.

Установлюється vCenter server, як звичайне windows додаток, на Windows server 2003-2008. Після установки в операційній системі з'являються кілька нових служб, які відповідають за роботу vCenter. Для підключення до vCenter server використовується vSphere client, той же самий, що й для підключення до окремих ESXi хостам.

VMware vSphere Acceleration Kits

VMware vSphere Acceleration Kits – це набори ліцензій для компаній, які в перший раз купують ліцензії vSphere або збираються організувати ЦОД у своїй філії

Що таке Acceleration Kit? Це просто набір ліцензій із знижкою, у який включені окремі ліцензії VMware vSphere і ліцензія на центр керування vCenter

server. Технічна підтримка обов'язкова при покупці ліцензій і продається на весь пакет. Інакше кажучи, ви одержуєте необхідні ліцензії дешевше, ніж, якби купували їхньої по-окремості. Немає ніяких обмежень на кількість хостів, як у пакетах Essentials, ліцензія дозволяє використовувати, наприклад, 6 однопроцесорних хостів або 3 двопроцесорних.

Потрібно більше ліцензій, чим у складі Acceleration kit?

У ліцензію Acceleration Kit Standard входять 6 ліцензій vSphere Standard і одна ліцензія vCenter server Standard. Якщо вам потрібно більше чим 6 ліцензій, потрібно добирати відсутню кількість окремими ліцензіями Standard.

Обсяг vRAM з виходом версії 5.5 всі обмеження на vRAM були прибрані.

Кількість vCPU на одну ВМ. Кожній віртуальній машині ви зможете виділити або 8, або 64 процесорні ядра, залежно від ліцензії.

Ліцензія SUSE Linux Enterprise Server додається до ліцензій VMware. Річну підтримку для SUSE потрібно буде здобувати окремо.

Короткий опис технологій

Thin Provisioning. Дана технологія дозволяє заощаджувати дисковий простір системи зберігання даних (СЗД). Так, створивши віртуальну машину з тонким диском обсягом в 100 гігабайт, на системі зберігання даних буде зайняте всього-лише 10 гігабайт із можливістю подальшого динамічного розширення.

vShield Endpoint – надійний захист віртуальних машин від вірусів і інших шкідливих програм.

vSphere Replication – дозволяє робити реплікацію гіпервізора по сховищах на рівні дисків окремих віртуальних машин.

Update Manager – засіб для відновлення хостів vSphere з vCenter server.

Data Recovery – дане програмне забезпечення дозволяє створювати резервні копії й відновлювати VMware vSphere.

High Availability – функція високої доступності, яка дозволяє поєднувати відразу декілька хостів в один кластер, при цьому спеціальний агент стежить за доступністю хостів кластера. У тому випадку, якщо один з хостів не відповідає,

відбувається автоматичний перезапуск віртуальної машини на робітників хостах, що дозволяє істотно скорочувати час простою.

vMotion – дозволяє робити міграцію віртуальної машини з одного хосту на інший без простою – це досягається завдяки тому, що файли при міграції залишаються на місці, а міняється тільки хост-сервер.

vStorage APIs for Data Protection – набір готових інтерфейсів, що дозволяє розроблювачам програмного забезпечення (Veeam, NetApp, EMC і т.д.) створювати ПЗ для резервного копіювання віртуальних машин.

Virtual Serial Port Concentrator – дозволяє прокидати й перенаправляти COM порт по мережі під час міграції віртуальних машин.

Hot Add – тепер додавати нові пристрої у віртуальну машину можливо без простоїв у роботі – навіть оперативну пам'ять або процесор.

vShield Zones – брандмауер для віртуальної інфраструктури, що дозволяє контролювати трафік від хостів і до хостам.

Fault Tolerance (FT) – функція безперервної доступності віртуальної машини. Якщо вона включена, на іншому хості кластера запускається дублікат віртуальної машини, і у випадку відмови хосту, автоматично запускається дублікат віртуальної машини без простоїв у роботі.

vStorage APIs for Array Integration – набір готових інтерфейсів для розроблювачів ПЗ, що дозволяє запускати процеси (vMotion, Snapshot, Thin provisioning і ін.) з використанням ресурсів винятково системи зберігання даних.

Storage APIs for Multipathing – готові інтерфейси для розроблювачів програмного забезпечення, що дозволяють набудовувати відказостійкі підключення до системи зберігання даних по декількох дублюючих каналах.

Storage vMotion – міграція файлів віртуальних машин з однієї системи зберігання даних на іншу (з LUN на LUN, зі СЗД на локальні диски сервера й т.п.) без простою.

Distributed Resource Scheduler (DRS) – вирівнювання навантаження хостів: система DRS автоматично переміщає кілька віртуальних машин з більше завантаженого хосту на інший.

Distributed Power Management (DPM) – режим економії енергії, при якому знижується загальне навантаження хостів кластера шляхом міграції віртуальних машин з одного хосту з подальшим його відключенням (наприклад, уночі).

Storage I/O control – розподіл навантаження системи зберігання даних: для якої задається ліміт на максимальне значення latency, при його досягненні vCenter сервер починає розподіляти навантаження на систему зберігання даних відповідно до виставлених пріоритетів між віртуальними машинами на хості.

Network I/O control – функція, що дозволяє розділяти трафік по типах і контролювати показник IN/OUT для кожного з них. Адміністратор задає для них правила, які починають працювати в моменти високого навантаження.

Distributed switch – віртуальний розподілений світч, що дозволяє в єдиному інтерфейсі робити мережні налаштування відразу для всіх хостів, що дуже корисно, якщо їх багато.

Host Profiles – шаблони налаштувань хосту.

Auto deploy – поліпшена модель розгортання й відновлення хостів ESXi, більше швидка й зручна. vCenter тепер не тільки може зберігати профілі хостів, але й образи ESXi для розгортання.

Storage DRS – функція підвищення ефективності роботи системи зберігання даних шляхом автоматичного вирівнювання навантаження шляхом розподілу дисків віртуальної машини між LUN-ами.

Profile-Driven Storage – можливість створювати рівні сервісу й призначати їхнім віртуальним машинам, щоб, наприклад, на дорогому RAID розділі не виявилося файлове сховище.

Microsoft Hyper-V Server

Microsoft Hyper-V Server надає надійне й оптимізоване рішення віртуалізації, що дозволяє організаціям підвищити ефективність використання серверів і знизити витрати. Завдяки тому, що Microsoft Hyper-V Server є окремим продуктом, що містить винятково Hyper-V і компоненти вируализації, вартість

його нижче. Він легко встановлюється у вже існуючу IT середовище організації. Нові можливості, такі як динамічний перенос і розширена підтримка пам'яті й процесора в обслуговуючих системах, дозволяють об'єднати робочі навантаження в рамках одного фізичного сервера й надають гарне рішення по консолідації серверів, а також по розробці й тестуванню.

Як відбувається віртуалізація на платформі Microsoft?

Перший спосіб. На підтримуючий віртуалізацію фізичний сервер установлюється ліцензійна операційна система Windows Server 2008 R2 у редакції Enterprise або Datacenter. Далі через меню управління сервера додається роль Hyper-V. Після чого на цьому сервері можна створювати віртуальні машини. Керування Hyper-V відбувається через консоль mmc самого сервера або через віддалене підключення й відповідне оснащення. Даний спосіб дозволяє використовувати набір функцій, що у таблиці нижче називається – Hyper-**V Console** і являє собою окремий сервер (standalone) .

Другий спосіб. Дозволяє об'єднати кілька окремих серверів і управляти ними спільно через Microsoft System Center Essentials 2010. Ліцензується дана модель за схемою ліцензія на сервер+ліцензія на керування пристроєм.

Третій спосіб (оптимальний). Дозволяє об'єднати кілька окремих серверів і управляти ними спільно через Microsoft Virtual Machine Manager 2008 R2 (у таблиці VMM 2008 R2). Придбати дану ліцензію можна трьома способами:

– окрему ліцензію на VMM 2008 R2 Enterprise (869$) – для керування хостами (фізичний сервер+віртуальні машини на ньому) у необмеженій кількості;

– окрему ліцензію на VMM 2008 R2 Workgroup (505$) – для керування 5-ю хостами;

– у складі в складі пакета Microsoft System Center Server Management Suite Enterprise.

Templates – можливість створення шаблонів віртуальних машин.

Candidate Identification – можливість оцінки операційних систем, як кандидатів для перекладу у віртуальне середовище.

Physical to Virtual Conversion – технологія конвертації фізичних серверів у віртуальні.

Virtual to Virtual Conversion (VMware) – технологія конвертації віртуальних машин VMware у віртуальні машини Hyper-V.

Migration Across physical machines – технологія міграції усередині Hyper-V.

Virtualization report – звіт про міграцію.

Monitoring VMs – моніторинг віртуальних машин.

Physical resource optimization (PRO) – інструмент, що застосовується для оптимізації й усунення неполадок в Hyper-V server. Він також видає підказки й пропозиції по оптимізації додатків усередині віртуальних машин.

Library – опція, що створює сервер на якому зберігаються бібліотеки дистрибутивів для установки операційних систем.

Provisioning – швидка установка нових віртуальних машин з «золотого» образа.

VM configuration & properties – можливість набудовувати конфігурацію віртуальних машин.

VM State – можливість створювати звіт про стан віртуальних машин.

Checkpoints (Snapshots) – можливість створювати миттєві знімки стану віртуальних машин.

64 bit guest OS – убудована підтримка 64 бітних операційних систем усередині віртуальних машин.

Hardware Assisted Virtualization – продукт, що допомагає визначити чи підтримує процесор апаратну віртуалізацію, якщо так, те допомагає включити неї.

Live thumbnail – як в Windows 7 вікна з мініатюрними зображеннями вікон віртуальних машин.

Synthetic Network Support – підтримка й оптимізація роботи мережі в Linux подібних операційних системах усередині віртуальної машини Hyper-V.

Import VM (multiple VHD + snapshots) – можливість імпортувати віртуальні машини Hyper-V з декількома віртуальними жорсткими дисками й миттєвими знімками.

Virtual Network Configuration – можливість набудовувати віртуальну мережу.

Inspect Disk – можливість управляти .VHD жорсткими дисками.

VM cloning – функція клонування віртуальних машин.

VMware Management – можливість установки гіпервізора ESX(i) у віртуальну машину Hyper-V.

Self-service console – власна консоль для керування.

System Center, ліцензування, ціни, опис

З випуском System Center і Windows server компанія Microsoft представила своє бачення організації приватної або публічної хмари. Якщо розглядати властивості хмари, то одне з основних його рис – це керування через єдину консоль, що дозволяє управляти інформаційною системою (хмарою) не замислюючись, як вона організована на фізичному рівні. По натисканню кнопки в System Center повинні відбуватися події, які раніше могли зажадати від адміністратора фізичного втручання з походом у серверну кімнату.

У ліцензуванні своїх продуктів Microsoft тепер повністю розділяє клієнтів на два типи, які використовують віртуалізацію і які її не використовують.По тім же принципі ліцензується System Center, якщо уважно подивитися таблицю нижче, те видно, що функціональна різниця між ліцензіями полягає в тім, що Datacenter призначено для керування системами, де на кожному фізичному сервері запущено багато віртуальних машин.

Власна хмара вимагає інших засобів керування, не тих які використовувалися раніше, спеціально для цього розроблений програмний пакет System Center. Розглянемо докладніше його состав.

System Center Configuration Manager – чим більш разнородна інфраструктура компанії, тим складніше нею управляти. Для доступу до

корпоративних додатків співробітники можуть використовувати стаціонарні комп'ютери, ноутбуки, тонкі клієнти, мобільні телефони, планшети, домашні комп'ютери робочі станції в інтернет кафе й багато чого іншого. Завдання IT відділу, забезпечити контрольований доступ до бізнес сервісам, можливість тимчасової установки додатків за запитом або самостійно (теж під контролем) (якщо врахувати, що співробітник виїхав у відрядження). Всі додатки повинні бути сумісні з тими пристроями, з яких користувач намагається працювати. З усіма цими завданнями покликаний справлятися Configuration Manager.

– Збільшення ефективності роботи користувачів.

– Уніфікована інфраструктура керування й безпеки.

– Більш просте адміністрування.

System Center Operations Manager – цей компонент призначений для моніторингу додатків і серверів у різнорідних середовищах за допомогою локально встановлених агентів. У підсумку надає комплексну інформацію про роботу центра обробки даних.

– Дозволяє оперативно одержувати інформацію про стан вашої системи.

– Убудовані інструменти прогнозування розвитку інфраструктури.

System Center Data Protection Manager – засіб резервного копіювання й відновлення від Microsoft. Дозволяє якісно захистити ваші сервери, віртуальні машини, окремі додатки, а також гранулярне відновлення окремих елементів додатків, наприклад, по помилці віддалених таблиць у базі даних. Даний продукт, як і інші, інтегрується в загальну консоль керування System Center Manager 2012 і дозволяє управляти схоронністю ваших даних централізовано.

– Сумісність і інтеграція в існуючу систему.

– Підтримка резервного копіювання на диски, на стрічку, у хмару.

System Center Service Manager – функціонал даного компонента придасться сервіс провайдерам, які надають свої хмарні ресурси клієнтам і хочуть забезпечити їх кваліфікованою технічною підтримкою. У продукт

убудовані готові процеси по усуненню інцедентів і проблем, керування змінами й життєвими циклами активів:

– готова технічна підтримка відповідно до ITIL.

System Center Virtual Machine Manager – головний інструмент для керування віртуальною інфраструктурою й автоматизації процесів спрямованих на відказостійкість і розподіл навантаження. Центральна консоль, у якій можна задати параметри для віртуальних машин, мереж передачі даних. У ній можна сформувати пули ресурсів і настроїти відказостійкі кластери із серверів:

– закінчений інструмент для створення приватної хмари.

– керування різнорідними середовищами, у тому числі Citrix і VMware .

System Center Endpoint Protection – компонент, що вбудовується в Configuration Manager і дозволяє забезпечувати комплексний захист серверів і робочих станцій від погроз безпеки (по суті отут антивірус, firewall, контроль доступу). Також дана система дозволяє контролювати дотримання корпоративної політики безпеки в організації:

– Можливість відмовитися від сторонніх антивірусів.

System Center orchestrator – інструмент для автоматизації процесів у вашій інфраструктурі. Дозволяє через свої інтерфейси взаємодіяти із системами керування від інших вендорів, таких як HP, IBM, EMC, BMC, CA і VMware через готові пакети інтеграції:

– Зниження витрат і поліпшення передбачуваності за допомогою автоматизації.

2.2 Обґрунтування вибору засобів для побудови системи та мови програмування

Як мова програмування обрана Python. Python – високорівнева мова програмування загального призначення з акцентом на продуктивність розроблювача й читаність коду. Синтаксис ядра Python мінімалістичний. У той

же час стандартна бібліотека включає великий обсяг корисних функцій. Python підтримує кілька парадигм програмування, у тому числі структурне, об'єктно-орієнтоване, функціональне, імперативне й аспектно-орієнтоване. Основні архітектурні риси – динамічна типізація, автоматичне керування пам'яттю, повна інтроспекція, механізм обробки виключень, підтримка багатопоточні обчислень і зручні високорівневі структури даних. Код у Python організовується у функції й класи, які можуть поєднуватися в модулі (які у свою чергу можуть бути об'єднані в пакети).

Еталонною реалізацією Python є інтерпретатор CPython, що підтримує більшість активно використовуваних платформ. Він поширюється вільно під дуже ліберальною ліцензією, що дозволяє використовувати його без обмежень у будь-яких застосунках, включаючи пропрієтарні. Є реалізації інтерпретаторів для JVM (з можливістю компіляції), MSIL (з можливістю компіляції), LLVM і інших. Проект PyPy пропонує реалізацію Python на самому Python, що зменшує витрати на зміни мови й постановку експериментів над новими можливостями.

Python – мова програмування, що активно розвивається, нові версії (з додаванням/зміною язикових властивостей) виходять приблизно раз у два з половиною року. Внаслідок цього й деяких інших причин на Python відсутні ANSI, ISO або інші офіційні стандарти, їхня роль виконує CPython.

Python портований і працює майже на всіх відомих платформах – від КПК до мейнфреймів. Існують порти під Microsoft Windows, практично всі варіанти UNIX (включаючи FreeBSD і Linux), Plan 9, Mac OS і Mac OS X, iPhone OS 2.0 і вище, Palm OS, OS/2, Amiga, AS/400 і навіть OS/390, Symbian і Android.

При цьому, на відміну від багатьох портуємих систем, для всіх основних платформ Python має підтримку характерних для даної платформи технологій (наприклад, Microsoft COM/DCOM). Більше того, існує спеціальна версія Python для віртуальної машини Java – Jython, що дозволяє інтерпретаторові виконуватися на будь-якій системі, що підтримує Java, при цьому класи Java можуть безпосередньо використовуватися з Python й навіть бути написаними на

Python. Також кілька проектів забезпечують інтеграцію із платформою Microsoft .NET, основні з яких – IronPython і Python.Net.

Python підтримує динамічну типізацію, тобто тип змінної визначається тільки під час виконання. Тому замість «присвоювання значення змінної» краще говорити про «зв'язування значення з деяким ім'ям». У Python є убудовані типи: бульові, рядки, Unicode-рядки, цілі числа довільної точності, числа із плаваючої коми, комплексні числа й деякі інші. З колекцій Python підтримує кортежі (*tuples*), списки, словники (асоціативні масиви) і, починаючи з версії 2.4, безлічі. Всі значення в Python є об'єктами, у тому числі функції, методи, модулі, класи.

Додати новий тип можна або написавши клас (class), або визначивши новий тип у модулі розширення (наприклад, написаному мовою C). Система класів підтримує спадкування (одиночне й множинне) і метапрограмування. Можливе спадкування від більшості убудованих типів і типів розширень.

Всі об'єкти діляться на посилальні й атомарні. До атомарного ставляться int, long, complex і деякі інші. При присвоюванні атомарних об'єктів копіюється їхнє значення, у той час як для посилальних копіюється тільки покажчик на об'єкт, таким чином, обидві змінні після присвоювання використовують те саме значення. Посилальні об'єкти бувають змінювані й незмінні. Наприклад, рядки й кортежі є незмінними, а списки, словники й багато інших об'єктів – змінюваними. Кортеж у Python є, по суті, незмінним списком. У багатьох випадках кортежі працюють швидше списків, тому якщо ви не плануєте змінювати послідовність, то краще використовувати саме їх.

Мова має чіткий і послідовний синтаксис, продуману модульність й масштабованість, завдяки чому вихідний код написаних на Python програм легко читаємий. Python – стабільна й розповсюджена мова. Він використовується в багатьох проектах і в різних якостях: як основна мова програмування або для створення розширень і інтеграції застосунків. На Python реалізоване велика кількість проектів, також він активно використовується для створення прототипів майбутніх програм. Python використовується в багатьох великих компаніях.

2.3 Розгорнута постановка завдання

Згідно з технічним завданням на випускну кваліфікаційну роботу за першим (бакалаврським) рівнем вищої освіти, реалізації підлягає програмне забезпечення, яке призначено для системи віртуалізації з підтримкою розподіленого сховища.

В процесі розробки випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти необхідно виконати наступний обсяг роботи:

а) провести аналіз існуючих систем-аналогів для виявлення їх позитивних і негативних якостей. Результати аналізу врахувати в подальших розробках;

б) вибрати та обґрунтувати методику побудови системи контролю роботи технологічного обладнання на виробництві в автоматизованому режимі. Розробити функціональну та структурну схеми системи;

в) розробити програмне забезпечення системи, що дозволить реалізувати поставлену технічним завданням задачу. Побудувати блок-схеми алгоритмів програми та підпрограми;

г) організувати інтерфейс користувача з метою формування та виводу на екран ЕОМ повідомлень про некоректні дії користувача та нестандартні ситуації в роботі технологічного обладнання;

д) розробити рекомендації по організаційних та методичних заходах, які забезпечать впровадження системи в промислову експлуатацію та її подальшу успішну експлуатацію;

е) провести розрахунки по визначенню економічної ефективності розробленої системи;

ж) розробити заходи по охороні праці при впровадженні та експлуатації системи, а також розробити заходи з цивільного захисту;

з) сформувати висновки про виконаний обсяг робіт та одержані результати.

3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

3.1 Опис функціонування системи

Віртуалізація серверів – технологія, що дозволяє запустити кілька віртуальних серверів у рамках одного фізичного сервера.

Якщо із составом засобів віртуалізації все ясно, то із традиційними характеристиками IT-сегментів – обсягами продажів у грошовому вираженні – ясності як не було, так і немає. Протягом всіх шести років проведення досліджень даного ринку Gartner не давала кількісних оцінок (як абсолютних, так і по темпах росту). Зрозуміло, у цьому важко дорікнути аналітиків, оскільки даний пробіл лише відбиває специфіку віртуалізаційного напрямку: його дуже складно виділити із загальної IT-середовища й порахувати. Технології віртуалізації сильно інтегровані в інфраструктурні IT-продукти, до того ж отут досить великий приватна безкоштовних пропозицій. Коротше кажучи, віртуалізація – це саме той випадок, коли для розуміння щирої ситуації і її динаміки потрібно використовувати натуральні характеристики.

По даним Gartner, сьогодні близько 75% робочого навантаження для серверної x 86-інфраструктури виконується на віртуалізованих комп'ютерах (рік назад говорилося про 70%, а два роки тому – 66%), і, швидше за все, це не межа для росту цієї приватні. Основна частина віртуалізованого середовища доводиться на віртуальні машини (технологія гіпервізорів), але останнім часом спостерігається помітне підвищення інтересу замовників і провайдерів до використання контейнерів – багато в чому завдяки росту рівня зрілості хмарних обчислень (боротьба за підвищення ефективності, стандартизація сервісів, ріст числа й спектра SaaS-пропозицій і пр.).

Істотна частина віртуалізованої IT-інфраструктури доводиться на IaaS-провайдерів, близько 20% всіх ВМ надається користувачам через публічні IaaS-

сервіси. У цьому сегмент за останній рік спостерігається вибухообразний ріст: у торішньому огляді про 6%, при тому що в 2011 р. ця приватна була 3%. В 2014 р., по оцінках Gartner, число нових ВМ у публічних хмарах уперше перевищило аналогічний показник для On-Premise-розвертування. Причому більшість створених ВМ використовуються для нових обчислювальних навантажень, а не для переносу в них On-Premise-обрахунків. У корпоративному сегменті, у якому переважає застосування успадкованих додатків, динаміка проникнення віртуалізації помітно нижче. У цілому можна констатувати, що основним драйвером у сфері віртуалізації є нові хмарні додатки й використання, що розширюється, методів гнучкої (agile) розробки ПЗ, при цьому прискореними темпами росте попит на полегшені технології віртуалізації, у тому числі – на контейнери.

Значимість ринку приватних хмар також росте. Спочатку замовники отут ішли по шляху створення On-Premise-інфраструктур, при цьому багато хто орієнтувалися на застосування віртуалізаційних технологій Open Source, у тому числі контейнерні. Зараз же більшість підприємств переходять або вже перейшли до роботи з безліччю різних архітектур (публічні й приватні хмари, On-Premise), що працюють на базі різних технологій віртуалізації. Приватна застосування моновіртуалізації на великому корпоративному ринку прагне до нуля. Вартість продовжує залишатися важливим фактором при прийнятті рішень про впровадження віртуалізації, але все-таки на передній план зараз виходять питання інтероперабельності й застосування хмарних моделей.

Gartner відзначає, що віртуалізація серверної x 86-інфраструктури є фундаментом, на якому базуються два вкрай важливих для всього IT-ринку напрямку його розвитку (причому взаємозалежних і в істотній мері пересічних): модернізація внутрішньої IT-інфраструктури замовників і хмарні обчислення. У першому випадку за допомогою віртуалізації вирішуються питання підвищення ефективності використання ресурсів, зниження вартості (як капітальних витрат, так і операційних), оптимізації споживання енергії, підвищення швидкості

розгортання нових ресурсів, загального підвищення рівня автоматизації IT. Значна частина серверного навантаження використовується для обслуговування хостуємих віртуальних десктопів (hosted virtual desktop, HVD), але оскільки саме в цьому напрямку віртуалізації спостерігається найвищий рівень консолідації (число віртуальних машин на один сервер), відсоток фізичних серверів, зайнятих під HVD, не настільки великий. При модернізації IT-інфраструктури віртуалізація виступає як загальний горизонтальний тренд, що торкає практично всіх підприємств (незалежно від розмірів і галузей) і всі обчислювальні робочі навантаження.

3.2 Розробка структурної схеми

У даній роботі підтримується в одному продукті обидва види серверної віртуалізації (віртуальні машини й контейнери) і віртуальне сховище.

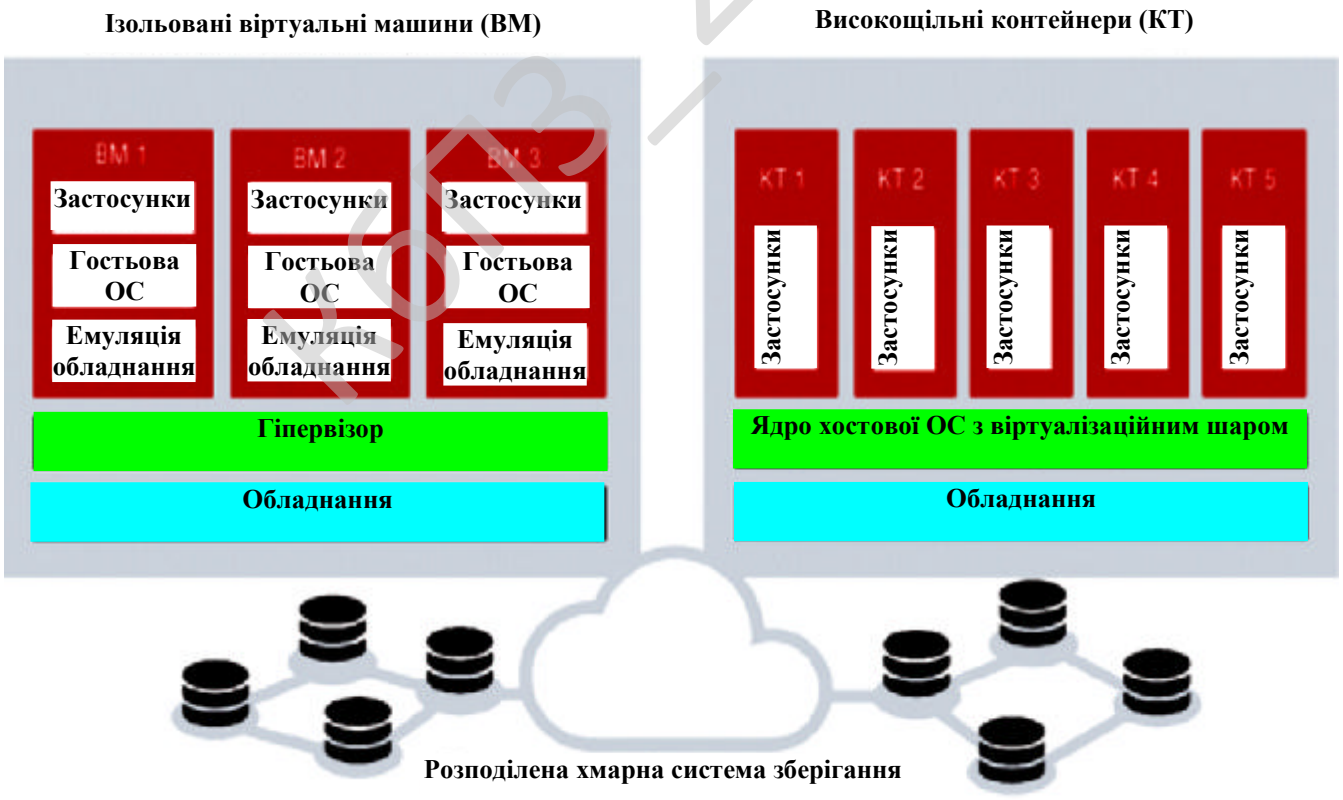

Рисунок 3.1 – Структурна схема системи.

Важливість віртуалізації для великих даних

Вирішення проблем із великими даними зазвичай потребує керування великими обсягами розподілених сховищ даних разом із використанням програм, що потребують інтенсивних обчислень і даних. Тому вам потрібне високоефективне IT-середовище для підтримки великих даних. Віртуалізація забезпечує додатковий рівень ефективності, щоб зробити платформи великих даних реальністю. Хоча віртуалізація технічно не є обов'язковою вимогою для аналізу великих даних, програмні інфраструктури, такі як MapReduce, які використовуються в середовищах великих даних, більш ефективні у віртуалізованому середовищі. Якщо вам потрібно масштабувати ваше середовище великих даних – майже без обмежень – вам слід віртуалізувати елементи вашого середовища. Віртуалізація має три характеристики, які підтримують масштабованість і ефективність роботи, необхідні для середовищ великих даних:

– Поділ: у віртуалізації багато програм і операційних систем підтримуються в одній фізичній системі шляхом поділу (розділення) доступних ресурсів.

– Ізоляція: кожна віртуальна машина ізольована від фізичної системи хоста та інших віртуалізованих машин. Завдяки цій ізоляції, якщо один віртуальний екземпляр аварійно завершує роботу, це не впливає на інші віртуальні машини та хост-систему. Крім того, дані не обмінюються між одним віртуальним екземпляром.

– Інкапсуляція: віртуальну машину можна представити (і навіть зберегти) як один файл, тож ви можете легко ідентифікувати її на основі послуг, які вона надає. Наприклад, файл, що містить інкапсульований процес, може бути повною бізнес-службою. Ця інкапсульована віртуальна машина може бути представлена додатку як повна сутність. Таким чином, інкапсуляція може захистити кожну програму, щоб вона не заважала іншим програмам.

Однією з найважливіших вимог для успіху з великими даними є належний рівень продуктивності для підтримки аналізу великих обсягів і різноманітних типів даних. Коли ви починаєте використовувати такі середовища, як Hadoop і MapReduce, дуже важливо, щоб у вас була допоміжна інфраструктура, яка може масштабуватися. Віртуалізація підвищує ефективність на кожному рівні IT-інфраструктури. Застосування віртуалізації у вашому середовищі допоможе досягти масштабованості, необхідної для аналізу великих даних.

Впровадження віртуалізації за допомогою наскрізного підходу забезпечить переваги для великих даних та інших типів робочих навантажень у вашому середовищі. Наскрізний підхід означатиме, що помилки можна виправляти швидше – вимога в середовищі великих даних. Під час роботи з великими даними ваша інфраструктура має бути готова керувати потенційно дуже великими (об'єм), дуже швидкими (швидкість) і дуже неструктурованими (різноманітність) даними.

У результаті все ваше IT-середовище потрібно оптимізувати на кожному рівні, від мережі до баз даних, сховищ і серверів. Якщо ви віртуалізуєте лише свої сервери, ви можете зіткнутися з вузькими місцями через інші елементи інфраструктури, такі як сховище та мережі. Якщо ви зосереджуєтеся лише на віртуалізації одного елемента вашої інфраструктури, ви з меншою ймовірністю досягнете необхідної затримки та ефективності, а також, швидше за все, піддасте свою компанію вищим витратам і ризикам для безпеки.

Реальність така, що більшість організацій не намагаються віртуалізувати всі елементи своєї інфраструктури одночасно. Багато організацій починають з віртуалізації серверів і досягають певного рівня підвищення ефективності. Реально інші елементи можуть бути віртуалізовані, якщо це необхідно, щоб продовжувати покращувати загальну продуктивність і ефективність системи. Нижче описано, як віртуалізація кожного елемента IT-середовища – серверів, сховищ, програм, даних, мереж, процесорів, пам'яті та служб – може позитивно вплинути на аналіз великих даних.

Віртуалізація сервера

У віртуалізації серверів один фізичний сервер розділений на кілька віртуальних серверів. Апаратне забезпечення та ресурси машини – включаючи оперативну пам'ять (RAM), ЦП, жорсткий диск і мережевий контролер – можна віртуалізувати (логічно розділити) на ряд віртуальних машин, кожна з яких запускає власні програми та операційну систему. Віртуальна машина (VM) – це програмне представлення фізичної машини, яка може виконувати або виконувати ті самі функції, що й фізична машина. Тонкий шар програмного забезпечення фактично вставляється в апаратне забезпечення, яке містить монітор віртуальної машини або гіпервізор. Гіпервізор можна розглядати як технологію, яка керує трафіком між віртуальними машинами та фізичною машиною.

Віртуалізація сервера використовує гіпервізор для забезпечення ефективності використання фізичних ресурсів. Звичайно, встановлення, налаштування та адміністративні завдання пов'язані з налаштуванням цих віртуальних машин. Це включає в себе керування ліцензіями, керування мережею та адміністрування робочого навантаження, а також планування потужності.

Віртуалізація сервера допомагає гарантувати, що ваша платформа може масштабуватися за потреби для обробки великих обсягів і різноманітних типів даних, які включені в аналіз великих даних. Ви можете не знати обсягу чи різноманітності структурованих і неструктурованих даних, необхідних перед початком аналізу. Ця невизначеність робить потребу у віртуалізації сервера ще більшою, надаючи вашому середовищу можливість задовольнити непередбачений попит на обробку дуже великих наборів даних.

Крім того, віртуалізація серверів забезпечує основу, яка дозволяє використовувати багато хмарних служб як джерела даних для аналізу великих даних. Віртуалізація підвищує ефективність хмари, що полегшує оптимізацію багатьох складних систем. У результаті організації мають продуктивність і оптимізацію, щоб мати доступ до даних, які раніше були недоступні або дуже важко зібрати. Платформи великих даних все частіше використовуються як

джерела величезної кількості даних про вподобання, настрої та поведінку клієнтів. Компанії можуть інтегрувати цю інформацію з внутрішніми даними про продажі та продукцію, щоб отримати уявлення про вподобання клієнтів і робити більш цілеспрямовані та персоналізовані пропозиції.

Віртуалізація програми

Віртуалізація інфраструктури додатків забезпечує ефективний спосіб керування додатками в контексті попиту клієнтів. Програма інкапсульована таким чином, що видаляє її залежності від основної фізичної комп'ютерної системи. Це допомагає покращити загальну керованість і портативність програми. Крім того, програмне забезпечення для віртуалізації інфраструктури додатків зазвичай дозволяє кодифікувати політику ділового та технічного використання, щоб переконатися, що кожна ваша програма використовує віртуальні та фізичні ресурси передбачуваним способом. Ефективність досягається завдяки тому, що ви можете легше розподіляти IT-ресурси відповідно до відносної бізнес-цінності ваших програм. Іншими словами, ваші найбільш критичні програми можуть отримати найвищий пріоритет, щоб за потреби використати пули доступних обчислювальних ресурсів і сховищ.

Віртуалізація інфраструктури додатків, яка використовується в поєднанні з віртуалізацією сервера, може допомогти забезпечити дотримання бізнес-угод про рівень обслуговування (SLA). Віртуалізація сервера відстежує використання процесора та пам'яті, але не враховує варіації бізнес-пріоритету під час розподілу ресурсів. Наприклад, ви можете вимагати, щоб усі програми оброблялися з однаковим пріоритетом бізнес-рівня. Впровадивши віртуалізацію інфраструктури додатків на додаток до віртуалізації сервера, ви можете гарантувати, що програми з найбільшим пріоритетом матимуть першочерговий доступ до ресурсів.

Ваші програми для великих даних можуть мати значні потреби в IT-ресурсах через великі обсяги даних або швидкість, з якою ці дані генеруються. Ваше середовище великих даних має мати належний рівень передбачуваності та

повторюваності, щоб переконатися, що програми мають доступ до необхідних ресурсів. Віртуалізація інфраструктури додатків може гарантувати, що кожна програма, розгорнута для аналізу великих даних, матиме доступ до необхідної обчислювальної потужності в потрібний час на основі її відносного пріоритету. Крім того, віртуалізація інфраструктури додатків полегшує запуск додатків на різних комп'ютерах, а раніше несумісні або застарілі додатки можна запускати разом на одній фізичній машині. Вам не потрібно буде створювати кілька версій, наприклад Windows або Linux.

Платформи великих даних, розроблені для підтримки розповсюджених додатків із інтенсивним об'ємом даних, працюватимуть краще та швидше у віртуальному середовищі. Це не означає, що ви захочете віртуалізувати всі програми, пов'язані з великими даними. Наприклад, текстова аналітична програма може найкраще працювати в автономному середовищі, і віртуалізація не додасть жодних переваг.

Віртуалізація мережі

Віртуалізація мережі – програмно визначена мережа – забезпечує ефективний спосіб використання мережі як пулу ресурсів підключення. Мережі віртуалізуються подібно до інших фізичних технологій. Замість того, щоб покладатися на фізичну мережу для керування трафіком між з'єднаннями, ви можете створити декілька віртуальних мереж, використовуючи одну фізичну реалізацію. Це може бути корисним, якщо вам потрібно визначити мережу для збору даних із певним набором характеристик продуктивності та ємності та іншу мережу для додатків з різною продуктивністю та ємністю. Обмеження на мережевому рівні можуть призвести до вузьких місць, які призведуть до неприйнятних затримок у середовищах великих даних. Віртуалізація мережі допомагає зменшити ці вузькі місця та покращити можливості керування великими розподіленими даними, необхідними для аналізу великих даних.

Віртуалізація процесора та пам'яті

Віртуалізація процесора допомагає оптимізувати процесор і максимізувати продуктивність. Віртуалізація пам'яті відокремлює пам'ять від серверів.

Під час аналізу великих даних у вас можуть бути повторні запити до великих наборів даних і створення розширених аналітичних алгоритмів, розроблених для пошуку закономірностей і тенденцій, які ще не зрозумілі. Ця розширена аналітика може вимагати багато процесорної потужності (CPU) і пам’яті (RAM). Для деяких із цих обчислень може знадобитися багато часу без достатніх ресурсів ЦП і пам’яті. Віртуалізація процесора та пам’яті може допомогти пришвидшити обробку та швидше отримати результати аналізу.

Віртуалізація даних і сховищ

Віртуалізацію даних можна використовувати для створення платформи для динамічних пов’язаних служб даних. Це дозволяє легко шукати дані та зв’язувати їх через уніфіковане довідкове джерело. У результаті віртуалізація даних надає абстрактну послугу, яка надає дані в узгодженій формі незалежно від основної фізичної бази даних. Крім того, віртуалізація даних надає кешовані дані всім програмам для підвищення продуктивності.

Віртуалізація сховищ об’єднує фізичні ресурси сховища для більш ефективного їх спільного використання. Це зменшує вартість зберігання та полегшує керування сховищами даних, необхідними для аналізу великих даних.

Віртуалізація даних і сховищ відіграє важливу роль у полегшенні та здешевленні зберігання, отримання й аналізу великих обсягів швидких і різноманітних типів даних. Пам’ятайте, що деякі великі дані можуть бути неструктурованими і їх важко зберігати за допомогою традиційних методів. Віртуалізація сховища полегшує зберігання великих і неструктурованих типів даних. У середовищі великих даних вигідно мати доступ до різноманітних сховищ оперативних даних за запитом. Наприклад, доступ до стовпцевої бази даних може знадобитися лише рідко. За допомогою віртуалізації базу даних

можна зберігати як віртуальний образ і викликати її, коли це необхідно, не споживаючи цінні ресурси або потужність центру обробки даних.

Керування віртуалізацією за допомогою гіпервізора

В ідеальному світі ви не хочете турбуватися про базову операційну систему та фізичне обладнання. Гіпервізор – це технологія, яка забезпечує впорядкований і повторюваний обмін ресурсами. Саме дорожній поліцейський дозволяє кільком операційним системам спільно використовувати один хост. Він створює та запускає віртуальні машини. Гіпервізор розташований на найнижчих рівнях апаратного середовища та використовує тонкий шар коду (часто званий структурою), щоб забезпечити динамічний спільний доступ до ресурсів. Завдяки гіпервізору створюється враження, що кожна операційна система має власні фізичні ресурси.

У світі великих даних вам може знадобитися підтримувати багато різних операційних середовищ. Гіпервізор стає ідеальним механізмом доставки технологічних компонентів стеку великих даних. Гіпервізор дозволяє показувати ту саму програму на багатьох системах без необхідності фізично копіювати цю програму в кожну систему. Як додаткова перевага, завдяки архітектурі гіпервізора він може завантажувати будь-які (чи багато) різні операційні системи так, ніби вони були просто черговою програмою. Таким чином, гіпервізор є дуже практичним способом віртуалізації речей швидко та ефективно

Потрібно розуміти природу гіпервізора. Він розроблений як серверна ОС, а не як ОС Windows. Кожна віртуальна машина, що працює на фізичній машині, називається гостьовою машиною. Таким чином, гіпервізор планує доступ гостьових операційних систем до всього, включаючи ЦП, пам’ять, дисковий ввід-вивід та інші механізми вводу-виводу. Гостьові операційні системи – це операційні системи, що працюють на віртуальних машинах. Завдяки технології віртуалізації ви можете налаштувати гіпервізор для розподілу ресурсів фізичного комп’ютера. Наприклад, ресурси можна розподілити 50/50 або 80/20 між двома гостьовими операційними системами.

Принадність цього розташування полягає в тому, що гіпервізор виконує всю важку роботу. Гостьовій операційній системі байдуже (або не має жодного уявлення), що вона працює у віртуальному розділі; він думає, що має лише комп'ютер.

В основному можна знайти два типи гіпервізорів:

– Гіпервізори типу 1 працюють безпосередньо на апаратній платформі. Вони досягають вищої ефективності, оскільки працюють безпосередньо на платформі.

– Гіпервізори типу 2 працюють на головній операційній системі. Вони часто використовуються, коли існує потреба підтримувати широкий діапазон пристроїв введення/виведення.

Абстракція та віртуалізація

Щоб IT-ресурси та послуги були віртуалізовані, вони відокремлені від базового середовища фізичної доставки. Технічний термін для цього акту розділення називається абстракція. Абстракція є ключовим поняттям у великих даних. MapReduce і Hadoop – це розподілене обчислювальне середовище, де все абстраговано. Деталі абстрагуються, тому розробнику або аналітику не потрібно турбуватися про те, де фактично розташовані елементи даних.

Абстракція мінімізує складність чогось, приховуючи деталі та надаючи лише відповідну інформацію. Наприклад, якщо ви збираєтеся зустріти когось, кого раніше ніколи не бачили, він може сказати вам місце зустрічі, який він зріст, колір волосся та що він одягне. Йому не потрібно розповідати, де він народився, скільки має грошей у банку, дату народження тощо. Це ідея з абстракцією – мова йде про надання високорівневої специфікації, а не про вдавання в подробиці того, як щось працює. У хмарі, наприклад, у моделі доставки «Інфраструктура як послуга» (IaaS) деталі фізичної та віртуальної інфраструктури абстрагуються від користувача.

Впровадження віртуалізації для роботи з великими даними

Віртуалізація допомагає зробити ваше IT-середовище достатньо розумним для аналізу великих даних. Оптимізуючи всі елементи вашої інфраструктури, включаючи апаратне забезпечення, програмне забезпечення та сховище, ви отримуєте ефективність, необхідну для обробки та керування великими обсягами структурованих і неструктурованих даних. З великими даними вам потрібно отримувати доступ до структурованих і неструктурованих даних, а також аналізувати їх у розподіленому середовищі.

Великі дані передбачають поширення. На практиці будь-який вид MapReduce працюватиме краще у віртуалізованому середовищі. Вам потрібна можливість переміщати робочі навантаження на основі вимог до обчислювальної потужності та пам’яті.

Віртуалізація дозволить вам вирішувати великі проблеми, які ще не охоплені. Ви можете не знати заздалегідь, як швидко вам потрібно буде масштабуватись.

Віртуалізація дозволить вам підтримувати різноманітні робочі сховища великих даних. Наприклад, базу даних графів можна розгорнути як зображення.

Найпрямішою перевагою віртуалізації є забезпечення кращої роботи механізмів MapReduce. Віртуалізація призведе до кращого масштабу та продуктивності MapReduce. Кожне із завдань Map і Reduce потрібно виконувати окремо. Якщо механізм MapReduce розпаралелено та налаштовано для роботи у віртуальному середовищі, ви можете зменшити витрати на керування та дозволити розширення та скорочення робочих навантажень завдань. MapReduce сам по собі є паралельним і розподіленим. Інкапсулюючи механізм MapReduce у віртуальний контейнер, ви можете запускати те, що вам потрібно, коли завгодно. За допомогою віртуалізації ви збільшуєте використання ресурсів, за які ви вже заплатили, перетворюючи їх на загальні пули ресурсів.

3.3 Розробка функціональної схеми

У даній платформі віртуалізації з підтримкою розподіленого сховища пропонується використовувати протокол Transmission Control Protocol (ARTCP). Запропонований у даній роботі, удосконалений протокол Adaptive Rate Transmission Control Protocol (ARTCP), позичає деякі механізми від протоколу TCP. В ARTCP повністю переглянутий алгоритм керування потоком, що і відрізняє його від TCP. Разом з тим, запропонований протокол може забезпечувати сумісність із TCP.

Аспекти новизни протоколу ARTCP

ARTCP відрізняється від стандартного TCP тим, що сегменти відправляються в мережу не у вигляді сплеску в межах вікна, а розділені часовими проміжками, тривалість яких визначається поточним значенням швидкості. Швидкість потоку регулюється не розміром змінного вікна, а значенням швидкості, зміною якої здійснюється адаптація алгоритму відповідно до умов. Механізм ковзного вікна в ARTCP застосовується тільки для запобігання переповнення буферів одержувача. На розмір вікна в ARTCP не накладено ніяких обмежень у жодному з режимів роботи. Вікно обмежене лише наявністю буферного простору в одержувача, тому для одержувача рекомендується встановлювати вікно на максимальний розмір.

У випадку, коли одержувач обмежений у буферному просторі, ARTCP буде поводитися як звичайний протокол TCP, обмежений вікном. Таким чином, протокол ARTCP сполучить у собі метод ковзного вікна для регулювання керування потоком від краю до краю й метод контролю швидкості для підстроювання під ПрЗд проміжних вузлів з'єднання.

Другою найважливішою відмінністю ARTCP є те, що як сигнал про стан перевантаження або наявності додаткових ресурсів у мережі використовуються темпоральні характеристики потоку – вимір шпаруватості потоку в одержувача й зміни часу RTT. Втрата пакета ніяк не відбивається на роботі ARTCP крім

здійснення ретрансляції загубленого пакета механізмом корекції помилок. Як і в TCP об втрати пакета повідомляють дві можливих події: спрацьовування ТПП або послідовне одержання двох підтверджень тих самих даних.

Таким чином, у порівнянні зі своїм попередником, ARTCP має наступні переваги:

1. ARTCP не потрібно доводити мережа до стану перевантаження, щоб визначити доступну частку ПрЗд, тому виключені втрати пакетів пов'язані із цим процесом.

2. ARTCP істотно знижує вимоги до міжмережних пристроїв. По-перше, для нормального функціонування даного протоколу потрібен менший об'єм буферного простору, ніж для TCP, оскільки режим передачі є згладженим. По-друге, ARTCP не вимагає й не залежить від наявності яких або механізмів диспетчеризації або керування чергами, таких як RED або WFQ.

3. ARTCP не інтерпретує втрату пакета як ознаку перевантаження мережі, використовуючи замість цього темпоральні характеристики потоку. Тому ARTCP повинен особливо ефективно працювати в системах бездротового зв'язку, там де використання TCP неефективно.

4. На відміну від TCP новий протокол не покладається цілком на потік підтверджень у зворотному напрямку для синхронізації процесу передачі. У зв'язку із цим можлива реалізація ARTCP з меншою частотою підтверджень, що не обмежувала б швидкість в асиметричних системах.

Евристика в основі алгоритму ARTCP

Аналіз робіт в області транспортних протоколів, дозволив укласти, що недоліки протоколу TCP досить істотні і є наслідком самого алгоритму, що лежить в основі протоколу TCP. Тому модифікація TCP без заміни його основних алгоритмів не може привести до істотного поліпшення характеристик протоколу.

Тому в цій роботі було вирішено створити вдосконалений алгоритм транспортного протоколу, що залишається, однак, повністю сумісним з архітектурою TCP/IP.

Для того щоб усунути недоліки, властиві TCP, необхідно було знайти спосіб одержання інформації про стан мережі, відмінний від застосування в цих цілях втрат сегментів. Найбільше добре на роль індикатора стану мережі підходять часові характеристики потоку: час RTT і міжсегментні інтервали. З використанням міжсегментних інтервалів можна також визначити частку ПрЗд каналу. Для цього потрібно запам'ятовувати міжсегментні інтервали потоку у відправника й вимірювати їх в одержувача. Порівняння значень інтервалів характеризує стан мережі, а мінімальне значення вимірюваних інтервалів в одержувача дозволяє визначити доступну частку ПрЗд.

Таким чином, виходить наступна схема: установка швидкості потоку відправником за допомогою ретельної диспетчеризації сегментів, вимір швидкості прибуття потоку в одержувача й передача цієї інформації відправникові разом з іншою контрольною інформацією. Різниця старого й нового значень швидкості відправлення потоку ARTCP на кожному кроці задається випадковою змінною, однак, при наявності сигналу про перевантаження мережі ймовірність зниження швидкості перевищує ймовірність її збільшення на кожному новому кроці.

Параметри й змінні

Нехай τ часовий інтервал між послідовними трансляціями пакетів. Задача функції диспетчеризації сегментів у тому, щоб затримувати відправлення чергового сегмента на час τ після початку передачі попереднього сегмента. Позначимо всі змінні, відносні до відправника індексом s, і r – відносні до одержувача. Отже, τ часовий інтервал між моментами початку відправлення в мережу сегмента i+1 і i-го, а τ_r – інтервал між послідовно прибулими до одержувача сегментами.

Нехай швидкість каналу зв'язку, безпосередньо до якого підключений відправник – R_{ls}, тоді час який потрібен на відправлення одного сегмента (з моменту початку передачі до моменту її закінчення) $t_{ls} = S / R_{ls}$, де s – розмір переданого сегмента.

Очевидно, що максимально можлива швидкість потоку:

$$R^{\max}{}_s = R_{ls} = S / \tau^{\min}{}_s. \quad (3.1)$$

Мінімальне значення міжсегментного інтервалу в цьому випадку буде $\tau^{\min}{}_s = t_{ls}$, коли пакети відправляються в мережу без затримок з максимальною швидкістю канального рівня. Шляхом зміни τ у межах $[\tau^{\min}, \infty)$, ARTCP може контролювати швидкість потоку в межах $[R_{ls}, 0)$. Ілюстрація принципу керування швидкістю наведена на рисунку 3.1.

Затримка відправлення готових сегментів виробляється за допомогою системного таймера. Наприклад, для повного використання ПрЗд каналу в 512 Кб/с при розмірі сегмента в 1000 байт кожний сегмент необхідно відправляти із затримкою τ =0.015625 с, щоб швидкість потоку склала 64 пакета/с.

У таблиці 3.1 наведений список параметрів і змінних використовуваних алгоритмом керування потоком протоколу ARTCP.

Формат повідомлення

Формат повідомлень використовуваних ARTCP може в точності збігатися з форматом пакета TCP.

Стандарт TCP [4] передбачає наявність додаткових полів у заголовку сегмента між стандартним заголовком і полем даних. ARTCP може передавати додаткову інформацію в цих полях, що буде гарантувати сумісність із TCP. Усього протокол ARTCP вимагає використання лише двох нових полів: значення попереднього порядкового номера "PS" у напрямку від відправника до одержувача й значення шпаруватості "TI" у напрямку від одержувача до відправника. Значення "TI" можна передавати у вигляді опції часової мітки [6], а значення "TI" вимагає поля, що дозволяє помістити порядковий номер сегмента.

Диспетчеризація сегментів

Диспетчер сегментів відправляє сегменти на лінію через строго задані міжсегментні часові інтервали. Значення інтервалів визначаються швидкістю відправлення потоку, що задається функцією адаптації. Меншій швидкості відповідає більш тривалий час передачі.

Таблиця 3.1 – Список параметрів і змінних

s	Розмір кадру канального рівня утримуючий сегмент
τ_s	Часовий проміжок між послідовними трансляціями сегментів (від першого біта до першого біта)
τ_R	Часовий проміжок, обмірюваний між послідовними моментами прибуттями сегментів (від останнього біта до останнього біта)
$R_S(t)$	Швидкість потоку, установлювана відправником
$R_R(t)$	Швидкість потоку, що обчислюється одержувачем
$R_e(t)$	Оцінка доступної пропускної здатності в момент t-*RTT*
R'_S	Перша похідна швидкості за часом, установлюється відправником
R_{pe}	Значення оцінки доступної пропускної здатності в момент *i-1*
A_c	Площа області компенсації
SSGR	Параметр експонентного росту R'_S у режимі SS
RTT	Часовий проміжок між моментом відправлення сегмента й моментом приходу його підтвердження
ERTT	Зважене ковзне середнє RTT
SmoothedRe	Зважене ковзне середнє Re
MaxERTT	Максимальне значення згладженого RTT
MinERTT	Мінімальне значення згладженого RTT
MDFACTOR	Коефіцієнт, використовуваний при мультиплікативному зниженні $R_s(t)$
R_S^{amd}	Значення швидкості відправлення даних безпосередньо на виході режиму MD1
R_S^{bmd}	Значення швидкості відправлення даних безпосередньо перед входом у режим MD1
R_e^{amd}	Значення оцінки доступної пропускної здатності в режимі MD1
speedup	Коефіцієнт, що визначає ймовірність збільшення швидкості в режимі FT
slowdown	Коефіцієнт, що визначає ймовірність зниження швидкості в режимі FT
Midpoint	Точка відліку швидкості на кожному кроці в режимі FT
sR_S	Зважене ковзне середнє значення R_s
K	Коефіцієнт критерію здійснення переходу
BER	Bit error ratio – резидентна ймовірність інвертування біта на лінії
R_S^{min}	Початкове мінімальне значення швидкості, з якого починається ріст у стані SS

Вимір швидкості

Очевидно, що швидкість прийому потоку одержувачем не може бути вище швидкості обслуговування потоку на ділянці з найменшої ПрЗд, через яку проходить з'єднання. Таким чином, знаючи швидкість прибуття потоку до одержувача, можна визначити доступну пропускну здатність мережі. Для коректного виміру швидкості необхідно не враховувати випавші з потоку, тобто загублені сегменти, а також сегменти, що доставляються мережею в зміненому порядку. Для виконання цієї умови в поле "PS" кожного сегмента, що відправляється, записується порядковий номер (або зсув) від попереднього сегмента.

Одержавши сегмент *i*, одержувач обчислює різницю поточного часу й часу прибуття попереднього (*j*) сегмента τ_R і у випадку, якщо поле "PS" *i-го* сегмента містить значення *j*, поміщає $R_r = s/\tau_R$ у поле "TI" підтвердження наступного в протилежному напрямку (рисунок 3.1). Одержувач витягає значення поля "TI" з одержуваних підтверджень і використовує його для керування швидкістю передачі.

Функція адаптації

Опис алгоритму роботи функції адаптації, безпосередньо здійснюючe керування потоком ARTCP, було дано в роботах [52,53,54,76].

Спосіб керування потоком повинен досягти, по-перше, швидкої реакції потоку на умови, що змінюються, з'єднання й, по-друге, стабілізувати швидкість передачі, коли вона дорівнює максимальної швидкості мережі. Алгоритм керування потоками ARTCP функціонує в декількох режимах.

Робота ARTCP починається з режиму швидкого збільшення швидкості, аналогічної механізму вповільненого старту стандартного TCP, для максимально швидкого досягнення з'єднанням верхньої межі доступної пропускної здатності. Після того, як верхня межа досягнута, алгоритм ARTCP переходить у режим точного настроювання, протягом якої втримує швидкість на рівні доступної ПрЗд. У випадку визначення зменшення доступної ПрЗд, ARTCP робить

мультиплікативне зниження швидкості, що у випадку тривалого стану перевантаження триває експоненційно. Отже, адаптація швидкості передачі потоку протоколом ARTCP відбувається в п'яти режимах (рисунок 3.2).

Режим прискореного старту (*SS*) має мету максимально швидко збільшити швидкість потоку від мінімального значення до значення, рівного або переважаючого ПрЗд каналу відразу після ініціалізації з'єднання. Для цього швидкість збільшується експоненційно:

$$R'(t+RTT) = R_S(t) \times SSGR, \tag{3.2}$$

$$R_S(t_i) = R_S(ti\text{-}1) + RS(t,) \times (t_i - ti\text{-}1). \tag{3.3}$$

Початкове значення швидкості встановлюється після синхронізації з'єднання як:

$$R_S^{\min} = \frac{SEGSIZE}{\min RTT}. \tag{3.4}$$

Вихід з режиму ***SS*** відбувається, коли $R_e(t_i) < (1-\varepsilon) \times R_S(t_i - \text{RTT})$. Після реалізації переходу 2, алгоритм переходить у стан мультиплікативного скидання **MD1**.

Режим мультиплікативного скидання (MD1) слідує за режимом ***SS***. Після виходу з ***SS*** значення $R_s(t)$ буде перевищувати $R_e(t)$, тому в режимі **MD1** швидкість потоку стрибкоподібно встановлюється свідомо нижче $R_e(t)$:

$$R_S(t_i) = R_e(t_i) - MDFACTOR \times (R_S(\text{ti}\text{-}1) - R_e(t_i)) \tag{3.5}$$

Після зниження швидкості алгоритм переходить у режим відновлення.

Режим відновлення (REC) має на меті, лінійно збільшуючи швидкість, довести її до вже відомого значення ПрЗд каналу: $R_e(t)$, компенсуючи виниклу в режимі ***SS*** перевантаження. У режимі **REC** обчислюється значення площі області компенсації $A_c(t_t)$ як площі фігури, утвореної значеннями $R_s(t)$ над прямій $R_e(t_i)$ за час, поки $R_s(t_t) > R_e(t_t)$ у режимі ***SS***:

Значення площі області компенсації дорівнює сумі площ набору трапецій утворених значеннями $R_s(t)$ над прямій $R_e(t_i)$. Площа кожної трапеції:

$$S_T = \Delta t_i / 2 \ ((2R_S(t_i) - R'(t_i) \times \Delta t_i - 2R_e(t_i)), \tag{3.6}$$

де Δt_i час, протягом якого не відбувалося змін значення $R'(t_i)$.

$$A_c(t_i) = \frac{1}{2}\Delta t_0\left[2R_s(t_i) - \Delta t_0 R_s^{\cdot}(t_i) - 2R_e(t_i)\right] +$$
$$+\frac{1}{2}RTT\left[2R_s(t_i) - \frac{R_s^{\cdot}(t_i)}{SSGR} - RTT\frac{R_s^{\cdot}(t_i)}{SSGR} - R_e(t_i)\right] +$$
$$+\frac{1}{2}RTT\left[2R_s(t_i) - \frac{R_s^{\cdot}(t_i)}{SSGR^2} - RTT\frac{R_s^{\cdot}(t_i)}{SSGR^2} - R_e(t_i)\right] + , \qquad (3.7)$$
$$+ \ldots +$$
$$+\frac{1}{2}\frac{\left[R_s(t_i) - \frac{R_s^{\cdot}(t_i)}{SSGR^N} - R_e(t_i)\right]^2}{\frac{R_s^{\cdot}(t_i)}{SSGR^N}}$$

де Δt_0 проміжок часу між моментом t_i і моментом попередньої зміни $R'(_i)$ і N – індекс доданка, при якому:

$$R_s(t_i) - \frac{R_s^{\cdot}(t_i)}{SSGR^N} > R_e(t_i). \qquad (3.8)$$

Отже, перший доданок наведеної формули є площа трапеції з висотою меншої RTT, останній доданок – площа трикутника, що складаються від 2 до N-1 площі трапецій з висотою рівної RTT.

Значення $A_c(t_i)$ застосовується для визначення величини $R'_s(t_i)$ у стані відновлення **REC**. Ідея в тому, що із-за затримки інформації про стан мережі на час RTT, у стані швидкого збільшення швидкості відправлення сегментів потік викличе наповнення буферів мережі. Пакети будуть накопичуватися в мережі протягом часу, коли швидкість відправлення сегментів перевищує ПрЗд мережі. Перебування з'єднання в стані відновлення необхідно для того, щоб мережа впоралася з виниклої до зменшення швидкості відправлення перевантаженням. Очевидно, що кількість даних, що нагромадилися в буферах мережі, визначається площею області $A_c(t_i)$, тому швидкість відправлення сегментів у стані **REC** повинна бути знижена таким чином, щоб площа фігури DFG була дорівнює $A_c(t_i)$. Швидкість у стані **REC** міняється лінійно, обумовлена значенням:

$$R_s^{\cdot}(t_i) = \frac{\left[R_{pe}(t_i) - R_s(t_i)\right]^2}{2A_c(t_i)}. \quad (3.9)$$

У стані **REC** швидкість відправлення даних зростає лінійно від значення отриманого в попередній стадії **MD1** за законом:

$$R_s(t_i) = R_s^{amd} + t \times R_s^{\cdot}(t_i). \quad (3.10)$$

Вихід зі стану **REC** (перехід 4) здійснюється в тому випадку, коли:

$$R_s(t\ ) \geq R_e^{amd}. \quad (3.11)$$

Режим тонкого настроювання (**FT**) слідує за режимом **REC**. У режимі **FT** швидкість відправлення даних повільно підбудовується під ПрЗд каналу. Відношення коефіцієнтів *speedup* і *slowdown* у стані **FT** визначає ймовірність зниження або підвищення швидкості на кожному кроці. Коефіцієнт *speedup,* відповідальний за підвищення швидкості обернено пропорційний швидкості даного з'єднання. Коефіцієнт *slowdown,* відповідальний за зниження швидкості, пропорційний відношенню вимірюваного RTT до мінімального значення RTT. Значення *speedup* більше при менших значеннях $R_s(t)$, що дає повільним з'єднанням перевага для одержання доступу до більшої відносної частки ПрЗд. Значення *slowdown* однаково для всіх з'єднань і росте при росту RTT. Таким чином, імовірність підвищення швидкості для повільних з'єднань більше, а ймовірність зниження швидкості однакова для всіх з'єднань. Вихід з режиму **FT** відбувається у випадку стрибкоподібної зміни вимірюваного RTT.

У стані **FT** значення швидкості передачі визначаються за законом:

$$R_s(t_i) = midpoint + \left[2 \times Rand - \frac{slowdown}{speedup}\right] \times \frac{speedup}{slowdown} \times INTERVAL \times midpoint, \quad (3.12)$$

де *rand* – рівномірно розподілена випадкова величина, генеруєма функцією drand48 з областю значень [0; 1]. Після влучення в стан **FT** (реалізації переходу 4) значення *midpoint* установлюється рівним R_e^{amd}, надалі значення *midpoint* установлюється рівним sR_s. Змінні *speedup* і *slowdown* визначають напрямок зміни швидкості відправлення даних залежно від зміни часу RTT.

Коефіцієнти *slowdown* і *speedup* для використання у формулі (3.12) визначаються по наступних формулах:

$$speedup = \left[1 + \frac{S}{\min RTT \times sR_s}\right]^2, \qquad (3.13)$$

де другий доданок являє собою відношення мінімальної кількості даних у транзиті по з'єднанню (*s* байт за час RTT) до реальної їхньої кількості (добуток мінімального часу RTT на середню швидкість відправлення потоку). Відповідно ріст реальної швидкості потоку виражається в зменшенні значення коефіцієнта *s*peedup, таким чином, за інших рівних умов імовірність росту R_s для з'єднання з меншою швидкістю буде більше. За рахунок цього усувається нерівномірність використання ресурсів різними з'єднаннями.

Коефіцієнт *s*lowdown обчислюється в такий спосіб:

Якщо $RTT > minERTT*(1+PRECISION)$,

те $slowdown = 2 \times (RTT/minERTT) \times speedup$

інакше

$slowdown = 1$

Відношення *speedup/slowdown* визначає знак відхилення миттєвого значення швидкості від середнього. Якщо $speedup > slowdown$ то відхилення від середнього значення для миттєвого значення швидкості буде позитивним, тобто швидкість потоку буде збільшуватися. У випадку $speedup < slowdown$ швидкість потоку буде знижуватися. Також, у стані **FT** максимальне відхилення миттєвого значення швидкості відправлення пакетів від середнього за попередній період, відповідно до формули (3.12), пропорційно середньому значенню швидкості. У зв'язку із цим потік, роблячи перехід 4 у стані **FT** при більшому значенні оцінки доступної ПрЗд, пристосовується до невеликих змін ПрЗд більш інтенсивно.

Умовою переходу з **FT** у режим мультиплікативного скидання **MD2** (перехід 6) є:

$$ERTT > K \times (FT \max RTT - \min ERTT)/ \qquad (3.14)$$

Режим мультиплікативного скидання (**MD2**) необхідний для швидкого зниження швидкості за умови різкого росту RTT*:*

$$midpoint_i = midpoint_{i-1} \times MDFACTOR \quad (3.15)$$

Після цього протокол переходить у стан **FT**, реалізуючи перехід 7. У тому випадку, якщо умова (3.14) продовжує залишатися дійсною, то мультиплікативне зменшення триває, оскільки послідовність переходів 6 – 7 реалізується неодноразово, виражаючись в експонентному зменшенні швидкості передачі даних.

Завершення роботи протоколу може відбутися з будь-якого стану {***SS***, **REC**, **FT**} – переходи (8, 9, 10).

Сумісність із TCP

Система, що підтримує ARTCP, може бути також сумісна з TCP. Для цього, ініціатор з'єднання, що підтримує ARTCP, поміщає в заголовку синхронізуючого пакета опцію "PS". Наявність поля "PS" у заголовку SYN пакета повинне включати механізм ARTCP одержувача. Якщо така можливість є, то одержувач включає в сегмент SYN-ACK поле "TI", якщо ні, то відсутність опції "TI" у відповіді одержувача відключає механізм ARTCP в ініціатора й подальший обмін відбувається по протоколі TCP. Якщо ж опція "TI" є присутнім у відповіді, то ініціатор упевнений, що й одержувач задіяв механізм ARTCP.

Порівняння ARTCP і TCP на основі аналізу алгоритму

Протокол TCP не може обходитися без втрат сегментів, які створюються самим протоколом шляхом переповнення черги, і які обмежують подальший ріст швидкості TCP.

Теорема: Протокол ARTCP не створює втрат сегментів, якщо $\frac{Q_{\max}}{R} > \varepsilon$, де $Q_{\max}$ – максимальна довжина черги, ε – граничне значення, використовуване алгоритмом ARTCP, а R – швидкість каналу.

Доказ:

Втрата сегмента може відбутися лише у випадку надходженні чергового сегмента в чергу обслуговуючого пристрою, коли розмір сегмента перевищує різниця $Q_{max} - Q$.

Якщо D сума затримок передачі в каналах на шляху від відправника до одержувача, то в припущенні, що канали в системі симетричні й трафік передається лише в одному напрямку, мінімальний час *min*RTT визначається як $2D$. У випадку якщо середня швидкість відправлення потоку R_s перевищує швидкість R каналу, що обслуговує чергу, виникає черга сегментів у маршрутизаторі. Поява черги довжини Q приведе до збільшення часу RTT на величину Q/R. Однак для ARTCP, якщо різниця RTT-minRTT перевищує деяке граничне значення ε, імовірність зниження швидкості відправлення потоку R_s стане відмінної від нуля й швидкість потоку буде знижуватися, поки значення дозволеного перевищення RTT над *min*RTT не стане нижче граничного значення. Таким чином, виконується нерівність: $Q < \varepsilon \times R$. Вибираючи досить мале значення ε, алгоритм ARTCP гарантує, що довжина черги не перевищить певного значення, меншого, ніж максимальна довжина черги $Q < Q_{max}$. Для виконання цього, необхідно вибрати ε, так, щоб $\varepsilon < \frac{Q_{max}}{R}$. Отже, при такому виборі граничного значення, відсутність втрат сегментів при роботі ARTCP гарантовано.

Наслідок з теореми: при числі потоків більшому 1, протокол ARTCP на відміну від TCP, може бути настроєний так, щоб взагалі не створювати втрат сегментів.

Для протоколу TCP факт втрати сегмента служить індикатором виникнення перевантаження мережі. Значення ймовірності втрати сегмента визначає швидкість передачі, яка розвивається TCP з'єднанням. Будь-яка втрата сегмента викликає стрибкоподібне зменшення розміру вікна й, отже, зниження швидкості передачі TCP. В умовах, коли причиною втрат є винятково переповнення черги, зниження швидкості TCP при виникненні втрат приводить до того, що виконується рівність середньої швидкості TCP і швидкості

обслуговування каналу. Очевидно, що зниження швидкості внаслідок додаткових втрат, викликаних помилками передачі, приведе до того, що середня швидкість TCP потоку стане менше, ніж швидкість каналу. Оскільки TCP не може визначити причину втрати сегмента й реагує зниженням швидкості на будь-яку втрату, те його ефективність у мережах, де втрати сегментів можуть бути наслідком помилок передачі, буде тим менше, ніж вище ймовірність втрати сегмента.

Протокол ARTCP на відміну від TCP не знижує швидкість передачі потоку при виникненні втрати сегмента. Загублені дані ретранслюються, не роблячи впливу на швидкість передачі. Внаслідок цього, втрати сегментів не роблять впливу на швидкість потоку ARTCP.

Сказане вище можна сформулювати в наступному вигляді:

Властивість: На відміну від TCP, протокол ARTCP не чутливий до втрат сегментів.

Напрямки подальшого розвитку ARTCP

Протокол ARTCP, запропонований у цій роботі, здатний працювати більш ефективно і якісно, ніж TCP, однак можна виділити кілька напрямків подальших досліджень нового протоколу, які можуть, по-перше, дати можливість ефективно використовувати його в асиметричних системах, а по-друге, досягти рівноправності між потоками з різною довжиною маршруту.

Асиметричні системи

Оскільки в протоколі ARTCP усунута ACK-синхронізація, властива TCP, то відправлення сегментів відбувається незалежно від прибуття підтверджень аж до вичерпання максимального вікна, то на відміну від TCP, ARTCP може бути вдосконалений так, щоб ефективно працювати в системах з асиметричними каналами.

Для використання ARTCP у таких системах необхідно зменшити частоту підтверджень. Оскільки штучна затримка підтверджень викличе збільшення затримки в петлі зворотного зв'язку, то, вимір затримки передачі сегментів

потрібно також зв'язати з одержувачем. Оскільки важко домогтися гарної синхронізації системних годин одержувача й відправника, то одержувач може лише помічати зміну часу передачі сегментів, якщо відправник використовує стандартне поле часової мітки [6]. Якщо різниця значень мітки в потоці й системних годинах одержувача змінюється, значить змінюється й абсолютне значення затримки. У цьому випадку одержувач повинен збільшити частоту підтверджень, щоб відправник міг зреагувати на зміну навантаження в мережі. Коли значення швидкості прибуття потоку й затримки передачі не міняються, частота підтверджень може бути знову зменшена.

Алгоритм ARTCP не містить перешкод для цього вдосконалення, крім того, модифікований таким способом ARTCP для асиметричних каналів збереже сумісність із представленим тут протоколом.

З'єднання з різними RTT

Для з'єднань, що володіють різним часом RTT, через розходження довжин їхніх маршрутів, середнє значення коефіцієнта F обмежується деяким числом, меншим 1, як показано в роботах [80,81,82,83]. Це вірно як для ARTCP, так і для TCP. Щоб досягти рівноправності поділу ПрЗд між ARTCP потоками з різною довжиною маршруту, необхідно усунути залежність коефіцієнта *speedup* від мінімального часу RTT. Цього результату можна досягти шляхом модифікації алгоритму адаптації таким чином, щоб у режимі **FT** повністю відмовитися від використання обмірюваного RTT як індикатора перевантаження, використовуючи лише значення міжсегментних інтервалів потоку.

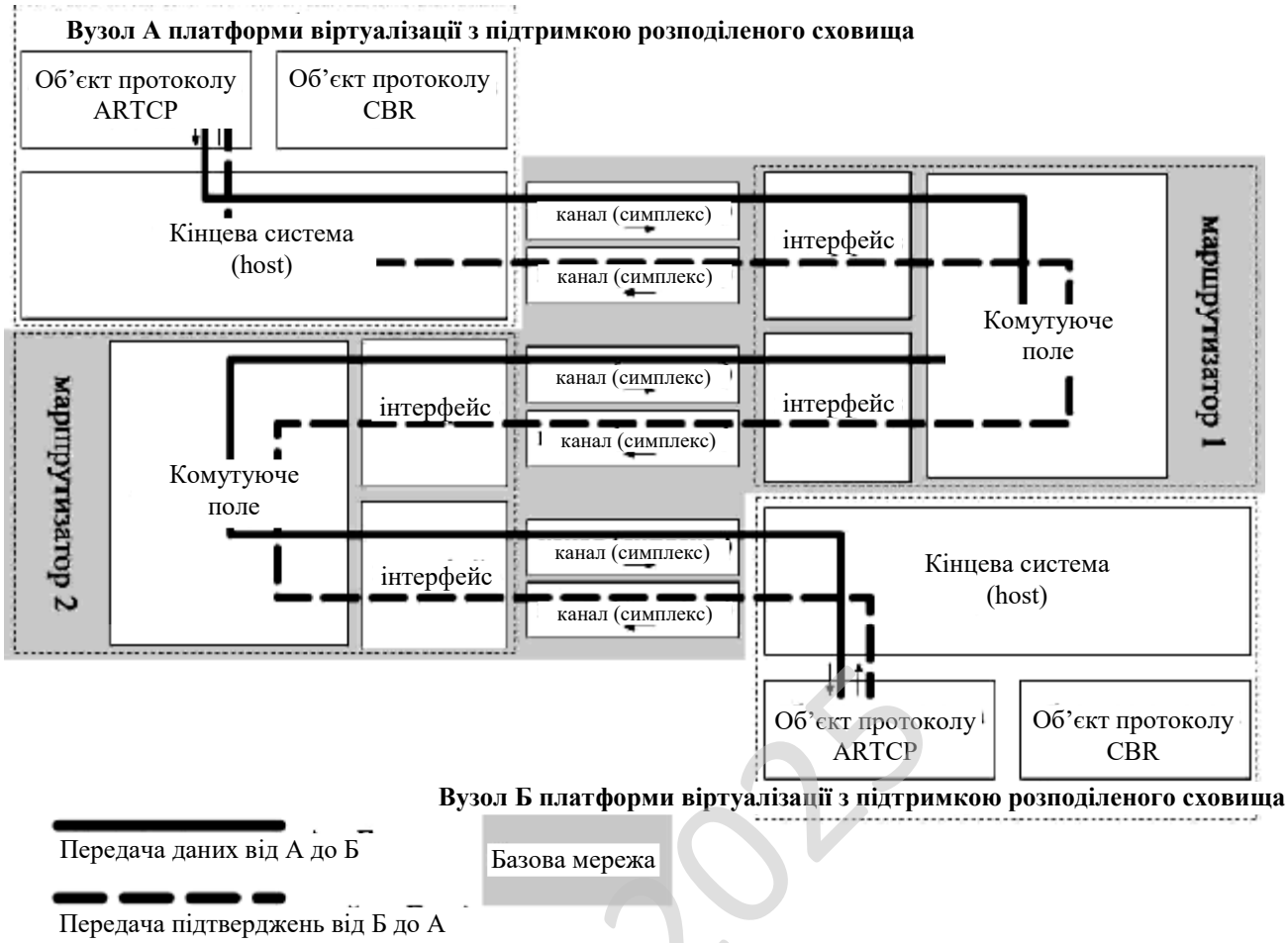

Рисунок 3.2 – Функціональна схема системи

Для реалізації протоколу платформи віртуалізації з підтримкою розподіленого сховища необхідно створити імітаційну модель. Для організації моделі платформи віртуалізації з підтримкою розподіленого сховища за найпростішою схемою потрібен набір з 14 об'єктів (рисунок 3.2). На рисунку 3.2 наведена функціональна схема розробленої системи, виходячи зі схеми зображеної на рисунку 3.1. З рисунка 3.2 бачимо, що для реалізації імітаційної моделі встановлюється два хоста та два маршрутизатора, через які йде трафік. Зафарбована область позначає топологічні елементи мережі (канали й маршрутизатори).

На функціональній схемі вказані усі елементи, які необхідні для передачі пакета по протоколу ARTCP у мережі. До них відносяться:

– об'єкти протоколу ARTCP та CBR;

– хости;

– симплексні канали передачі даних;

– інтерфейси по яким відбувається з’єднання між маршрутизаторами;

– маршрутизатори;

– комутуюче поле у маршрутизаторі (таблиця маршрутизації).

Розглянувши усі блоки функціональної схеми перейдемо до розгляду діаграми взаємодії процесів, які відбуваються у системі.

3.4 Розробка діаграми процесів

Діаграма процесів розробленої системи зображена на рисунку 3.3. При детальному її розгляді можна побачити як саме проходе взаємодія у розробленій системі. Використовується модель проектування, графічне представлення «потоків» даних в інформаційній системі.

Діаграма взаємодії процесів використовується для візуалізації процесів обробки даних (структурне проектування).

Для розробника вважається звичним спочатку креслити діаграму взаємодії процесів даних рівня контексту, завдяки чому буде показано взаємодію системи.

Ця діаграма в подальшому підлягає уточненню шляхом деталізації процесів та потоків даних з метою показати систему що розробляється.

Діаграми потоків даних містять чотири типи елементів:

– Процеси які являють собою трансформацію даних в рамках описуваної системи.

– Сховища даних (репозиторії).

– Зовнішні по відношенню до системи сутності.

– Потоки даних між елементами трьох попередніх типів.

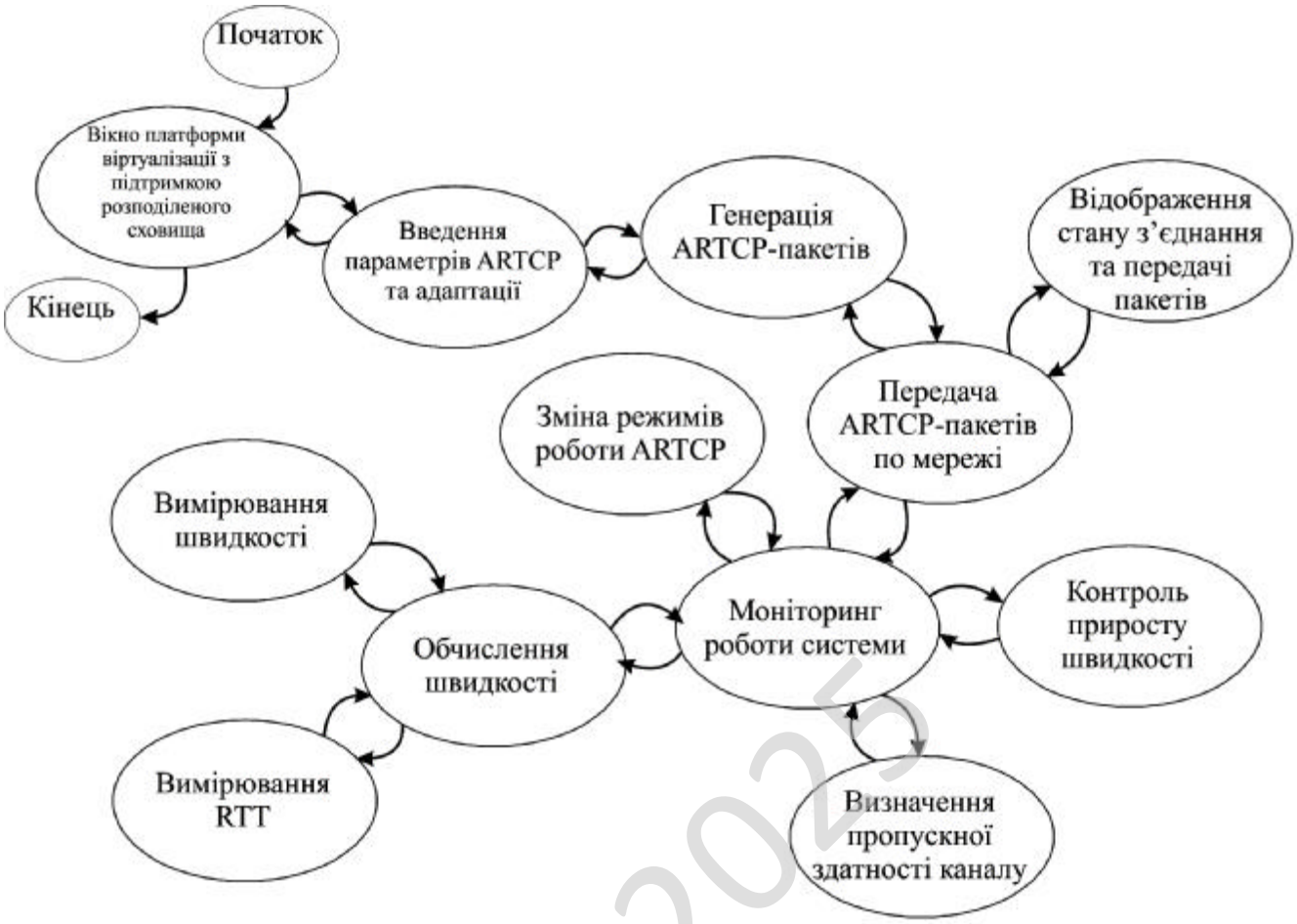

Рисунок 3.3 – Діаграма взаємодії процесів

Таким чином, розглянувши опис системи, структурну, функціональну схеми системи, та діаграму взаємодії процесів перейдемо до опису блок-схем основної програми, та підпрограм, які використовуються, для реалізації системи.

4 РЕАЛІЗАЦІЯ ПРОЕКТУ. РОЗРАХУНКИ І ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ПРАВИЛЬНІСТЬ ПРОЕКТНИХ РІШЕНЬ

4.1 Блок-схеми та опис алгоритмів функціонування системи

Під час роботи над бакалаврською дипломною роботою було створено блок-схеми. Перед їх розглядом необхідно провести роз’яснення який саме тип блок-схем використовується.

Блок-схема це представлення задачі для її аналізу або розв'язування за допомогою спеціальних символів (геометричних образів), які позначають такі елементи, як операції, потік, дані тощо. Блок вхідних та вихідних даних прийнято позначати паралелограмом, блок обчислень (обробки) даних – прямокутником, блок прийняття рішень – ромбом, еліпсом – початок та кінець алгоритму.

У інформаційних технологіях функціональна схема складається з функціональних блоків, які являють собою конструктивно відособлені частини (елементи або пристрої) автоматичних систем, які виконують певні функції. Функціональні блоки на схемі позначають прямокутниками, всередині яких надписують їх найменування відповідно до функцій, що виконуються. Зв'язки між функціональними блоками (внутрішні впливи) позначаються лініями зі стрілками, які вказують напрям впливів.

Функціональні схеми можуть виконуватися в укрупненому і розгорненому вигляді. У першому випадку на схемі зображають найважливіші блоки системи і зв'язки між ними.

У другому варіанті схема відображається більш детально, що полегшує її читання та ілюструє принцип роботи.

Основні елементи схем алгоритму це термінатор, процес, рішення, зумовлений процес (підпрограма), дані та з'єднувач.

Термінатор це елемент відображає вхід із зовнішнього середовища або вихід з неї (найчастіше застосування – початок і кінець програми). Всередині фігури записується відповідна дія.

Процес це виконання однієї або кількох операцій, обробка даних будь-якого виду (зміна значення даних, форми подання, розташування). Всередині фігури записують безпосередньо самі операції.

Рішення це показує рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення умов, визначених всередині цього елементу. Вхід в елемент позначається лінією, що входить зазвичай у верхню вершину елементу. Якщо виходів два чи три то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижній). Якщо виходів більше трьох, то їх слід показувати однією лінією, що виходить з вершини (частіше нижній) елемента, яка потім розгалужується. Відповідні результати обчислень можуть записуватися поруч з лініями, що відображають ці шляхи.

Зумовлений процес (підпрограма) це символ відображає виконання процесу, що складається з однієї або кількох операцій, що визначені в іншому місці програми (у підпрограмі, модулі). Всередині символу записується назва процесу і передані в нього дані.

Дані це перетворення у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Цей символ не визначає носія даних (для вказівки типу носія даних використовуються специфічні символи).

З'єднувач це символ відображає вихід в частину схеми і вхід з іншої частини цієї схеми. Використовується для обриву лінії та продовження її в іншому місці (приклад: поділ блок-схеми, що не поміщається на листі). Відповідні сполучні символи повинні мати одне (при тому унікальне) позначення.

Блок-схеми є першоджерелами стратегії розвитку ПЗ. Тому від точності і детальної блок-схеми залежить результат всієї програми.

При виборі початкової точки відліку при побудові схем було враховано, що виходячи з вибору мови програмування і інших технічних засобів, програма буде об’єктно-орієнтована що вимагає оптимізації програми високого рівня, також те, що при розробці програми слід надати особливу увагу модулю платформи віртуалізації з підтримкою розподіленого сховища.

На рисунку 4.1 зображена основна блок-схема програми, на рисунку 4.2 зображено роботу підпрограми.

З яких видно що робота основної програми складається з початкових етапів ініціалізації ПЗ, перевірки наявності ресурсів системи, блоку початку основного циклу з чеканням запиту від користувача в якому відбувається виклик підпрограми та останньої стадії – перевірка поточного стану з завершенням роботи розробленого ПЗ. При роботі підпрограми виконується основний функціонал системи з циклічними послідовностями, перевіркою поточного стану та поверненням в основну програму прапорів стану виконання.

Було використано підходи з використанням UML, це уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. UML є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, називаної UML-моделлю. UML був створений для визначення, візуалізації, проектування й документування в основному програмних систем. UML не є мовою програмування, але в засобах виконання UML-моделей як інтерпретованого коду можлива кодогенерація.

UML може бути застосовано на всіх етапах життєвого циклу аналізу бізнес-систем і розробки прикладних програм. Різні види діаграм які підтримуються UML, і найбагатший набір можливостей представлення певних аспектів системи робить UML універсальним засобом опису як програмних, так і ділових систем.

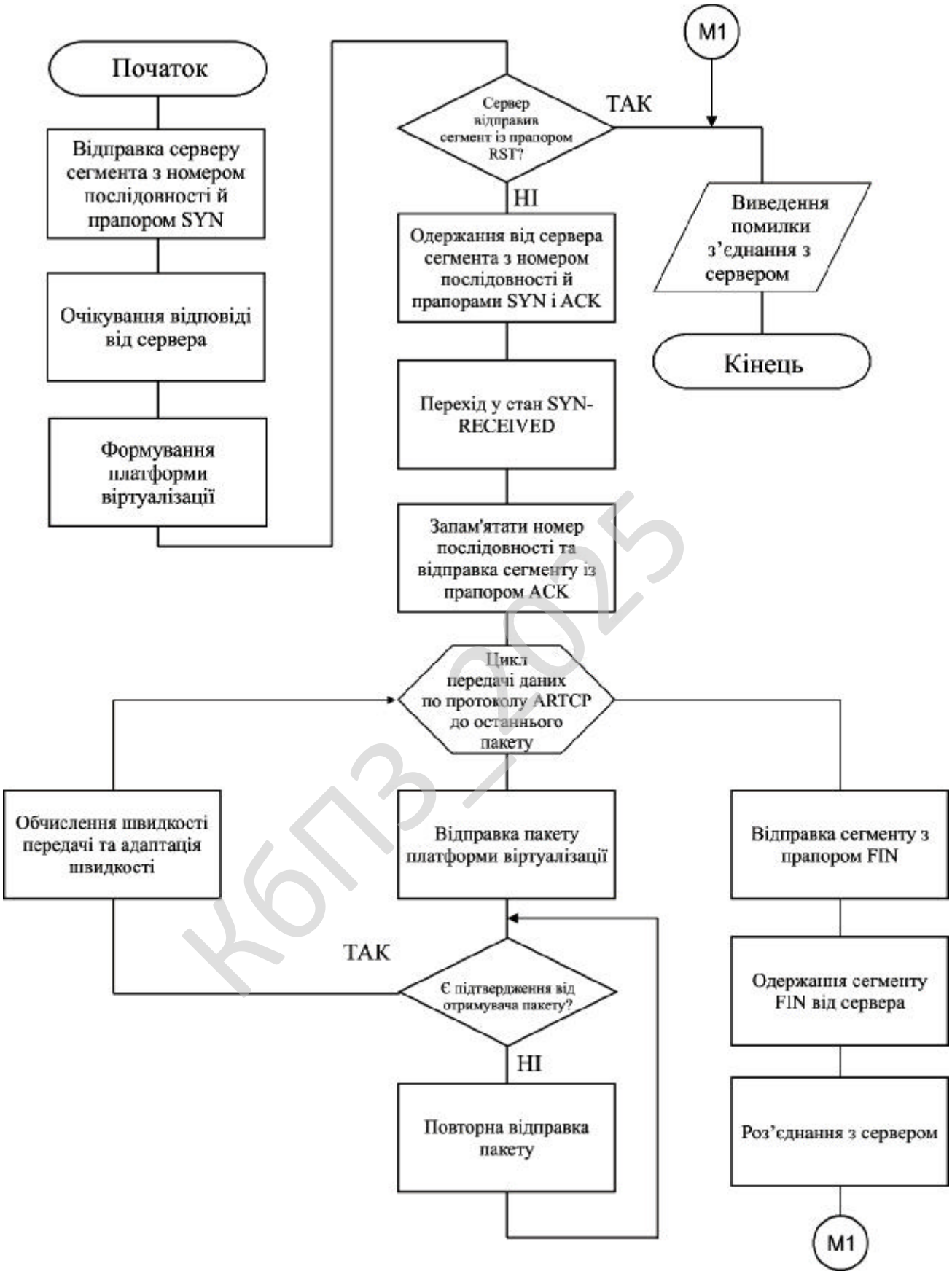

Рисунок 4.1 – Блок-схема основної програми

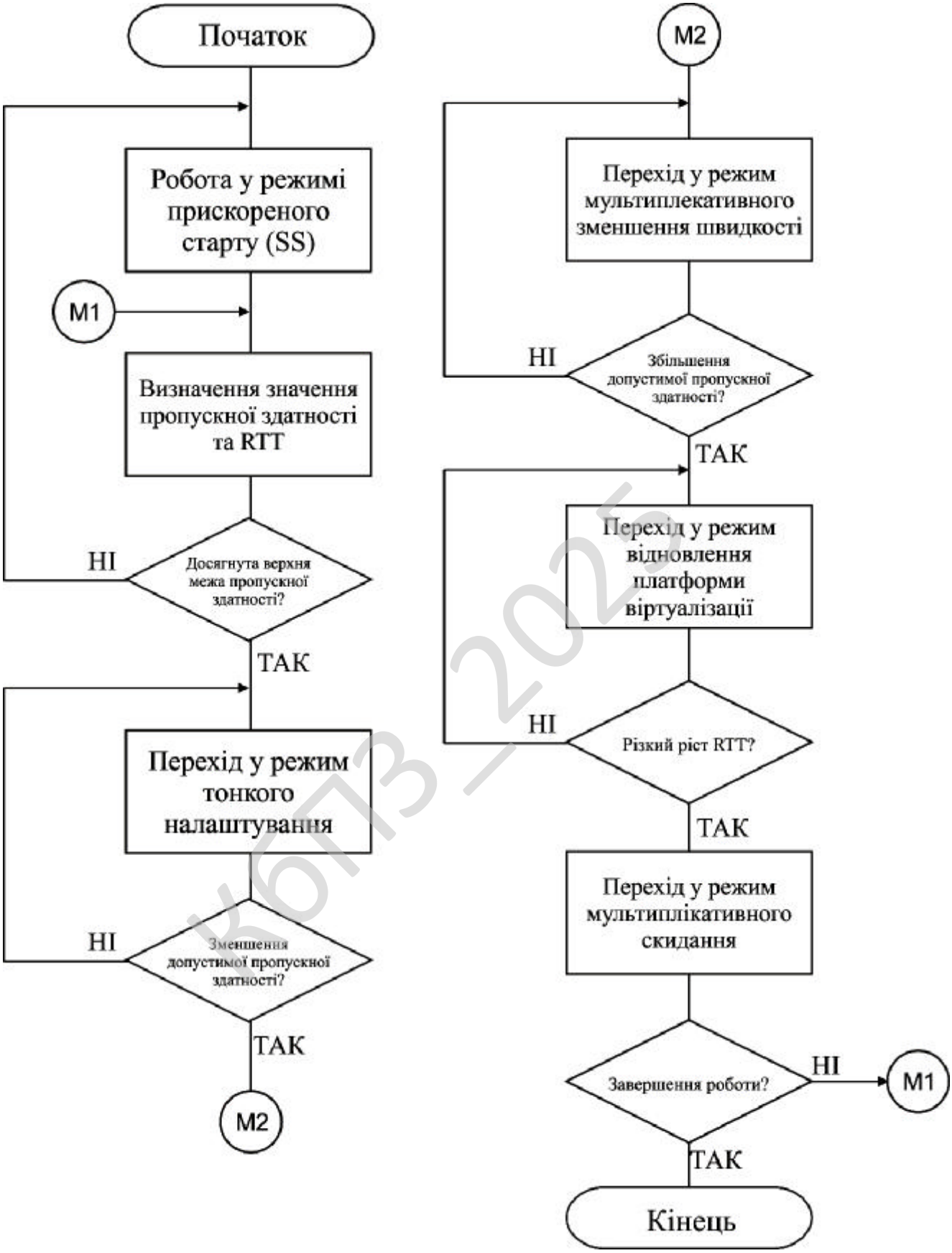

Рисунок 4.2 – Блок-схема роботи підпрограми

Діаграми дають можливість представити систему (як ділову, так і програмну) у такому вигляді, щоб її можна було легко перевести в програмний код. Основною причиною використання мови UML є спілкування розробників між собою.

Крім того, UML спеціально створювалася для оптимізації процесу розробки програмних систем, що дозволяє збільшити ефективність їх реалізації у кілька разів і помітно поліпшити якість кінцевого продукту.

UML прекрасно зарекомендувала себе в багатьох успішних програмних проектах. Засоби автоматичної генерації кодів дозволяють перетворювати моделі мовою UML у вихідний код об’єктно-орієнтованих мов програмування, що ще більш прискорює процес розробки. Практично усі CASE-засоби (програми автоматизації процесу аналізу і проектування) мають підтримку UML. Моделі розроблені в UML, дозволяють значно спростити процес кодування і направити зусилля програмістів безпосередньо на реалізацію системи.

Діаграми підвищують супроводжуваність проекту і полегшують розробку документації.

UML необхідний:

– Керівникам проектів, які керують розподілом завдань і контролем за проектом.

– Проектувальникам інформаційних систем які розробляють технічні завдання для програмістів.

– Бізнес-аналітикам, які досліджують реальну систему і здійснюють інжиніринг і реінжиніринг бізнесу компанії.

– Програмістам які реалізовують модулі інформаційної системи.

При модифікації системи об'єктний підхід дозволяє легко включати в систему нові об'єкти і виключати застарілі без істотної зміни її життєздатності. Використання побудованої моделі при модифікаціях системи дає можливість усунути небажані наслідки змін, оскільки вони не ламають структури системи, а тільки змінюють поведінку об'єктів.

Архітектура клієнт-сервер є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних програм і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти:

– набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;

– набір клієнтів, які використовують сервіси, що надаються серверами;

– мережа, яка забезпечує взаємодію між клієнтами та серверами.

Сервери є незалежними один від одного. Клієнти також функціонують паралельно і незалежно один від одного. Немає жорсткої прив'язки клієнтів до серверів. Більш ніж типовою є ситуація, коли один сервер одночасно обробляє запити від різних клієнтів; з іншого боку, клієнт може звертатися то до одного сервера, то до іншого. Клієнти мають знати про доступні сервери, але можуть не мати жодного уявлення про існування інших клієнтів.

Дуже важливо ясно уявляти, хто або що розглядається як «клієнт». Можна говорити про клієнтський комп'ютер, з якого відбувається звернення до інших комп'ютерів. Можна говорити про клієнтське та серверне програмне забезпечення. Нарешті, можна говорити про людей, які бажають за допомогою відповідного програмного та апаратного забезпечення отримати доступ до тієї чи іншої інформації.

Загальноприйнятим є положення, що клієнти та сервери – це перш за все програмні модулі. Найчастіше вони знаходяться на різних комп'ютерах, але бувають ситуації, коли обидві програми – і клієнтська, і серверна, фізично розміщуються на одній машині; в такій ситуації сервер часто називається локальним.

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером. Логічно можна відокремити три рівні операцій:

– рівень представлення даних, який по суті являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;

– прикладний рівень, який реалізує основну логіку ПЗ і на якому здійснюється необхідна обробка інформації;

– рівень управління даними, який забезпечує зберігання даних та доступ до них.

Дворівнева клієнт-серверна архітектура передбачає взаємодію двох програмних модулів – клієнтського та серверного. В залежності від того, як між ними розподіляються наведені вище функції, розрізняють:

– модель тонкого клієнта, в рамках якої вся логіка ПЗ та управління даними зосереджена на сервері. Клієнтська програма забезпечує тільки функції рівня представлення;

– модель товстого клієнта, в якій сервер тільки керує даними, а обробка інформації та інтерфейс користувача зосереджені на стороні клієнта. Товстими клієнтами часто також називають пристрої з обмеженою потужністю: кишенькові комп'ютери, мобільні телефони та ін.

Типовим прикладом клієнт-серверної взаємодії є WWW. Існує величезна кількість веб-серверів, на яких розміщується та чи інша інформація. У найпростішому випадку ця інформація являє собою набір веб-сторінок, які можуть зберігатися на сервері у вигляді файлів, розмічених за допомогою мови розмітки HTML. Але ситуація, як правило, є складнішою; значна частина веб-ресурсів на сучасному етапі є динамічними, тобто вони не існують в заздалегідь підготовленому вигляді, а створюються безпосередньо в процесі обробки запиту від користувача.

Для того, щоб людина, яка працює в Інтернеті, могла переглянути ту чи іншу сторінку, на її комп'ютері повинно бути встановлено відповідне програмне забезпечення. Програми для перегляду веб-сторінок називаються браузерами.

Але, крім браузерів, до серверів можуть звертатися і інші клієнти, а саме – автономні програми. Вони можуть передбачати взаємодію з людиною, а можуть працювати в цілком автоматичному режимі. Типовим класом таких програм є роботи, призначені для автоматичного перегляду веб-ресурсів. Зокрема, роботи є важливим елементом пошукових систем і використовуються ними для перегляду сторінок і збору інформації про них.

Для запиту до веб-сервера клієнтська програма повинна задати місцезнаходження комп'ютера, на якому розміщується серверна програма, назву потрібного документа і, можливо, інші дані, які специфікують запит. Мережа забезпечує знаходження сервера і передачу йому клієнтського запиту. Серверні програми обробляють цей запит, відповідь пересилається по мережі клієнтові.

Трирівнева клієнт-серверна архітектура, яка почала розвиватися з середини 90-х років, передбачає відділення прикладного рівня від управління даними. Відокремлюється окремий програмний рівень, на якому зосереджується прикладна логіка ПЗ. Програми проміжного рівня можуть функціонувати під управлінням спеціальних серверів ПЗ, але запуск таких програм може здійснюватися і під управлінням звичайного веб-сервера. Нарешті, управління даними здійснюється сервером даних.

Для роботи з системою користувач використовує стандартне програмне забезпечення –звичайний браузер. Це позбавляє його необхідності завантажувати та інсталювати спеціальні програми (хоча інколи така необхідність все-таки виникає).

Але користувачеві слід надати в розпорядженні інтерфейс, який дозволяв би йому взаємодіяти з системою і формувати запити до неї. Форми, що визначають цей інтерфейс, розміщуються на веб-сторінках та завантажуються разом з ними.

Веб-оглядач формує запит та пересилає його до сервера, який здійснює обробку. При необхідності сервер викликає серверні програмні модулі, які забезпечують обробку запиту і в разі потреби звертаються до сервера даних.

Сервер даних здійснює операції з даними, що зберігаються в системі та складають її інформаційну основу. Зокрема, він може здійснити вибірку з інформаційної бази відповідно до запиту та передати її модулю проміжного рівня для подальшої обробки. Дані, з якими працює сервер даних, найчастіше організовані як реляційна база даних.

Найчастіше веб-сервер і серверні модулі проміжного рівня розміщуються на одному комп'ютері, хоч і являють собою окремі і логічно незалежні програмні модулі.

На сучасному етапі для програмування модулів проміжного рівня використовується мова серверних сценаріїв PHP, а для управління даними – СУБД MySQL. Таким чином, зв'язку PHP-MySQL слід розглядати як стандартний інструмент для створення порівняно простих інтерактивних веб-сайтів та систем електронної комерції; близько 90% комерційних систем сьогодні створюється саме на цій основі.

Водночас як засоби управління даними, так і middleware-засоби можуть бути найрізноманітнішими. Так, для створення серверних програм, крім PHP, широко застосовуються Java, Perl, Python, Delphi.

Взагалі, технології створення розподілених, зокрема веб-програм, стрімко розвиваються. Слід згадати про технології EJB (Enterprise Java Beans), CORBA, а також про .NET – порівняно нову ініціативу компанії Microsoft. Для зберігання даних та їх передачі часто використовується так звана розширювана мова розмітки XML (Extensible Markup Language).

Redmine – вільне серверне ПЗ для управління проектами та відстежування помилок. До системи входить календар-планувальник та діаграми Ганта для візуального представлення ходу робіт за проектом та строків виконання.

Redmine написано на мові Ruby і є ПЗ розробленим з використанням відомого веб-фреймворку Ruby on Rails, що означає легкість в розгортанні системи та її адаптації під конкретні вимоги.

Для кожного проекту можна вести свої вікі та форуми.

Функціональні можливості:

– Ведення декількох проектів.

– Гнучка система доступу з використанням ролей.

– Система відстеження помилок.

– Діаграми Ганта та календар.

– Ведення новин проекту, документів та управління файлами.

– Сповіщення про зміни за допомогою RSS-потоків та електронної пошти.

– Власна Wiki для кожного проекту.

– Форуми для кожного проекту.

– Облік часових витрат.

– Налаштування власних (custom) полів для задач, затрат часу, проектів та користувачів.

– Легка інтеграція із системами керування версіями (SVN, CVS, Git, Mercurial, Bazaar и Darcs).

– Створення записів про помилки на основі отриманих листів

– Підтримка LDAP автентифікації.

– Можливість самореєстрації нових користувачів.

– Багатомовний інтерфейс (у тому числі українська мова).

– Підтримка СКБД: MySQL, PostgreSQL, SQLite.

Діаграма Ганта (Gantt chart, також стрічкова діаграма, графік Ганта) – це популярний тип діаграм, який використовується для ілюстрації плану, графіка робіт за будь-яким проектом. Є одним з методів планування та управління проектами.

Діаграма Ганта являє собою відрізки (графічні плашки), розміщені на горизонтальній шкалі часу. Кожен відрізок відповідає окремому завданню або підзадачі. Завдання і підзадачі, складові плану, розміщуються по вертикалі. Початок, кінець і довжина відрізка на шкалі часу відповідають початку, кінцю і

тривалості завдання. На деяких діаграмах Ганта також показується залежність між завданнями.

Діаграма може використовуватися для представлення поточного стану виконання робіт: частина прямокутника, що відповідає завданню, заштриховується, відзначаючи відсоток виконання завдання; показується вертикальна лінія, що відповідає моменту «сьогодні».

Часто діаграма Ганта використовується спільно з таблицею зі списком робіт, рядки якої відповідають окремо взятій задачі, зображеній на діаграмі, а стовпці містять додаткову інформацію про задачу.

Система відстеження помилок Багтрекер – прикладна програма для допомоги розробникам програмного забезпечення (програмістам, тестувальникам тощо) враховувати і контролювати помилки, знайдені у програмах, питання щодо функціональності, рішення та оновлення, побажання користувачів, а також стежити за процесом їх виконання.

Кожному, хто розробляв програмні продукти, добре знайоме співвідношення «20/80» – останні 20 % роботи тривають 80 % часу.

Як це не парадоксально, але нічого дивного в цій пропорції немає, адже саме на завершальній стадії починається тестування проекту, коли виявляються помилки, і що більший проект, то більше буде знайдено помилок.

Водночас досить часто виявляється, що більшість цих помилок були відомі та могли бути виправлені з меншими витратами на попередніх стадіях роботи, але не були вчасно описані, а потім загубилися серед інших важливих завдань.

Отже, система відстеження помилок у найпростішому варіанті – це процес, що включає в себе виявлення помилки, її опис, виправлення і перевірку цього виправлення, тобто процес «стеження» за багом протягом всього як його життєвого циклу, так і життєвого циклу розробки в цілому.

Сукупність інформації про дефект. Головний компонент такої системи – база даних, що містить відомості про виявлені дефекти. Ці відомості можуть включати в себе:

– номер (ідентифікатор) дефекту;

– хто повідомив про дефект;

– дата і час виявлення дефекту;

– версія продукту, в якій виявлено дефект;

– серйозність (критичність) дефекту та пріоритет рішення;

– опис кроків для відтворення дефекту (неправильної поведінки програми);

– відповідальний за усунення дефекту;

– обговорення можливих рішень та їх наслідків;

– поточний стан виправлення дефекту;

– версії продукту, в якій дефект виправлений.

Крім того, розвинені системи надають можливість прикріплювати файли, які допомагають описати проблему, наприклад, дамп пам'яті або скріншот.

Використання. Основна перевага систем відстеження помилок полягає в забезпеченні чітких централізованих оглядів, запитів на розробку (включаючи помилки і виправлення) та їх стан. У корпоративному середовищі, системи відстеження помилок можуть бути використані для генерації звітів по продуктивності програмістів виправлення помилок. Однак, це може іноді приводити до неточних результатів, тому що різні помилки можуть мати різні ступені пріоритету та серйозності, що пов'язано з складністю їх фіксації.

Життєвий цикл дефекту. Як правило, система відстеження помилок використовує той чи інший варіант «життєвого циклу» помилки, стадія якого визначається поточним станом помилки.

Типовий життєвий цикл дефекту:

1. Новий – дефект зареєстрований тестувальником.

2. Призначений – призначений відповідальний за виправлення дефекту.

3. Дозволений – дефект переходить назад у сферу відповідальності тестувальника. Як правило, супроводжується резолюцією, наприклад:

– Виправлено (виправлення включені у версію таку-то).

– Дубль (повторює дефект, що вже знаходиться в роботі).

– Не виправлено (працює відповідно до специфікації, має занадто низький пріоритет, виправлення відкладено до наступної версії тощо).

– «В мене все працює» (запит додаткової інформації про умови, в яких дефект проявляється).

4. Далі тестувальник проводить перевірку виправлення, залежно від чого дефект або знову переходить у стан «Призначений» (якщо він описаний як виправлений, але не виправлений), або у стан «Закрито».

5. Відкрито повторно – дефект знайдено знову в іншій версії.

Система може надавати адміністраторові можливість налаштування користувачі, які можуть переглядати і редагувати помилки залежно від їх стану, переводити їх в інший стан або видаляти.

У корпоративному середовищі, система відстеження помилок може використовуватися для отримання звітів, що показують продуктивність програмістів при виправленні помилок. Однак, часто такий підхід не дає достатньо точних результатів через те, що різні помилки мають різну ступінь серйозності та складності. При цьому серйозність проблеми прямо не стосується складності її усунення.

4.2 Захист розробленого програмного забезпечення

Для захисту розробленого програмного забезпечення запропоновано використовувати алгоритм SEED – у криптографії симетричний блоковий криптоалгоритм на основі Мережі Фейстеля, розроблений Корейським агентством інформаційної безпеки (Korean Information Security Agency, KISA) в 1998 році. В алгоритмі використовується 128-бітний блок і ключ довжиною 128 біт. Алгоритм одержав широке поширення й використовується фінансовими й банківськими структурами, виробничими підприємствами й бюджетними установами Південної Кореї, оскільки 40-бітний SSL не забезпечує на даний момент мінімально необхідного рівня безпеки. Агентством по захисту інформації

специфіковане використання шифру SEED у протоколах TLS і S/MIME. У той же час, алгоритм SEED не реалізований у більшості сучасних браузерів і інтернет-додатків, що утрудняє його використання в даній сфері поза межами Південної Кореї.

SEED являє собою мережу Фейстеля з 16 раундами, 128-бітовими блоками й 128-бітовим ключем. Алгоритм використовує дві 8×8 таблиці підстановки, які, як такі з Safer, виведені з дискретного зведення в ступінь (у цьому випадку, x^{247} і x^{251} – плюс деякі «несумісні операції»). Це є деякою подібністю с MISTY1 у рекурсивності його структури: 128-бітовий повний шифр – мережа Фейстеля з F-функцією, що впливає на 64-бітові половини, у той час як сама F-функція – Мережа Фейстеля, складена з G-функції, що впливає на 32-розрядні половини. Однак рекурсія не простягнеться далі, тому що G-функція – не Мережа Фейстеля. В G-функції 32-розрядне слово розглядають як чотири 8-бітових байта, кожний з яких проходить через одну або іншу таблицю підстановки, потім поєднується в помірковано комплексному наборі булевих функцій таким чином, що кожний біт виводу залежить від 3 з 4 вхідних байтів.

SEED має складний ключовий розклад, генеруючи тридцять два 32-розрядних додаткових символу, використовуючи G-функції на серіях обертань вихідного неопрацьованого ключа, комбінованого зі спеціальними раундовими константами (як в TEA) від «Золотого співвідношення» (англ. Golden ratio).

Згідно з дослідженнями KISA, алгоритм SEED «надійно протистоїть відомим атакам».

5 МЕТОДИКА ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ

Розглянемо розроблене ПЗ платформи віртуалізації з підтримкою розподіленого сховища яке зображено на рисунку 5.1. З рисунку можна побачити що інтерфейс головного вікна розподілено на наступні функціональні розділи:

– Навігаційне меню: Файл; Генерація пакету; Параметри; Довідка.

– Функції роботи з пакетними даними.

– Розділу параметрів системи.

– Розділу введення та виведення результату роботи системи.

– Навігаційного меню яке викликається натисканням правої клавіші маніпулятора миші.

– Функціональних кнопок ПЗ.

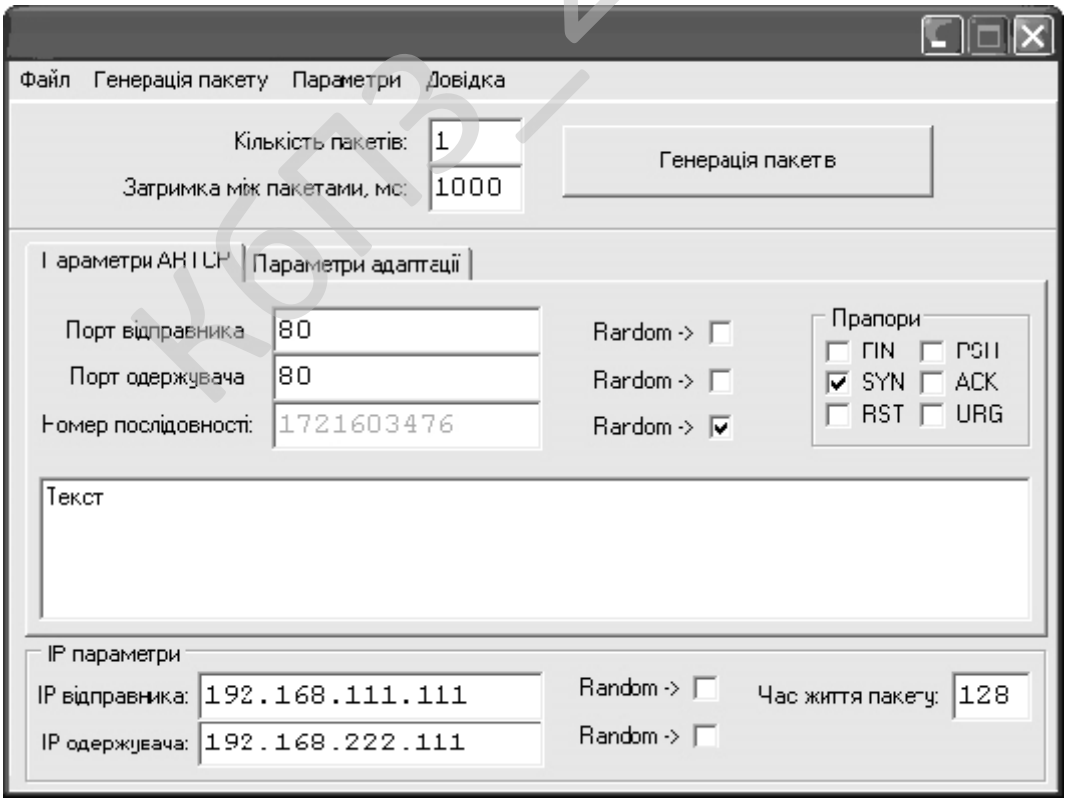

Рисунок 5.1 – Головне вікно ПЗ

Розроблена програма має дуже простий і інтуїтивно зрозумілий інтерфейс з користувачем. Кожен, хто в достатньому обсязі володіє операційним середовищем Windows без особливих складностей освоїть і цю програму, оскільки її інтерфейс інтуїтивно зрозумілий.

Якщо програма не видала ніяких помилок, і працює, то можна використовувати, інакше слід слідувати інструкціям, які пропонує програма.

На рисунку 5.2 зображено авторські дані розробленого програмного забезпечення.

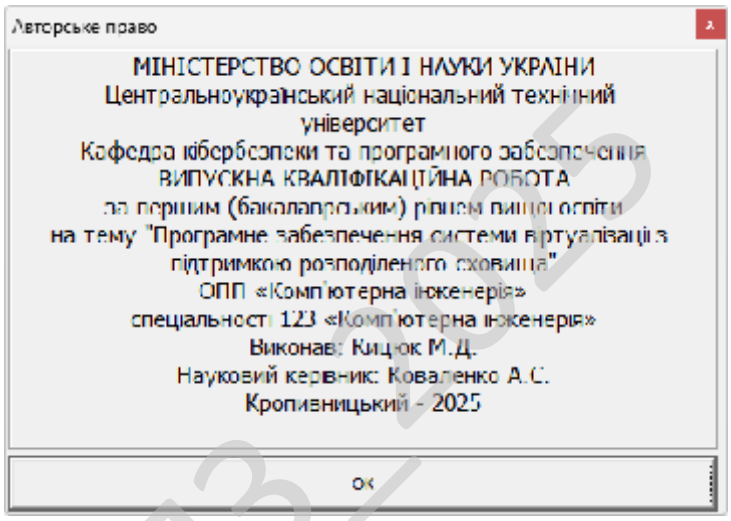

Рисунок 5.2 – Авторське право

Розглянемо процес впровадження програмного забезпечення, це процес налаштування програмного забезпечення під певні умови використання, а також навчання користувачів роботі з програмним продуктом. Впровадження програмного забезпечення це усі дії, що роблять розроблену програмну систему готовою до використання. Даний процес є частинною життєвого циклу програмного забезпечення.

Загалом процес розгортання складається з кількох взаємопов'язаних дій із можливими переходами між ними. Ця активність може відбуватися як з боку виробника так і з боку споживача.

Оскільки кожна програмна система є унікальною, то усі процеси та процедури під час розгортання важко передбачити. Тому, "розгортання" можна трактувати як загальний процес відповідно до певних вимог та характеристик. Розгортання може здійснюватись програмістом і в процесі розробки програмного забезпечення.

До діяльностей пов'язаних із розгортанням програмного забезпечення відносять:

– Випуск.

– Встановлення та активація.

– Деактивація.

– Адаптація.

– Обновлення.

– Вмонтування.

– Відстежування версій.

– Видалення.

– Вилучення з обігу.

При впровадженні програмного забезпечення потрібно урахувати наступні дії:

– Виділення критичних, з точки зору загального результату, процедур в діяльності організації. Коли набір таких процедур визначений, необхідно в першу чергу використовувати ІТ рішення для автоматизації операцій усередині саме цих процедур. Таким чином, розроблене ІТ рішення автоматично стає життєво важливим і затребуваним для організації, а також буде забезпечена публічність процесу впровадження;

– Розширення нормативної бази організації шляхом включення до неї регламентів, що описують порядок виконання процедур автоматизованих процесів. В іншому випадку є небезпека виникнення неузгодженості між автоматизованими процедурами та іншими процесами організації.

– Виконання робіт з загальної стандартизації існуючої діяльності організації, коли виділяються кращі практики виконання процедур і включаються в IT рішення за принципом найбільшої корисності для більшості учасників. Відсоток таких процедур щодо загального обсягу автоматизації може бути невеликий, але це надає процесу побудови рішення вагу в організації за рахунок збільшення його необхідності.

Під час роботи над програмою було проведено тестування програмного забезпечення, тобто технічне дослідження, призначене для виявлення інформації про якість продукту відносно контексту, в якому воно має використовуватись.

Тестування включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою їх оцінки.

Проводилась оцінка:

– відповідності поставленим вимогам;

– правильна відповідь для усіх можливих вхідних даних;

– виконання функцій за прийнятний час;

– практичність;

– сумісність з ОС та стороннім ПЗ.

Оскільки число можливих тестів для програмних компонент практично нескінченне, тому стратегія тестування полягала в тому, щоб провести всі можливі тести з урахуванням наявного часу та ресурсів.

Як результат ПЗ тестувалось стандартним виконанням програми з метою виявлення помилок або інших дефектів.

Проводилось тестування форматом білої скриньки засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління.

У цьому випадку формуються тестові варіанти, в яких:

– Гарантується перевірка всіх незалежних маршрутів програми.

– Знаходяться гілки True, False для всіх логічних рішень.

– Виконуються всі цикли (у межах їхніх кордонів та діапазонів).

– Аналізується правильність внутрішніх структур даних.

Недоліки тестування "білої скриньки":

– Кількість незалежних маршрутів може бути дуже велика.

– Повне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.

– У програмі можуть бути пропущені деякі маршрути.

– Не можна виявити помилки, поява яких залежить від даних.

Переваги тестування "білої скриньки" пов'язані з тим, що принцип «білої скриньки» дозволяє врахувати особливості програмних помилок:

– Кількість помилок мінімально в «центрі» і максимально на «периферії» програми.

– Попередні припущення про ймовірність потоку керування або даних у програмі часто бувають некоректними. У результаті типовим може стати маршрут, модель обчислень за яким опрацьована слабо.

– При записі алгоритму програмного забезпечення у вигляді тексту на мові програмування можливе внесення типових помилок трансляції (синтаксичних та семантичних).

– Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Проводилось тестування чорної скриньки.

Основне місце програми тестів «чорної скриньки» – інтерфейс ПЗ. Відомі: функції програми. Досліджується: робота кожної функції на всій області визначення.

Ці тести демонструють:

– Як виконуються функції програми.

– Як приймаються вихідні дані.

– Як виробляються результати.

– Як зберігається цілісність зовнішньої інформації.

При тестуванні «чорної скриньки» розглядаються системні характеристики програм, ігнорується їхня внутрішня логічна структура. Вичерпне тестування, як правило, неможливе.

Наприклад, якщо в програмі 10 вхідних величин і кожна приймає по 10 значень, то кількість тестових варіантів становитиме 10^10. Тестування «чорної скриньки» не реагує на багато особливостей програмних помилок.

Тестування «чорної скриньки» (функціональне тестування) дозволяє отримати комбінації вхідних даних, які забезпечують повну перевірку всіх функціональних вимог до програми.

Програмний виріб тут розглядається як «чорна скринька», чию поведінку можна визначити тільки дослідженням його входів та відповідних виходів. При такому підході бажано мати:

– Набір, утворений такими вхідними даними, які призводять до аномалій у поведінці програми (назвемо його ITc).

– Набір, утворений такими вхідними даними, які демонструють дефекти програми (назвемо його OT).

Будь-який спосіб тестування «чорної скриньки» повинен:

– Виявити такі вхідні дані, які з високою ймовірністю належать набору ITc;

– Сформулювати такі очікувані результати, які з високою імовірністю є елементами набору OT.

Принцип «чорної скриньки» не альтернативний принципу «білої скриньки». Скоріше це доповнює підхід, який виявляє інший клас помилок.

Тестування «чорної скриньки» забезпечує пошук наступних категорій помилок:

– Некоректних чи відсутніх функцій.

– Помилок інтерфейсу.

– Помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних.

– Помилок характеристик (необхідна ємність пам'яті і т.д.).

– Помилок ініціалізації та завершення.

Обрано умови розповсюдження – Shareware.

Під умовно-безплатним програмним забезпеченням можна розуміти спосіб або метод розповсюдження комерційного ПЗ на ринку (тобто на шляху до кінцевого користувача), при якому випробувачеві пропонується обмежена за можливостями (не повнофункціональна або демонстраційна версія), терміном дії (тріал версія) або версія з вбудованим набридливим нагадуванням про необхідність оплати використання програми.

В угоді про використання (ліцензії для кінцевого користувача, EULA) також може бути обумовлена заборона на комерційне або професійне (не тестове) її використання.

Основний принцип умовно-безплатного ПЗ – «спробуй, перш ніж купити» (try before you buy). ПЗ що поширюється як умовно-безплатний, надається користувачам безоплатно. Звичайно користувач платить тільки за час завантаження файлів через Інтернет або за носій (CD диск, флешку, ключ). Протягом певного терміну, що становить зазвичай тридцять днів, він може користуватися програмою, тестувати її, освоювати її можливості.

Якщо після закінчення цього терміну користувач вирішить продовжити використання ПЗ, він зобов'язаний купити його (зареєструватися), заплативши авторові певну суму.

В іншому випадку користувач повинен припинити використання ПЗ та видалити його зі свого комп'ютера.

Обрано умови розповсюдження – commercial software. Програмне забезпечення, створене комерційною організацією з метою отримання прибутку від його використання іншими, наприклад, шляхом продажу копій.

Найважливішою особливістю комерційний програмних продуктів є підтримка великих компаній, прямо зацікавлених у поширенні програм. Багато організацій надають виключно платну підтримку своїх продуктів, такий підхід, як правило, використовують організації надають відкриті вихідні коди. Для

продуктів, що розповсюджуються на комерційній основі діють зазвичай безкоштовні служби підтримки, покликані збільшити рівень довіри у клієнтів і потенційних покупців.

Далеко не завжди, але як правило терміни критично важливих змін в комерційних продуктах значно менше, ніж у некомерційних проектів. Це пов'язано з тим, що над комерційним продуктом працюють цілі групи розробників і ця робота є їх основним заняттям. Розробникам-початківцям як правило доводиться шукати додаткові способи заробітку, і це збільшує час, що витрачається на доповнення і зміни програм. Так як основним рушійним фактором створення комерційного ПЗ є одержання прибутку, то комерційні програмні продукти першими заповнюють вільні ніші та пропонують варіанти вирішення завдань відразу по мірі виявлення вакууму в будь-якому секторі ринку.

Окремий вид комерційних програм, коли їх розробка оплачується безпосередньо замовником. Такі програми найчастіше позбавлені всіх переваг комерційних продуктів, оскільки мають обмежений бюджет, але більш адаптовані до вимог замовника, ніж аналоги.

6 ОСНОВНІ ВИСНОВКИ

Програмне забезпечення, створене в результаті виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти, призначено для системи віртуалізації з підтримкою розподіленого сховища.

В межах України в недостатній мірі представлені вітчизняні розробки в цій області.

Рішення завдання полягало у вирішенні наступних задач:

– Був проведений огляд існуючих систем віртуалізації з підтримкою розподіленого сховища.

– Досліджена система віртуалізації з підтримкою розподіленого сховища.

– На основі отриманих результатів досліджень створена програмна реалізація системи віртуалізації з підтримкою розподіленого сховища.

Розроблені під час виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти алгоритми дозволяють успішно вирішувати завдання віртуалізації з підтримкою розподіленого сховища.

Розроблене програмне забезпечення має простий, дружній та зручний інтерфейс користувача, що забезпечує легкість у освоєнні роботи програмного продукту, зручність у використанні, і не потребує особливих спеціальних знань.

При створені програмного забезпечення було використано об’єктно-орієнтований підхід, що відповідає сучасним тенденціям у галузі розробки комерційних програмних систем.

Програма реалізована на мові високого рівня Python. Дана мова програмування дозволяє найбільш ефективно обробляти дані призначені для системи віртуалізації з підтримкою розподіленого сховища. Це дозволило мінімізувати строк розробки програмного забезпечення, і, як слід, зменшити витрати на його розробку. Запропоноване програмне забезпечення ділиться на загальне програмне забезпечення, що поставляється із засобами обчислювальної

техніки й спеціальне програмне забезпечення, що спеціально розроблене для даної конкретної системи й включає програми, що реалізують її функції.

Програма призначена для виконання під управлінням багатозадачної операційної системи Windows 10/11.

Даються необхідні рекомендації з установки розробленого програмного забезпечення.

Для підвищення рівня безпеки запропоновано застосовувати алгоритм SEED.

В цілому створене програмне забезпечення підтверджує правильність використаних проектних рішень та повністю відповідає вимогам технічного завдання. Створене програмне забезпечення має потенційну можливість для подальшого вдосконалення і застосування у різних галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wendell Odom. «CCNA 200-301 Official Cert Guide, Volume 1». Cisco Press. 2020. – 848 p.

2. Wendell Odom. «CCNA 200-301 Official Cert Guide, Volume 2 Premium Edition eBook and Practice Test». Cisco Press. 2020. – 624 p.

3. Scott Jernigan «CompTIA Network+ Certification All-in-One Exam Guide, Eighth Edition». 2022. – 976 p.

4. Doug Lowe «Networking For Dummies 12th Edition». 2020. – 480 p.

5. Ramon Nastase «Computer Networking: The Beginner's guide for Mastering Computer Networking, the Internet and the OSI Model». 2018. – 186 p.

6. Russ White & Ethan Banks «Computer Networking Problems and Solutions: An Innovative Approach to Building Resilient, Modern Networks». 2017. – 832 p.

7. Kuznetsov, O., Kryvinska, N., Ilchenko, O., Smirnova, T., Ulianovska, Y. «Comparative Analysis of Cryptocurrency Trading Platforms Using the Analytic Hierarchy Process». *CEUR Workshop Proceedings*, 2023, 3628, pp. 106-115.

8. Al-Mudhafar Aqeel, A.M., Smirnova, T., Buravchenko, K., Smirnov, O. «The method of assessing and improving the user experience of subscribers in software-configured networks based on the use of machine learning». *Advanced Information Systems*, 2023, 7(2), pp. 49-56.

9. Smirnov, O., Sydorenko, V., Aleksander, M., Zhyharevych, O., Yenchev, S. «Simulation of the cloud IoT-based monitoring system for critical infrastructures». *CEUR Workshop Proceedings,* Volume 3530, 2023, pp. 256-265.

10. Smirnov, O., Odarchenko, R., Smirnova, T., Bondar, S., Volosheniuk, D. «Optimal Structure Construction of Private 5G Network for the Needs of Enterprises». *Lecture Notes on Data Engineering and Communications Technologies*, 2023, 178, pp. 208–223.

11. Smirnova, T., Gnatyuk, S., Yudin, O., Sydorenko, V., Polozhentsev, A., «The Model for Calculating the Quantitative Criteria for Assessing the Security Level of Information and Telecommunication Systems». CEUR Workshop Proceedings Volume 3156, 2022, Pages 390-399.

12. Smirnova T., Gnatyuk S., Berdibayev R., Avkurova Zh., Iavich M. «Cloud-Based Cyber Incidents Response System and Software Tools». *Communications in Computer and Information Science*, 2021, vol 1486. Springer, Cham. pp 169-184.

13. Smirnov O., Kuznetsov A., Kiian A., Kuznetsova T. «Non-binary constant weight coding technique». *CEUR Workshop Proceedings*. Volume 2740, 2020, Pages 102-114.

14. Smirnov O., Alimseitova Zh., Adranova A., Akhmetov B., Lakhno V., Zhilkishbayeva G. «Models and algorithms for ensuring functional stability and cybersecurity of virtual cloud resources». *Journal of theoretical and applied information technology* Vol.98. No 21, 2020, P. 3334-3346.

15. Smirnov O., Kuznetsov A., Kiian A., Cherep A., Kanabekova M., Chepurko I. «Testing of code-based pseudorandom number generators for post-quantum application». *2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT),* Ukraine, Kyiv, May 14-18. 2020. P. 172-177.

16. Smirnov O., Kuznetsov A., Pushkar'ov A., Serhiienko R., Babenko V., Kuznetsova T., «Representation of Cascade Codes in the Frequency Domain». In: Radivilova T., Ageyev D., Kryvinska N. (eds) Data-Centric Business and Applications. Lecture Notes on Data Engineering and Communications Technologies, vol 48. Springer, Cham. 2021. pp 557-587.

17. Smirnov, O., Markovets, O. Vovk, N., Turchyn, Y., «Model of informational support for social network administrators' content creation». *CEUR Workshop Proceedings* Volume 2616, 2020, Pages 125-136.

18. Smirnov, O., Drieieva, H., Drieiev, O., Polishchuk, Y., Brzhanov, R., Aleksander, M. «Method of fractal traffic generation by a model of generator on the graph». *CEUR Workshop Proceedings* Volume 2616, 2020, Pages 366-379.

19. Smirnov, O., Drieieva, H., Drieiev, O., Simakhin, V., Bondar, S., Odarchenko, R. «Managing multifractal properties of the binary sequence generated with the Markov chains», *CEUR Workshop Proceedings* Volume 2608, 2020, Pages 633-645.

20. Smirnov O. Kuznetsov A., Zaicenko Yu., Pastukhov M., Oleshko O., Kuznetsova K., «Formation of Discrete Signals with Special Correlation Properties». *International Conference on Information and Telecommunication Technologies and Radio Electronics, UkrMiCo 2019*; Odessa; Ukraine; 9-13 September 2019. P.22-28.

21. Smirnov, O., Kuznetsov, A., Kolovanova, I., Kuznetsova, T., «Noise immunity of the algebraic geometric codes». International Journal of Computing; 2019, Volume 18, Issue 4 – Research Institute for Intelligent Computer Systems – 2019. – P. 393-407.

22. Smirnov, O., Kuznetsov, A., Reshetniak, O., Ivko, N., Katkova, T., Kuznetsova, T., «Generators of Pseudorandom Sequence with Multilevel Function of Correlation». *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T),* Kyiv, Ukraine, 8 – 11 October 2019 . P.517-522.

23. Smirnov, O., Odarchenko, R., Abakumova, A., Usik, P., Kundyz, M., «QoE optimization technique for media delivery in 5G networks». *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T),* Kyiv, Ukraine, 8 – 11 October 2019. P.597-601.

24. Smirnov, O., Krasnobayev, V., Yanko, A., Kuznetsova, T. «Methods of nulling numbers in the system of residual classes». *CEUR Workshop Proceedings*, Vol 2588, P. 90-106, 2019.

25. Smirnov, O., Kuznetsov, A., Kovalchuk, D., Averchev, A., Pastukhov, M., Kuznetsova, K., «Formation of Pseudorandom Sequences with Special Correlation

Properties», *2019 3rd International Conference on Advanced Information and Communications Technologies, AICT -2019*/ Lviv, Ukraine, 2-6 July, 2019, P. 395-399.

26. Smirnov, O., Kuznetsov, A., Kiian, A., Zamula, A., Rudenko, S., Hryhorenko, V., «Variance Analysis of Networks Traffic for Intrusion Detection in Smart Grids», *2019 IEEE 6th International Conference On Energy Smart Systems (2019 IEEE ESS)*, Kyiv, Ukraine April 17-19, 2019 P. 353-358.

27. Smirnov, O., Kuznetsov, A., Kavun, S., Babenko, B., Nakisko, O., Kuznetsova, K., «Malware Correlation Monitoring in Computer Networks of Promising Smart Grids», *2019 IEEE 6th International Conference On Energy Smart Systems (2019 IEEE ESS)*, Kyiv, Ukraine April 17-19, 2019 P. 347-352.

28. Smirnov, O., Kuznetsov, A., Kovalchuk, D., Pastukhov, M., Kuznetsova, K., Prokopovych-Tkachenko, D., «Discrete Signals with Special Correlation Properties», *CEUR Workshop Proceedings* Volume 2353, *CEUR Workshop Proceedings* 2019, Pages 618-629.

29. Smirnov A.A., Kuzne tsov A.A., Danilenko D.A., Berezovsky A., «The statistical analysis of a network traffic for the intrusion detection and prevention systems», *Telecommunications and Radio Engineering*. – Volume 74, Issue 1. – Begel House Inc. – 2015. – P. 61-78.

30. Смірнова Т.В., Конопліцька-Слободенюк О.К., Буравченко К.О., Смірнов С.А., Кравчук О.В., Козірова Н.Л., Смірнов О.А. «Дослідження технологій забезпечення кібербезпеки хмарних сервісів IaaS, PaaS та SaaS». *Кібербезпека: освіта, наука, техніка.* 2024. №4(24), С. 6-27.

31. Батрак О., Смірнова Т., Гнатюк В., Одарченко Р., Смірнов О. «Дослідження показників ефективності функціонування та перспектив розвитку систем IP-телефонії». *Підводні технології,* 2024, № 13, с. 28-35.

32. Аль-Мудхафар Акіл Абдулхуссейн М., Смірнова Т.В., Буравченко К.О., Смірнов О.А. «Метод оцінки та підвищення користувальницького досвіду абонентів в програмно-конфігурованих мережах на основі використання машинного навчання». *Сучасні інформаційні системи,* 2023, том 7, № 2, С. 49-56.

33. Смірнова Т.В., Гнатюк С.О., Сидоренко В.М., Юдін О.Ю., Сидоренко С.Ю., «Модель визначення критичності галузевих інформаційно-телекомунікаційних систем». *Проблеми інформатизації та управління,* № 2(70). 2022. С. 28-37.

34. Смірнов О.А., Смірнова Т.В., Якименко Н.М., Смірнов С.А., Поліщук Л.І., «Дослідження стійкості до диференціального криптоаналізу запропонованої функції гешування удосконаленого модуля криптографічного захисту в інформаційно-комунікаційних системах» *Системи управління, навігації та зв'язку,* 2022, № 3(69). С. 93-98.

35. Смірнов О.А., Смірнова Т.В., Якименко Н.М., Поліщук Л.І., Смірнов С.А. «Дослідження статистичної стійкості та швидкісних характеристик запропонованої функції гешування удосконаленого модуля криптографічного захисту в інформаційно-комунікаційних системах» *Вісник Хмельницького національного університету. Серія: «Технічні науки»,* № 2 (307). С. 46-52. 2022.

36. Смірнов О.А., Смірнова Т.В., Константинова Л.В., Смірнов С.А., Якименко Н.М., «Дослідження стійкості до лінійного криптоаналізу запропонованої функції гешування удосконаленого модуля криптографічного захисту в інформаційно-комунікаційних системах» *Системи управління, навігації та зв'язку*, 2022, № 1(67). С. 84-89.

37. Смірнов О.А., Смірнова Т.В., Буравченко К.О., Кравченко С.С., Горбов В.О., «Хмарна система підтримки прийняття рішень технологічного процесу відновлення поверхонь конструкцій і деталей машин». *Сучасні інформаційні системи*. 2021. Т. 5, № 4. С. 79-95

38. Смірнов О.А., Усік П.С., Миронець І.В., Буравченко К.О., Якименко Н.М. «Метод підвищення ефективності розподіленої обробки даних у комп'ютерних системах операторів стільникового зв'язку» *Вісник Черкаського державного технологічного університету. Технічні науки.* №4. С. 103-110. 2020.

39. О.А.Смірнов, Т.В.Смірнова, Л.І. Поліщук, К.О. Буравченко, А.О.Макевнін, «Дослідження хмарних технологій як сервісів», *Кібербезпека: освіта, наука, техніка*. № 3(7). С. 43-62. 2020.

40. Смірнов О.А., Конопліцька-Слободенюк О.К., Смірнов С.А., Буравченко К.О., Смірнова Т.В., Поліщук Л.І. Інформаційна безпека в комп'ютерних мережах. Навчальний посібник – Кропивницький: вид. Лисенко В.Ф. 2020. – 294 с.

41. О.А. Смірнов, П.С. Усік, «Дослідження перспектив використання технологічних рішень в мережах 5G» у *Кібербезпека та інформаційні технології: монографія.* – Х. : ТОВ «ДІСА ПЛЮС», 2020.С. 122-135.

42. Смірнов О.А., Дрєєва Г.М., Дрєєв О.М., Смірнова Т.В. «Фрактальний аналіз генератора самоподібного трафіку на основі ланцюга Маркова». *Центральноукраїнський науковий вісник. Технічні науки.* № 2(33). с. 161-172, 2019.

43. Смірнов О.А., Конопліцька-Слободенюк О.К., Смірнов С.А., Буравченко К.О., Смірнова Т.В. Поліщук Л.І. Проектування комп'ютерних систем та мереж. Навчальний посібник – Кропивницький: вид. Лисенко В.Ф. 2019. – 264 с.

44. Smirnov, O., Kuznetsov, A., Kuznetsova., K. Synthesis of Discrete Signals with Improved Correlation Properties. Монографія: In.: ISCI'2019: Information Security in Critical Infrastructures. Collective monograph. Edited by Ivan D. Gorbenko and Alexandr A. Kuznetsov, ASC Academic Publishing, USA, 2019, pp. 281-299. – ISBN: 978-0-9989826-8-7 (Hardback), ISBN: 978-0-9989826-9-4 (Ebook).

45. Смірнов О.А., Дрєєва Г.М. Метод генерування фрактального трафіку за допомогою моделі генератора на графі. Монографія: Інформаційна безпека та інформаційні технології : монографія / за заг. ред. В. С. Пономаренка. – Х. : Вид. Рожко С.Г. 2019. С. 123-139

46. Дрєєва Г.М., Смірнов О.А., Дрєєв О.М. Метод генерування фрактальноподібної числової послідовності на основі скінченного автомату для

моделювання трафіку у мережі. Центральноукраїнський науковий вісник. Технічні науки. № 1(32). с. 173-183, 2019.

47. Смірнова Т.В., Солових Є.К., Смірнов О.А., Дрєєв О.М. Побудова хмарних інформаційних технологій оптимізації технологічного процесу відновлення та зміцнення поверхонь деталей. Центральноукраїнський науковий вісник. Технічні науки. № 1(32). с. 184-194, 2019.

48. Смірнов О.А., Смірнов С.А., Поліщук Л.І., Смірнова Т.В., Конопліцька-Слободенюк О.К. Метод формування антивірусного захисту даних з використанням безпечної маршрутизації метаданих. Кібербезпека: освіта, наука, техніка. – Том 3 № 3. – Київ: КУ ім. Бориса Грінченка. – 2019. – С. 63-87.

49. Смірнов О.А., Гнатюк С.О., Кавун С.В., Терейковський І.А., Жмурко Т.О., Смірнов С.А., Коваленко А.С. Основи безпеки в комп'ютерних мережах. Навчальний посібник – Кропивницький: вид. Лисенко В.Ф. 2018. – 177 с.

50. Смірнов О.А., Котелянець В.В. Стійкі до колізій стохастичні моделі функціонування безпроводових сенсорних мереж. Вісник інженерної академії України, №3, с. 145-152, 2018

51. Смірнов О.А., Смірнов С.А., Дідик А.К., Дрєєв А.М. Алгоритми формування безлічі маршрутів передачі метаданих у антивірусні хмарні системи. Збірник наукових праць "Системи обробки інформації". - Випуск 5 (142). - Х.: ХУПС - 2016. - С. 148-152.

52. Смірнов О.А., Смірнов С.А. Дідик А.К., Дрєєв О.М. Моделі системи нейромережевих експертів безпечної маршрутизації у хмарних антивірусних системах. Збірник наукових праць "Системи обробки інформації". - Випуск 3 (140). - Х.: ХУПС - 2016. - С. 36-39.

Додаток А
(обов'язковий)

Технічне завдання

Зміст

1 Найменування та область застосування

Це технічне завдання розповсюджується на розробку системи віртуалізації з підтримкою розподіленого сховища.

2 Підстава для розробки

Підставою для розробки служить завдання на випускну кваліфікаційну роботу за першим (бакалаврським) рівнем вищої освіти, видане на кафедрі кібербезпеки та програмного забезпечення (нак. № 48-02 від 17.01.2025 року).

3 Мета та призначення розробки

Метою випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти є розробка програмного забезпечення системи віртуалізації з підтримкою розподіленого сховища.

4 Джерела розробки

Джерелом цієї випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти є стосовна до теми література і існуючі аналоги.

5 Технічні вимоги

5.1 Склад продукції

Складниками розробки є:

– вибір і обґрунтування методів реалізації проекту;

– розробка програмної частин системи, а також розробка взаємодії системи з ОС та з користувачем;

– розробка програми, що реалізує спроектовані алгоритми роботи системи.

5.2 Показники призначення

Система повинна забезпечувати:

– системи віртуалізації з підтримкою розподіленого сховища;

– цілісність даних у процесі роботи та при зберіганні;

– простий, інтуїтивно зрозумілий інтерфейс.

5.3 Вимоги до функціональних характеристик

Розроблене програмне забезпечення не повинно мати обмежень на версію драйверів та операційної системи.

5.4 Вимоги до архітектури

Компонент, що розробляється повинен використовувати системні засоби та апаратні засоби, що на даному етапі розвитку обчислювальної техніки найбільше поширені.

5.5 Вимоги до надійності

Програмні модулі написані по всім правилам, які стосуються стандартних викликів процедур, функцій, методів і форм, визначених технічною документацією на середовище розробки.

5.6 Умови експлуатації

Робочі місця користувачів ПЗ повинні задовольняти наступним умовам експлуатації:

– температура повітря: 19-20 град. по Цельсію;

– відносна вологість повітря до 80%;

– атмосферний тиск 107 кПа.

5.7 Вимоги до складу та параметрів технічних засобів

Програмне забезпечення повинно бути реалізоване на ПЕОМ, працювати в ОС Windows 10/11 і з сумісними з цією платформою пристроями і прикладним програмним забезпеченням.

5.8 Вимоги до інформаційної і програмної сумісності

Переносність програмного забезпечення повинна бути забезпечена за рахунок його реалізації стандартного інтерфейсу взаємодії з ОС, що працюють під управлінням ОС Windows 10/11.

5.8.1 Обладнання

Комп’ютер Intel® Celeron/8 Mb/1.2 Gb/SVGA 14” 1Mb або сумісні з ним.

5.8.2 Мова програмування

Середовище Python.

5.8.3 Вхідні дані

Опис алгоритму роботи запропонованої системи.

5.8.4 Вихідні дані

Робоча програма.

6 Вимоги до програмної документації

Програмна продукція повинна бути представлена у виді опису структури даних, схем та опису алгоритму, а також текстів вихідних модулів програмного забезпечення згідно ЄСПД .

7 Перелік документів, що розробляються

- Структурна схема системи – 1 аркуш.
- Функціональна схема системи – 1 аркуш.
- Діаграма процесів – 1 аркуш.
- Блок-схема алгоритму роботи програми – 2 аркуша.
- Пояснювальна записка – 82 аркуші.

8 Етапи розробки

8.1 Збір і обробка інформації по темі випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти. Постановка задачі на виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти (складання ТЗ).

8.2 Проведення досліджень або експериментальних робіт для уточнення основних положень випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти.

8.3 Розробка функціональних схем, блок схем алгоритмів роботи програмного забезпечення.

8.4 Побудова схем взаємодії даних.

8.5 Створення прототипу ПЗ.

8.6 Віднаходження ПЗ, аналіз отриманих результатів.

8.7 Оформлення пояснювальної записки і виконання робіт по графічній частині.

9 Порядок контролю та приймання

9.1 Подання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти на попередній захист 23.05.2025 р.

9.2 Подання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти на захист 5.06.2025 р.

Додаток Б

(обов’язковий)

Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

ЗАТВЕРДЖУЮ

Керівник випускної кваліфікаційної роботи за

першим (бакалаврським) рівнем вищої освіти

_____ Коваленко А.С.

Програмне забезпечення системи віртуалізації з підтримкою розподіленого сховища

Лістинг програми

Код документу 12

Носій: CD/DVD-диск / USB-флеш-накопичувач

Загальна кількість аркушів: 29

Літера: РП

Кропивницький – 2025 року

Основна програма

```
#!/usr/bin/env python3
# Імпортуємо необхідні модулі
import random
import time
import threading
import logging

# Налаштування системи логування
logging.basicConfig(level=logging.INFO)

# Оголошення глобальних констант
MAX_VM_ID = 9999
MIN_VM_ID = 1000

# Оголошення класу VirtualMachine для управління віртуальними машинами
class VirtualMachine:
# Конструктор класу VirtualMachine
    def __init__(self, vm_id, name, cpu, memory, storage):
# Призначаємо ідентифікатор віртуальної машини
        self.id = vm_id
# Призначаємо ім'я віртуальної машини
        self.name = name
# Задаємо кількість CPU
        self.cpu = cpu
# Задаємо обсяг пам'яті
        self.memory = memory
# Задаємо розмір сховища
        self.storage = storage
# Початковий стан віртуальної машини встановлюємо як 'stopped'
        self.status = 'stopped'
# Логування створення віртуальної машини
        logging.info("Створено VM: %s", self.name)

# Метод для запуску віртуальної машини
    def start(self):
# Змінюємо стан на 'running'
        self.status = 'running'
# Логування запуску віртуальної машини
        logging.info("VM %s запущено", self.name)

# Метод для зупинки віртуальної машини
    def stop(self):
# Змінюємо стан на 'stopped'
        self.status = 'stopped'
# Логування зупинки віртуальної машини
        logging.info("VM %s зупинено", self.name)

# Метод для перезавантаження віртуальної машини
    def reboot(self):
# Встановлюємо стан 'stopped' перед перезавантаженням
        self.status = 'stopped'
# Логування початку перезавантаження
        logging.info("VM %s перезавантажується", self.name)
# Затримка для імітації процесу перезавантаження
        time.sleep(1)
# Встановлюємо стан 'running' після перезавантаження
        self.status = 'running'
# Логування завершення перезавантаження
        logging.info("VM %s перезапущено", self.name)

# Метод для отримання інформації про віртуальну машину
```

```
    def info(self):
# Повертаємо словник з інформацією про VM
        return {
            "id": self.id,
            "name": self.name,
            "cpu": self.cpu,
            "memory": self.memory,
            "storage": self.storage,
            "status": self.status
        }

# Оголошення класу StorageNode для керування окремими вузлами сховища
class StorageNode:
# Конструктор класу StorageNode
    def __init__(self, node_id, capacity):
# Призначення ідентифікатора вузла
        self.id = node_id
# Встановлення загальної місткості вузла
        self.capacity = capacity
# Ініціалізація використаного простору як 0
        self.used = 0
# Ініціалізація словника для зберігання файлів
        self.files = {}
# Логування створення вузла сховища
        logging.info("Створено вузол сховища з ID: %s", self.id)

# Метод для збереження файлу на вузлі
    def store_file(self, file_name, size):
# Перевірка доступності місця для збереження файлу
        if self.used + size <= self.capacity:
# Додавання файлу до словника файлів
            self.files[file_name] = size
# Оновлення використаного простору
            self.used += size
# Логування успішного збереження файлу
            logging.info("Файл '%s' збережено на вузлі %s", file_name, self.id)
# Повернення успішного результату
            return True
# Якщо місця недостатньо, повертаємо False
        return False

# Метод для видалення файлу з вузла
    def remove_file(self, file_name):
# Перевірка наявності файлу у вузлі
        if file_name in self.files:
# Отримання розміру файлу
            size = self.files.pop(file_name)
# Зменшення використаного простору
            self.used -= size
# Логування видалення файлу
            logging.info("Файл '%s' видалено з вузла %s", file_name, self.id)
# Повернення успішного результату
            return True
# Якщо файл не знайдено, повертаємо False
        return False

# Метод для отримання статусу використання вузла
    def get_usage(self):
# Повертаємо інформацію про використання, місткість та вільний простір
        return {
            "used": self.used,
            "capacity": self.capacity,
            "free": self.capacity - self.used
```

```
        }

# Оголошення класу DistributedStorage для управління розподіленим сховищем
class DistributedStorage:
# Конструктор класу DistributedStorage
    def __init__(self, replication_factor=2):
# Ініціалізація списку вузлів сховища
        self.nodes = []
# Встановлення коефіцієнта реплікації
        self.replication_factor = replication_factor
# Логування ініціалізації розподіленого сховища
        logging.info("Розподілене сховище ініціалізовано з реплікацією: %d",
self.replication_factor)

# Метод для додавання нового вузла в систему
    def add_node(self, node):
# Додавання вузла до списку
        self.nodes.append(node)
# Логування додавання вузла
        logging.info("Додано вузол сховища з ID: %s", node.id)

# Метод для видалення вузла за його ідентифікатором
    def remove_node(self, node_id):
# Перебір вузлів у списку
        for node in self.nodes:
# Перевірка збігу ідентифікатора
            if node.id == node_id:
# Видалення вузла зі списку
                self.nodes.remove(node)
# Логування видалення вузла
                logging.info("Видалено вузол сховища з ID: %s", node_id)
# Переривання циклу після видалення
                break

# Метод для збереження даних з реплікацією на кілька вузлів
    def store_data(self, file_name, size):
# Ініціалізація списку вузлів, на яких збережено файл
        stored_on = []
# Ітерація по всіх вузлах
        for node in self.nodes:
# Спроба зберегти файл на поточному вузлі
            if node.store_file(file_name, size):
# Додавання ідентифікатора вузла до списку успішних збережень
                stored_on.append(node.id)
# Перевірка досягнення необхідної кількості реплік
                if len(stored_on) >= self.replication_factor:
# Завершення циклу при досягненні реплікації
                    break
# Повернення списку вузлів, на яких збережено файл
        return stored_on

# Метод для видалення файлу з усіх вузлів
    def remove_data(self, file_name):
# Ініціалізація списку вузлів, з яких видалено файл
        removed_from = []
# Перебір всіх вузлів
        for node in self.nodes:
# Спроба видалити файл з поточного вузла
            if node.remove_file(file_name):
# Додавання ідентифікатора вузла до списку видалень
                removed_from.append(node.id)
# Повернення списку вузлів, з яких видалено файл
        return removed_from
```

```
# Метод для запиту статусу всіх вузлів сховища
    def query_status(self):
# Ініціалізація словника для зберігання статусу вузлів
        status = {}
# Перебір кожного вузла в системі
        for node in self.nodes:
# Отримання статусу використання для поточного вузла
            status[node.id] = node.get_usage()
# Повернення зібраного статусу вузлів
        return status

# Оголошення класу VirtualizationSystem для управління системою віртуалізації
class VirtualizationSystem:
# Конструктор класу VirtualizationSystem
    def __init__(self):
# Ініціалізація словника для зберігання віртуальних машин
        self.virtual_machines = {}
# Ініціалізація розподіленого сховища
        self.storage_system = DistributedStorage()
# Логування створення системи віртуалізації
        logging.info("Система віртуалізації ініціалізована")

# Метод для створення нової віртуальної машини
    def create_vm(self, name, cpu, memory, storage):
# Генерація випадкового ідентифікатора для віртуальної машини
        vm_id = random.randint(MIN_VM_ID, MAX_VM_ID)
# Створення екземпляру VirtualMachine
        vm = VirtualMachine(vm_id, name, cpu, memory, storage)
# Додавання віртуальної машини до словника
        self.virtual_machines[vm_id] = vm
# Логування створення віртуальної машини з детальною інформацією
        logging.info("Створено VM: %s", vm.info())
# Повернення екземпляру віртуальної машини
        return vm

# Метод для видалення віртуальної машини за ідентифікатором
    def delete_vm(self, vm_id):
# Перевірка наявності віртуальної машини у системі
        if vm_id in self.virtual_machines:
# Отримання інформації про віртуальну машину перед видаленням
            vm = self.virtual_machines.pop(vm_id)
# Логування видалення віртуальної машини
            logging.info("Видалено VM: %s", vm.info())
# Повернення успішного результату видалення
            return True
# Якщо VM не знайдено, повернення False
        return False

# Метод для отримання списку всіх віртуальних машин
    def list_vms(self):
# Повернення списку інформації про всі VM
        return [vm.info() for vm in self.virtual_machines.values()]

# Метод для додавання вузла сховища до системи
    def add_storage_node(self, node):
# Виклик методу додавання вузла в розподілене сховище
        self.storage_system.add_node(node)
# Логування додавання вузла через систему
        logging.info("Додано вузол сховища через систему: %s", node.id)

# Метод для видалення вузла сховища за ідентифікатором
    def remove_storage_node(self, node_id):
```

```
# Виклик методу видалення вузла з розподіленого сховища
        self.storage_system.remove_node(node_id)
# Логування видалення вузла через систему
        logging.info("Видалено вузол сховища через систему: %s", node_id)

# Метод для збереження файлу в розподіленому сховищі
    def store_file(self, file_name, size):
# Виклик методу збереження даних у розподілене сховище
        nodes = self.storage_system.store_data(file_name, size)
# Логування інформації про розміщення файлу
        logging.info("Файл '%s' збережено на вузлах: %s", file_name, nodes)
# Повернення списку вузлів, на яких збережено файл
        return nodes

# Метод для видалення файлу з розподіленого сховища
    def remove_file(self, file_name):
# Виклик методу видалення даних з розподіленого сховища
        nodes = self.storage_system.remove_data(file_name)
# Логування інформації про видалення файлу
        logging.info("Файл '%s' видалено з вузлів: %s", file_name, nodes)
# Повернення списку вузлів, з яких видалено файл
        return nodes

# Метод для запиту статусу розподіленого сховища
    def query_storage(self):
# Отримання статусу вузлів розподіленого сховища
        status = self.storage_system.query_status()
# Логування поточного статусу сховища
        logging.info("Статус розподіленого сховища: %s", status)
# Повернення зібраного статусу
        return status

# Функція для симуляції операцій над віртуальними машинами
def query_vm_operations(system):
# Створення тестової віртуальної машини
    vm1 = system.create_vm("TestVM1", cpu=2, memory=4096, storage=50)
# Затримка для симуляції часу запуску
    time.sleep(0.5)
# Запуск віртуальної машини
    vm1.start()
# Затримка для симуляції роботи машини
    time.sleep(0.5)
# Перезавантаження віртуальної машини
    vm1.reboot()
# Затримка після перезавантаження
    time.sleep(0.5)
# Зупинка віртуальної машини
    vm1.stop()
# Повернення інформації про VM після операцій
    return vm1.info()

# Функція для симуляції операцій над розподіленим сховищем
def query_storage_operations(system):
# Створення першого вузла сховища
    node1 = StorageNode(1, capacity=500)
# Створення другого вузла сховища
    node2 = StorageNode(2, capacity=500)
# Створення третього вузла сховища
    node3 = StorageNode(3, capacity=500)
# Додавання першого вузла до системи
    system.add_storage_node(node1)
# Додавання другого вузла до системи
    system.add_storage_node(node2)
```

```
# Додавання третього вузла до системи
    system.add_storage_node(node3)
# Затримка для стабілізації операцій
    time.sleep(0.5)
# Спроба зберегти файл у розподіленому сховищі
    nodes_used = system.store_file("data1.bin", 100)
# Затримка після збереження файлу
    time.sleep(0.5)
# Видалення файлу з розподіленого сховища
    system.remove_file("data1.bin")
# Отримання поточного статусу сховища
    status = system.query_storage()
# Повернення статусу сховища
    return status

# Функція для виконання періодичних перевірок системи
def perform_system_checks(system):
# Ітерація для виконання декількох перевірок
    for i in range(3):
# Логування початку ітерації перевірки
        logging.info("Виконання перевірки системи, ітерація: %d", i+1)
# Отримання списку всіх віртуальних машин
        vm_list = system.list_vms()
# Отримання поточного статусу розподіленого сховища
        storage_status = system.query_storage()
# Логування списку віртуальних машин
        logging.info("Список VM: %s", vm_list)
# Логування статусу розподіленого сховища
        logging.info("Статус сховища: %s", storage_status)
# Затримка між перевірками
        time.sleep(1)
# Повернення успішного завершення перевірок
    return True

# Функція для обслуговування віртуальних машин у окремому потоці
def vm_maintenance_task(system):
# Безкінечний цикл для періодичного обслуговування
    while True:
# Логування початку обслуговування віртуальних машин
        logging.info("Виконання обслуговування VM")
# Перебір усіх віртуальних машин у системі
        for vm in list(system.virtual_machines.values()):
# Перевірка, чи знаходиться VM у стані 'stopped'
            if vm.status == "stopped":
# Логування виявлення зупиненої VM
                logging.info("Знайдено зупинену VM, запускаємо: %s", vm.name)
# Запуск зупиненої VM
                vm.start()
# Якщо VM у стані 'running'
            elif vm.status == "running":
# Випадковий вибір дії для VM
                if random.choice([True, False]):
# Логування випадкового перезавантаження VM
                    logging.info("Випадкове перезавантаження VM: %s", vm.name)
# Перезавантаження VM
                    vm.reboot()
# Затримка між обслуговуваннями
        time.sleep(2)

# Функція для обслуговування вузлів розподіленого сховища у окремому потоці
def storage_maintenance_task(system):
# Безкінечний цикл для періодичного обслуговування сховища
    while True:
```

```
# Логування початку обслуговування сховища
        logging.info("Виконання обслуговування сховища")
# Отримання статусу розподіленого сховища
        status = system.query_storage()
# Перебір вузлів для перевірки завантаженості
        for node_id, node_status in status.items():
# Перевірка, чи перевищує використання 80% від загальної місткості
            if node_status["used"] > 0.8 * node_status["capacity"]:
# Логування попередження про завантаженість вузла
                logging.info("Вузол %s перевищує 80%% завантаження", node_id)
# Затримка між перевірками
        time.sleep(3)

# Функція для симуляції мережевих запитів
def simulate_network_queries():
# Ітерація для симуляції декількох мережевих запитів
    for i in range(5):
# Логування номеру мережевого запиту
        logging.info("Симуляція мережевого запиту, номер: %d", i+1)
# Формування тексту відповіді для запиту
        response = "Відповідь для запиту " + str(i+1)
# Логування отриманої відповіді
        logging.info("Отримано: %s", response)
# Затримка для симуляції часу відповіді
        time.sleep(0.3)
# Повернення успішного результату симуляції мережевих запитів
    return True

# Функція для симуляції реплікації файлів у розподіленому сховищі
def simulate_file_replication(system):
# Список файлів для збереження з їх розмірами
    files = [("fileA.txt", 20), ("fileB.txt", 50), ("fileC.txt", 75)]
# Ініціалізація словника для зберігання інформації про реплікацію
    replication_details = {}
# Перебір кожного файлу зі списку
    for file_name, size in files:
# Збереження файлу у розподіленому сховищі
        nodes = system.store_file(file_name, size)
# Запис інформації про вузли, на яких збережено файл
        replication_details[file_name] = nodes
# Затримка між збереженнями файлів
        time.sleep(0.4)
# Повернення деталей реплікації файлів
    return replication_details

# Функція для симуляції робочого навантаження на віртуальні машини
def simulate_vm_workload(system):
# Ініціалізація списку для зберігання ідентифікаторів VM
    vm_ids = []
# Створення декількох VM для симуляції навантаження
    for i in range(5):
# Створення VM з відповідними параметрами
        vm = system.create_vm("WorkloadVM" + str(i+1), cpu=4, memory=8192,
storage=100)
# Додавання ідентифікатора VM до списку
        vm_ids.append(vm.id)
# Затримка після створення кожної VM
        time.sleep(0.2)
# Запуск усіх створених VM
    for vm in list(system.virtual_machines.values()):
# Запуск VM
        vm.start()
# Затримка між запуском кожної VM
```

```
        time.sleep(0.1)
# Повернення списку ідентифікаторів створених VM
    return vm_ids

# Функція для ініціалізації системи віртуалізації
def initialize_system():
# Створення екземпляру VirtualizationSystem
    vsystem = VirtualizationSystem()
# Логування ініціалізації системи
    logging.info("Ініціалізація системи віртуалізації завершена")
# Додавання початкових вузлів розподіленого сховища
    for node_id in range(4, 7):
# Створення вузла сховища з заданою місткістю
        node = StorageNode(node_id, capacity=1000)
# Додавання вузла до системи
        vsystem.add_storage_node(node)
# Затримка після додавання кожного вузла
        time.sleep(0.2)
# Повернення ініціалізованої системи віртуалізації
    return vsystem

# Головний блок виконання програми
if __name__ == "__main__":
# Ініціалізація системи віртуалізації
    system = initialize_system()
# Створення окремого потоку для обслуговування віртуальних машин
    vm_thread = threading.Thread(target=vm_maintenance_task, args=(system,))
# Встановлення потоку як демонічного
    vm_thread.daemon = True
# Запуск потоку обслуговування VM
    vm_thread.start()
# Створення окремого потоку для обслуговування розподіленого сховища
    storage_thread = threading.Thread(target=storage_maintenance_task,
args=(system,))
# Встановлення потоку як демонічного
    storage_thread.daemon = True
# Запуск потоку обслуговування сховища
    storage_thread.start()
# Виконання симуляції операцій над VM
    query_vm_operations(system)
# Виконання симуляції операцій над сховищем
    query_storage_operations(system)
# Симуляція мережевих запитів
    simulate_network_queries()
# Симуляція реплікації файлів у розподіленому сховищі
    simulate_file_replication(system)
# Симуляція робочого навантаження на віртуальні машини
    simulate_vm_workload(system)
# Виконання періодичних перевірок системи
    perform_system_checks(system)
# Затримка для продовження роботи потоків
    time.sleep(10)
# Логування завершення симуляції системи
    logging.info("Симуляція системи завершена")
```

Файл VirtualMachine.py

```
#!/usr/bin/env python3
# Імпортуємо необхідні бібліотеки та модулі
import random
import time
import threading
import logging
import copy
import datetime
import sched

# Налаштування базового логування
logging.basicConfig(level=logging.INFO)

################################################################################
# БАЗОВИЙ КОД СИСТЕМИ ВІРТУАЛІЗАЦІЇ
################################################################################

# Константи для ідентифікації віртуальних машин
MAX_VM_ID = 9999
MIN_VM_ID = 1000

# Клас VirtualMachine для управління віртуальними машинами
class VirtualMachine:
# Конструктор для ініціалізації VM
    def __init__(self, vm_id, name, cpu, memory, storage):
# Ідентифікатор VM
        self.id = vm_id
# Ім'я VM
        self.name = name
# Кількість CPU
        self.cpu = cpu
# Обсяг пам'яті (МБ)
        self.memory = memory
# Обсяг дискового сховища (ГБ)
        self.storage = storage
# Початковий стан VM
        self.status = 'stopped'
# Зберігаємо історію операцій для аналітики
        self.history = []
# Логування створення VM
        logging.info("Створено VM: %s з ID: %d", self.name, self.id)

# Метод для запуску VM
    def start(self):
# Встановлюємо статус як running
        self.status = 'running'
# Додаємо запис в історію
        self.history.append((datetime.datetime.now(), "start"))
# Логування запуску
        logging.info("VM %s запущено", self.name)

# Метод для зупинки VM
    def stop(self):
# Встановлюємо статус як stopped
        self.status = 'stopped'
# Запис у історії операцій
        self.history.append((datetime.datetime.now(), "stop"))
# Логування зупинки
        logging.info("VM %s зупинено", self.name)

# Метод для перезавантаження VM
```

```
    def reboot(self):
# Запис про зупинку у історії
        self.history.append((datetime.datetime.now(), "reboot_start"))
# Встановлюємо статус як stopped
        self.status = 'stopped'
# Логування початку перезавантаження
        logging.info("VM %s перезавантажується", self.name)
# Затримка для імітації перезавантаження
        time.sleep(1)
# Запис про запуск у історії
        self.history.append((datetime.datetime.now(), "reboot_end"))
# Встановлюємо статус як running
        self.status = 'running'
# Логування завершення перезавантаження
        logging.info("VM %s перезапущено", self.name)

# Метод для отримання інформації про VM
    def info(self):
# Повертаємо словник з даними про VM
        return {
            "id": self.id,
            "name": self.name,
            "cpu": self.cpu,
            "memory": self.memory,
            "storage": self.storage,
            "status": self.status,
            "history": self.history
        }

# Клас StorageNode для управління вузлами сховища
class StorageNode:
# Конструктор для створення вузла
    def __init__(self, node_id, capacity):
# Ідентифікатор вузла
        self.id = node_id
# Загальна місткість вузла
        self.capacity = capacity
# Використаний простір
        self.used = 0
# Словник для зберігання файлів
        self.files = {}
# Логування створення вузла
        logging.info("Створено вузол сховища з ID: %s", self.id)

# Метод для збереження файлу
    def store_file(self, file_name, size):
# Перевірка доступного простору
        if self.used + size <= self.capacity:
# Додавання файлу до сховища
            self.files[file_name] = size
# Оновлення використаного простору
            self.used += size
# Логування збереження файлу
            logging.info("Файл '%s' збережено на вузлі %s", file_name, self.id)
# Повертаємо успіх операції
            return True
# Якщо місця недостатньо, повертаємо False
        return False

# Метод для видалення файлу з вузла
    def remove_file(self, file_name):
# Перевірка наявності файлу
        if file_name in self.files:
```

```
# Отримання розміру файлу
            size = self.files.pop(file_name)
# Оновлення використаного простору
            self.used -= size
# Логування видалення файлу
            logging.info("Файл '%s' видалено з вузла %s", file_name, self.id)
# Повертаємо успіх операції
            return True
# Якщо файл не знайдено, повертаємо False
        return False

# Метод для отримання статусу вузла
    def get_usage(self):
# Повертаємо інформацію про використання простору
        return {
            "used": self.used,
            "capacity": self.capacity,
            "free": self.capacity - self.used
        }

# Клас DistributedStorage для керування розподіленим сховищем
class DistributedStorage:
# Конструктор для ініціалізації системи сховища
    def __init__(self, replication_factor=2):
# Список вузлів сховища
        self.nodes = []
# Коефіцієнт реплікації даних
        self.replication_factor = replication_factor
# Логування ініціалізації сховища
        logging.info("Розподілене сховище ініціалізовано з реплікацією: %d",
self.replication_factor)

# Метод для додавання вузла
    def add_node(self, node):
# Додавання вузла до списку
        self.nodes.append(node)
# Логування додавання вузла
        logging.info("Додано вузол сховища з ID: %s", node.id)

# Метод для видалення вузла за ID
    def remove_node(self, node_id):
# Перебір вузлів для пошуку
        for node in self.nodes:
# Перевірка збігу ID
            if node.id == node_id:
# Видалення вузла зі списку
                self.nodes.remove(node)
# Логування видалення
                logging.info("Видалено вузол сховища з ID: %s", node_id)
# Завершення циклу після видалення
                break

# Метод для збереження даних з реплікацією
    def store_data(self, file_name, size):
# Список вузлів, на яких збережено файл
        stored_on = []
# Ітерація по вузлах
        for node in self.nodes:
# Спроба зберегти файл на поточному вузлі
            if node.store_file(file_name, size):
# Додавання ID вузла до списку
                stored_on.append(node.id)
# Перевірка досягнення необхідної кількості реплік
```

```
                if len(stored_on) >= self.replication_factor:
# При досягненні реплікації вихід з циклу
                    break
# Повертаємо список вузлів з файлом
        return stored_on

# Метод для видалення файлу з усіх вузлів
    def remove_data(self, file_name):
# Список вузлів, з яких видалено файл
        removed_from = []
# Ітерація по вузлах
        for node in self.nodes:
# Спроба видалення файлу
            if node.remove_file(file_name):
# Додавання ID вузла до списку
                removed_from.append(node.id)
# Повертаємо список вузлів
        return removed_from

# Метод для отримання статусу всіх вузлів
    def query_status(self):
# Словник для зберігання статусу
        status = {}
# Перебір вузлів
        for node in self.nodes:
# Отримання інформації про використання кожного вузла
            status[node.id] = node.get_usage()
# Повертаємо зібраний статус
        return status

# Клас VirtualizationSystem для управління всією системою
class VirtualizationSystem:
# Конструктор для ініціалізації системи віртуалізації
    def __init__(self):
# Словник для зберігання VM
        self.virtual_machines = {}
# Ініціалізація розподіленого сховища
        self.storage_system = DistributedStorage()
# Логування створення системи
        logging.info("Система віртуалізації ініціалізована")

# Метод для створення нової VM
    def create_vm(self, name, cpu, memory, storage):
# Генерація унікального ID для VM
        vm_id = random.randint(MIN_VM_ID, MAX_VM_ID)
# Створення екземпляру VM
        vm = VirtualMachine(vm_id, name, cpu, memory, storage)
# Додавання VM до словника
        self.virtual_machines[vm_id] = vm
# Логування створення VM з детальною інформацією
        logging.info("Створено VM: %s", vm.info())
# Повернення створеного екземпляру VM
        return vm

# Метод для видалення VM
    def delete_vm(self, vm_id):
# Перевірка наявності VM у системі
        if vm_id in self.virtual_machines:
# Отримання VM перед видаленням
            vm = self.virtual_machines.pop(vm_id)
# Логування видалення VM
            logging.info("Видалено VM: %s", vm.info())
# Повертаємо успішність видалення
```

```
            return True
# Якщо VM не знайдено, повертаємо False
        return False

# Метод для отримання списку всіх VM
    def list_vms(self):
# Повертаємо список інформації про кожну VM
        return [vm.info() for vm in self.virtual_machines.values()]

# Метод для додавання вузла сховища до системи
    def add_storage_node(self, node):
# Виклик методу додавання вузла до системи сховища
        self.storage_system.add_node(node)
# Логування додавання вузла через систему
        logging.info("Додано вузол сховища через систему: %s", node.id)

# Метод для видалення вузла сховища
    def remove_storage_node(self, node_id):
# Виклик методу видалення вузла зі сховища
        self.storage_system.remove_node(node_id)
# Логування видалення вузла через систему
        logging.info("Видалено вузол сховища через систему: %s", node_id)

# Метод для збереження файлу в системі
    def store_file(self, file_name, size):
# Виклик збереження файлу у розподілене сховище
        nodes = self.storage_system.store_data(file_name, size)
# Логування інформації про розміщення файлу
        logging.info("Файл '%s' збережено на вузлах: %s", file_name, nodes)
# Повертаємо список вузлів з файлом
        return nodes

# Метод для видалення файлу з системи
    def remove_file(self, file_name):
# Виклик видалення файлу з розподіленого сховища
        nodes = self.storage_system.remove_data(file_name)
# Логування видалення файлу
        logging.info("Файл '%s' видалено з вузлів: %s", file_name, nodes)
# Повертаємо список вузлів, з яких видалено файл
        return nodes

# Метод для отримання статусу сховища
    def query_storage(self):
# Отримання інформації про стан вузлів сховища
        status = self.storage_system.query_status()
# Логування статусу сховища
        logging.info("Статус розподіленого сховища: %s", status)
# Повертаємо зібраний статус
        return status

##############################################################################
# ФУНКЦІОНАЛЬНІ РОЗШИРЕННЯ
##############################################################################

##############################################################################
# 1. СИСТЕМА ЗНІМКІВ ТА РЕЗЕРВНОГО КОПІЮВАННЯ (SNAPSHOT & BACKUP SYSTEM)
##############################################################################
class SnapshotManager:
# Конструктор для ініціалізації менеджера знімків
    def __init__(self):
# Словник для зберігання знімків, ключ - ID VM, значення - список знімків
        self.snapshots = {}
# Логування ініціалізації SnapshotManager
```

```
        logging.info("Ініціалізовано менеджер знімків та резервного копіювання")

# Метод для створення знімка VM
    def create_snapshot(self, vm):
# Отримання поточного часу для відмітки часу знімка
        timestamp = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
# Створення знімка як копії інформації VM
        snapshot = copy.deepcopy(vm.info())
# Додавання відмітки часу до знімка
        snapshot['snapshot_time'] = timestamp
# Перевірка наявності знімків для даної VM
        if vm.id not in self.snapshots:
# Якщо знімків немає, створюємо новий список
            self.snapshots[vm.id] = []
# Додавання знімка до списку
        self.snapshots[vm.id].append(snapshot)
# Логування створення знімка
        logging.info("Створено знімок для VM %s в час %s", vm.name, timestamp)
# Повертаємо створений знімок
        return snapshot

# Метод для відновлення VM зі знімка
    def restore_snapshot(self, vm, snapshot_index=0):
# Перевірка наявності знімків для VM
        if vm.id in self.snapshots and len(self.snapshots[vm.id]) >
snapshot_index:
# Отримання вибраного знімка
            snapshot = self.snapshots[vm.id][snapshot_index]
# Логування відновлення знімка
            logging.info("Відновлення VM %s зі знімка, зробленого в %s",
vm.name, snapshot['snapshot_time'])
# Відновлення даних VM (імітація шляхом оновлення властивостей)
            vm.cpu = snapshot.get('cpu', vm.cpu)
            vm.memory = snapshot.get('memory', vm.memory)
            vm.storage = snapshot.get('storage', vm.storage)
            vm.status = snapshot.get('status', vm.status)
# Додавання запису в історію відновлення
            vm.history.append((datetime.datetime.now(), "restore"))
# Повертаємо відновлену VM
            return vm
# Якщо знімок не знайдено, логування помилки
        logging.error("Знімок для VM %s не знайдено", vm.name)
        return None

# Метод для переліку всіх знімків для VM
    def list_snapshots(self, vm):
# Перевірка наявності знімків для VM
        if vm.id in self.snapshots:
# Повертаємо список знімків
            return self.snapshots[vm.id]
# Якщо знімків немає, повертаємо порожній список
        return []

################################################################################
# 2. СИСТЕМА МОНІТОРИНГУ РЕСУРСІВ ТА ОПОВІЩЕННЯ (MONITORING & ALERTING)
################################################################################
class MonitoringSystem:
# Конструктор для ініціалізації системи моніторингу
    def __init__(self, system, cpu_threshold=80, memory_threshold=80):
# Збереження посилання на систему віртуалізації
        self.system = system
# Встановлення порогових значень для CPU
        self.cpu_threshold = cpu_threshold
```

```
# Встановлення порогових значень для пам'яті
        self.memory_threshold = memory_threshold
# Ініціалізація списку оповіщень
        self.alerts = []
# Прапорець для зупинки моніторингу
        self.running = False
# Логування ініціалізації MonitoringSystem
        logging.info("Ініціалізовано систему моніторингу з порогами CPU: %d%%,
Memory: %d%%", cpu_threshold, memory_threshold)

# Метод для запуску моніторингу в окремому потоці
    def start_monitoring(self, interval=5):
# Встановлення прапорця роботи моніторингу
        self.running = True
# Запуск потоку моніторингу
        threading.Thread(target=self._monitor_loop, args=(interval,),
daemon=True).start()
# Логування старту моніторингу
        logging.info("Моніторинг розпочато з інтервалом %d секунд", interval)

# Приватний метод циклічного моніторингу
    def _monitor_loop(self, interval):
# Цикл виконання моніторингу поки прапорець активний
        while self.running:
# Перевірка ресурсів для кожної VM
            for vm in self.system.virtual_machines.values():
# Імітація вимірювання використання CPU (рандомне значення)
                cpu_usage = random.randint(0, 100)
# Імітація вимірювання використання пам'яті (рандомне значення)
                mem_usage = random.randint(0, 100)
# Логування вимірювань
                logging.info("Моніторинг VM %s: CPU %d%%, Memory %d%%", vm.name,
cpu_usage, mem_usage)
# Перевірка чи перевищують значення пороги
                if cpu_usage > self.cpu_threshold:
# Додавання повідомлення про перевищення CPU
                    alert = f"ALERT: VM {vm.name} CPU usage high: {cpu_usage}%"
                    self.alerts.append(alert)
                    logging.warning(alert)
                if mem_usage > self.memory_threshold:
# Додавання повідомлення про перевищення пам'яті
                    alert = f"ALERT: VM {vm.name} Memory usage high:
{mem_usage}%"
                    self.alerts.append(alert)
                    logging.warning(alert)
# Перевірка стану сховища
            storage_status = self.system.query_storage()
# Логування статусу сховища
            for node_id, usage in storage_status.items():
# Обчислення відсотка використання
                percent_used = (usage['used'] / usage['capacity']) * 100
# Перевірка порогу використання сховища
                if percent_used > 80:
                    alert = f"ALERT: Storage Node {node_id} usage high:
{percent_used:.2f}%"
                    self.alerts.append(alert)
                    logging.warning(alert)
# Затримка між циклами моніторингу
            time.sleep(interval)

# Метод для зупинки моніторингу
    def stop_monitoring(self):
# Вимикаємо прапорець роботи моніторингу
```

```
        self.running = False
# Логування зупинки моніторингу
        logging.info("Моніторинг зупинено")

# Метод для отримання списку оповіщень
    def get_alerts(self):
# Повертаємо накопичені оповіщення
        return self.alerts

################################################################################
# 3. СИСТЕМА ПЛАНУВАННЯ ЗАВДАНЬ ТА АВТОМАТИЗАЦІЇ (TASK SCHEDULER)
################################################################################
class TaskScheduler:
# Конструктор для ініціалізації планувальника завдань
    def __init__(self):
# Ініціалізація внутрішнього планувальника з бібліотеки sched
        self.scheduler = sched.scheduler(time.time, time.sleep)
# Список для зберігання завдань
        self.tasks = []
# Прапорець для зупинки планувальника
        self.running = False
# Логування ініціалізації TaskScheduler
        logging.info("Ініціалізовано систему планування завдань")

# Метод для додавання завдання
    def add_task(self, delay, task_func, args=(), kwargs=None, task_name=""):
# Якщо kwargs не визначено, ініціалізуємо порожній словник
        if kwargs is None:
            kwargs = {}
# Створення завдання як кортеж з параметрами
        task = {"delay": delay, "func": task_func, "args": args, "kwargs":
kwargs, "name": task_name}
# Додавання завдання до списку
        self.tasks.append(task)
# Логування додавання завдання
        logging.info("Додано завдання '%s' з затримкою %d секунд", task_name,
delay)

# Метод для запуску планувальника завдань
    def run(self):
# Встановлення прапорця роботи планувальника
        self.running = True
# Запуск окремого потоку для виконання завдань
        threading.Thread(target=self._run_tasks, daemon=True).start()
# Логування запуску планувальника
        logging.info("Запущено планувальник завдань")

# Приватний метод для виконання завдань
    def _run_tasks(self):
# Для кожного завдання в списку
        for task in self.tasks:
# Планування виконання завдання через scheduler
            self.scheduler.enter(task["delay"], 1, task["func"], task["args"],
task["kwargs"])
# Логування планування завдання
            logging.info("Заплановано завдання '%s'", task["name"])
# Запуск scheduler (блокується до виконання всіх завдань)
        while self.running:
            self.scheduler.run(blocking=False)
            time.sleep(1)

# Метод для зупинки планувальника
    def stop(self):
```

```
# Зміна прапорця роботи планувальника
        self.running = False
# Логування зупинки планувальника
        logging.info("Планувальник завдань зупинено")

################################################################################
# 4. АНАЛІТИКА ТА ЗВІТНІСТЬ (ANALYTICS & REPORTING)
################################################################################
class AnalyticsManager:
# Конструктор для ініціалізації аналітичного менеджера
    def __init__(self, system):
# Збереження посилання на систему віртуалізації
        self.system = system
# Список для зберігання історичних даних
        self.records = []
# Логування ініціалізації AnalyticsManager
        logging.info("Ініціалізовано систему аналітики та звітності")

# Метод для запису події з системи
    def record_event(self, event_type, details):
# Отримання поточного часу
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
# Формування запису події
        record = {"timestamp": timestamp, "event": event_type, "details":
details}
# Додавання запису до історії
        self.records.append(record)
# Логування запису події
        logging.info("Записано подію: %s", record)

# Метод для генерації звіту про віртуальні машини
    def generate_vm_report(self):
# Отримання інформації про всі VM
        vm_report = self.system.list_vms()
# Формування текстового звіту
        report = "Звіт про віртуальні машини:\n"
# Ітерація по кожній VM для додавання інформації до звіту
        for vm in vm_report:
            report += f"VM {vm['name']} (ID: {vm['id']}): Статус:
{vm['status']}, CPU: {vm['cpu']}, Memory: {vm['memory']}, Storage:
{vm['storage']}\n"
        # Запис події генерації звіту
        self.record_event("VM_REPORT", report)
# Повернення згенерованого звіту
        return report

# Метод для генерації звіту про сховище
    def generate_storage_report(self):
# Отримання статусу розподіленого сховища
        storage_status = self.system.query_storage()
# Формування текстового звіту
        report = "Звіт про розподілене сховище:\n"
# Ітерація по кожному вузлу для додавання інформації до звіту
        for node_id, usage in storage_status.items():
            report += f"Вузол {node_id}: Використано: {usage['used']}, Загальна
місткість: {usage['capacity']}, Вільно: {usage['free']}\n"
        # Запис події генерації звіту
        self.record_event("STORAGE_REPORT", report)
# Повернення згенерованого звіту
        return report

# Метод для отримання всіх записів аналітики
    def get_all_records(self):
```

```
# Повертаємо історію записів
        return self.records

################################################################################
# 5. СИСТЕМА РОЗШИРЮВАННЯ ЧЕРЕЗ ПЛАГІНИ (PLUGIN SYSTEM)
################################################################################
class PluginManager:
# Конструктор для ініціалізації менеджера плагінів
    def __init__(self):
# Словник для зберігання плагінів, ключ - ім'я плагіна, значення - функція або
об'єкт плагіна
        self.plugins = {}
# Логування ініціалізації PluginManager
        logging.info("Ініціалізовано менеджер плагінів")

# Метод для реєстрації нового плагіна
    def register_plugin(self, name, plugin_func):
# Додавання плагіна до словника
        self.plugins[name] = plugin_func
# Логування реєстрації плагіна
        logging.info("Зареєстровано плагін: %s", name)

# Метод для виконання плагіна за його ім'ям
    def execute_plugin(self, name, *args, **kwargs):
# Перевірка наявності плагіна в системі
        if name in self.plugins:
# Логування виконання плагіна
            logging.info("Виконується плагін: %s", name)
# Виконання плагіна з переданими аргументами
            result = self.plugins[name](*args, **kwargs)
# Логування результату виконання плагіна
            logging.info("Плагін %s повернув результат: %s", name, result)
# Повернення результату виконання
            return result
        else:
# Логування помилки у випадку відсутності плагіна
            logging.error("Плагін %s не знайдено", name)
            return None

# Метод для отримання списку всіх зареєстрованих плагінів
    def list_plugins(self):
# Повертаємо список імен плагінів
        return list(self.plugins.keys())

################################################################################
# ФУНКЦІЇ ДЛЯ СИМУЛЯЦІЇ РОБОТИ СИСТЕМИ
################################################################################
# Функція для симуляції операцій над віртуальними машинами
def simulate_vm_operations(system, snapshot_mgr, analytics_mgr):
# Створення тестової VM
    vm1 = system.create_vm("SimVM1", cpu=2, memory=2048, storage=50)
# Затримка для симуляції роботи
    time.sleep(0.5)
# Запуск VM
    vm1.start()
# Створення знімка поточного стану VM
    snapshot_mgr.create_snapshot(vm1)
# Затримка
    time.sleep(0.5)
# Перезавантаження VM
    vm1.reboot()
# Запис операції у систему аналітики
    analytics_mgr.record_event("VM_OPERATION", f"VM {vm1.name} перезавантажено")
```

```
# Затримка
    time.sleep(0.5)
# Зупинка VM
    vm1.stop()
# Запис операції у систему аналітики
    analytics_mgr.record_event("VM_OPERATION", f"VM {vm1.name} зупинено")
# Повертаємо інформацію про VM
    return vm1.info()

# Функція для симуляції операцій над сховищем
def simulate_storage_operations(system, analytics_mgr):
# Створення вузлів сховища
    node1 = StorageNode(1, capacity=500)
    node2 = StorageNode(2, capacity=500)
    node3 = StorageNode(3, capacity=500)
# Додавання вузлів до системи
    system.add_storage_node(node1)
    system.add_storage_node(node2)
    system.add_storage_node(node3)
# Затримка
    time.sleep(0.5)
# Спроба зберегти файл
    system.store_file("backup.bin", 100)
# Запис події в аналітику
    analytics_mgr.record_event("STORAGE_OPERATION", "Файл 'backup.bin'
збережено")
# Затримка
    time.sleep(0.5)
# Видалення файлу
    system.remove_file("backup.bin")
# Запис події в аналітику
    analytics_mgr.record_event("STORAGE_OPERATION", "Файл 'backup.bin'
видалено")
# Повертаємо статус сховища
    return system.query_storage()

# Функція для демонстрації роботи плагінів
def sample_plugin_function(data):
# Проста обробка даних плагіном
    result = f"Оброблено дані: {data}"
# Повертаємо результат обробки
    return result

################################################################################
# ГОЛОВНА ФУНКЦІЯ ДЛЯ ЗАПУСКУ ВСІЄЇ СИСТЕМИ
################################################################################
def main():
# Ініціалізація системи віртуалізації
    system = VirtualizationSystem()
# Ініціалізація менеджера знімків
    snapshot_mgr = SnapshotManager()
# Ініціалізація системи моніторингу
    monitor = MonitoringSystem(system, cpu_threshold=70, memory_threshold=70)
# Запуск системи моніторингу
    monitor.start_monitoring(interval=3)
# Ініціалізація планувальника завдань
    scheduler = TaskScheduler()
# Запуск планувальника завдань
    scheduler.run()
# Ініціалізація аналітичного менеджера
    analytics_mgr = AnalyticsManager(system)
# Ініціалізація менеджера плагінів
    plugin_mgr = PluginManager()
```

```
# Реєстрація прикладного плагіна
    plugin_mgr.register_plugin("SamplePlugin", sample_plugin_function)
# Запуск симуляції операцій над VM
    simulate_vm_operations(system, snapshot_mgr, analytics_mgr)
# Запуск симуляції операцій над сховищем
    simulate_storage_operations(system, analytics_mgr)
# Додавання завдання до планувальника: створення знімка для всіх VM
    def snapshot_all_vms():
# Ітерація по всіх VM для створення знімка
        for vm in system.virtual_machines.values():
            snapshot_mgr.create_snapshot(vm)
# Запис події в аналітику
        analytics_mgr.record_event("TASK", "Створено знімки для всіх VM")
    scheduler.add_task(delay=5, task_func=snapshot_all_vms,
task_name="SnapshotAllVMs")
# Додавання завдання до планувальника: генерація звіту про VM
    def report_vm_status():
# Генерація звіту та вивід результату
        report = analytics_mgr.generate_vm_report()
        logging.info("\n%s", report)
    scheduler.add_task(delay=10, task_func=report_vm_status,
task_name="ReportVMStatus")
# Виконання плагіна для демонстрації
    plugin_result = plugin_mgr.execute_plugin("SamplePlugin", "Демо дані")
    logging.info("Результат роботи плагіна: %s", plugin_result)
# Основний цикл для підтримки роботи системи
    for i in range(15):
# Логування поточного циклу роботи
        logging.info("Основний цикл роботи системи, ітерація: %d", i+1)
# Запис події у систему аналітики
        analytics_mgr.record_event("MAIN_LOOP", f"Ітерація {i+1} виконана")
# Затримка для імітації роботи
        time.sleep(1)
# Зупинка моніторингу
    monitor.stop_monitoring()
# Зупинка планувальника завдань
    scheduler.stop()
# Генерація фінального звіту по аналітиці
    final_report = analytics_mgr.generate_vm_report()
# Логування фінального звіту
    logging.info("Фінальний звіт:\n%s", final_report)
# Вивід усіх записів аналітики
    all_records = analytics_mgr.get_all_records()
    logging.info("Усі записи аналітики: %s", all_records)

# Виклик головної функції, якщо скрипт запущено безпосередньо
if __name__ == "__main__":
    main()
```

Файл LiveMigrationManager.py

```
#!/usr/bin/env python3
# Доповнення до системи віртуалізації з додатковими функціями

import random
import time
import threading
import logging
import datetime
import copy

# Налаштування базового логування
logging.basicConfig(level=logging.INFO)

################################################################################
# 6. LIVE MIGRATION SYSTEM
################################################################################
class LiveMigrationManager:
# Ініціалізація менеджера для живої міграції
    def __init__(self, system):
# Збереження посилання на систему віртуалізації
        self.system = system
# Логування ініціалізації менеджера живої міграції
        logging.info("Ініціалізовано менеджер живої міграції")

# Метод для виконання живої міграції VM
    def migrate_vm(self, vm_id, destination_host):
# Перевірка, чи існує VM у системі
        if vm_id not in self.system.virtual_machines:
# Логування помилки при відсутності VM
            logging.error("VM з ID %s не знайдено для міграції", vm_id)
            return False
# Отримання VM з системи
        vm = self.system.virtual_machines[vm_id]
# Логування початку процесу міграції
        logging.info("Початок міграції VM %s на новий хост: %s", vm.name,
destination_host)
# Імітація підготовки до міграції
        time.sleep(1)
# Зупинка VM перед міграцією
        vm.stop()
# Логування передачі стану VM
        logging.info("Передача стану VM %s...", vm.name)
# Імітація передачі стану
        time.sleep(2)
# Зміна імені VM для відображення міграції
        vm.name = vm.name + f"_migrated_to_{destination_host}"
# Запуск VM після міграції
        vm.start()
# Логування успішного завершення міграції
        logging.info("VM %s успішно переміщено на хост %s", vm.name,
destination_host)
        return True

################################################################################
# 7. SECURITY & AUTHENTICATION MODULE
################################################################################
class SecurityManager:
# Ініціалізація менеджера безпеки та автентифікації
    def __init__(self):
# Словник для зберігання даних користувачів
        self.users = {}
```

```
# Логування ініціалізації менеджера безпеки
        logging.info("Ініціалізовано менеджер безпеки та автентифікації")

# Метод для реєстрації нового користувача
    def register_user(self, username, password, role="user"):
# Перевірка, чи користувач вже існує
        if username in self.users:
# Логування попередження про дублювання користувача
            logging.warning("Користувач %s вже зареєстрований", username)
            return False
# Збереження даних нового користувача
        self.users[username] = {"password": password, "role": role}
# Логування успішної реєстрації
        logging.info("Користувача %s зареєстровано з роллю %s", username, role)
        return True

# Метод для автентифікації користувача
    def login(self, username, password):
# Перевірка наявності користувача
        if username not in self.users:
# Логування помилки відсутності користувача
            logging.error("Користувач %s не знайдено", username)
            return False
# Перевірка відповідності пароля
        if self.users[username]["password"] == password:
# Логування успішного входу
            logging.info("Користувач %s успішно увійшов в систему", username)
            return True
        else:
# Логування помилки невірного пароля
            logging.error("Невірний пароль для користувача %s", username)
            return False

# Метод для перевірки прав доступу користувача до певної дії
    def check_permission(self, username, action):
# Визначення дозволених дій для ролей
        permissions = {
            "admin": ["migrate", "create_vm", "delete_vm", "configure"],
            "user": ["create_vm", "delete_vm"]
        }
# Перевірка наявності користувача в системі
        if username not in self.users:
# Логування помилки відсутності користувача
            logging.error("Користувач %s не знайдено для перевірки прав",
username)
            return False
# Отримання ролі користувача
        role = self.users[username]["role"]
# Перевірка, чи дозволено виконання дії
        if action in permissions.get(role, []):
# Логування позитивної перевірки прав
            logging.info("Користувач %s має дозвіл на дію %s", username, action)
            return True
        else:
# Логування відмови у праві доступу
            logging.warning("Користувач %s не має дозволу на дію %s", username,
action)
            return False

################################################################################
# 8. AUTO-SCALING RESOURCE MANAGEMENT
################################################################################
class AutoScaler:
```

```
# Ініціалізація AutoScaler з пороговими значеннями масштабування
    def __init__(self, system, scale_up_threshold=75, scale_down_threshold=30):
# Збереження посилання на систему віртуалізації
        self.system = system
# Порогове значення для масштабування вгору
        self.scale_up_threshold = scale_up_threshold
# Порогове значення для масштабування вниз
        self.scale_down_threshold = scale_down_threshold
# Прапорець для керування циклом масштабування
        self.scaling = False
# Логування ініціалізації AutoScaler
        logging.info("Ініціалізовано AutoScaler з порогами підвищення: %d%%,
зниження: %d%%", scale_up_threshold, scale_down_threshold)

# Метод для розрахунку середнього CPU використання серед VM
    def get_average_cpu(self):
        total_cpu = 0
        count = 0
# Ітерація по всіх VM у системі
        for vm in self.system.virtual_machines.values():
# Імітація вимірювання CPU використання
            cpu_usage = random.randint(10, 90)
            total_cpu += cpu_usage
            count += 1
# Логування CPU використання для кожної VM
            logging.info("VM %s має CPU використання: %d%%", vm.name, cpu_usage)
# Обчислення середнього CPU використання
        if count == 0:
            return 0
        avg = total_cpu / count
        logging.info("Середнє CPU використання: %.2f%%", avg)
        return avg

# Метод для автоматичного масштабування
    def auto_scale(self):
# Встановлення прапорця роботи масштабування
        self.scaling = True
        while self.scaling:
# Отримання середнього CPU використання
            avg_cpu = self.get_average_cpu()
# Якщо середнє CPU перевищує поріг, створення нової VM
            if avg_cpu > self.scale_up_threshold:
                logging.info("Середнє CPU перевищує поріг. Масштабування
вгору.")
                new_vm =
self.system.create_vm(f"AutoScaled_VM_{random.randint(1000,9999)}", cpu=2,
memory=2048, storage=50)
                new_vm.start()
# Якщо середнє CPU нижче порогу та є зайві VM, видалення однієї
            elif avg_cpu < self.scale_down_threshold and
len(self.system.virtual_machines) > 1:
                logging.info("Середнє CPU нижче порогу. Масштабування вниз.")
                vm_to_remove =
random.choice(list(self.system.virtual_machines.values()))
                self.system.delete_vm(vm_to_remove.id)
# Затримка між ітераціями
            time.sleep(5)

# Метод для зупинки процесу автоматичного масштабування
    def stop_scaling(self):
        self.scaling = False
        logging.info("Автоматичне масштабування зупинено")
```

```
################################################################################
# 9. ADVANCED NETWORK MANAGEMENT
################################################################################
class VirtualNetwork:
# Ініціалізація віртуальної мережі з CIDR блоком
    def __init__(self, network_id, cidr):
# Ідентифікатор мережі
        self.network_id = network_id
# CIDR мережі
        self.cidr = cidr
# Список підключених пристроїв
        self.connected_devices = []
# Логування створення віртуальної мережі
        logging.info("Створено віртуальну мережу %s з CIDR %s", network_id,
cidr)

# Метод для підключення пристрою до мережі
    def connect_device(self, device):
        self.connected_devices.append(device)
        logging.info("Пристрій %s підключено до мережі %s", device,
self.network_id)

# Метод для відключення пристрою від мережі
    def disconnect_device(self, device):
        if device in self.connected_devices:
            self.connected_devices.remove(device)
            logging.info("Пристрій %s відключено від мережі %s", device,
self.network_id)
        else:
            logging.warning("Пристрій %s не знайдено в мережі %s", device,
self.network_id)

# Метод для отримання списку підключених пристроїв
    def list_devices(self):
        return self.connected_devices

class NetworkManager:
# Ініціалізація менеджера мережевого управління
    def __init__(self):
# Словник для зберігання віртуальних мереж
        self.networks = {}
# Логування ініціалізації менеджера мережі
        logging.info("Ініціалізовано менеджер розширеного управління мережею")

# Метод для створення нової мережі
    def create_network(self, network_id, cidr):
        if network_id in self.networks:
            logging.warning("Мережа %s вже існує", network_id)
            return None
        network = VirtualNetwork(network_id, cidr)
        self.networks[network_id] = network
        return network

# Метод для видалення існуючої мережі
    def delete_network(self, network_id):
        if network_id in self.networks:
            del self.networks[network_id]
            logging.info("Мережу %s видалено", network_id)
            return True
        logging.error("Мережа %s не знайдена", network_id)
        return False

# Метод для підключення VM до мережі
```

```
    def connect_vm_to_network(self, vm, network_id):
        if network_id not in self.networks:
            logging.error("Мережа %s не знайдена", network_id)
            return False
        network = self.networks[network_id]
        network.connect_device(vm.name)
        logging.info("VM %s підключено до мережі %s", vm.name, network_id)
        return True

# Метод для відключення VM від мережі
    def disconnect_vm_from_network(self, vm, network_id):
        if network_id not in self.networks:
            logging.error("Мережа %s не знайдена", network_id)
            return False
        network = self.networks[network_id]
        network.disconnect_device(vm.name)
        logging.info("VM %s відключено від мережі %s", vm.name, network_id)
        return True

# Метод для отримання інформації про всі мережі
    def list_networks(self):
        networks_info = {}
        for net_id, net in self.networks.items():
            networks_info[net_id] = {"cidr": net.cidr, "devices":
net.list_devices()}
        return networks_info

################################################################################
# 10. INTEGRATION WITH CONTAINER TECHNOLOGIES
################################################################################
class Container:
# Ініціалізація контейнера з параметрами
    def __init__(self, container_id, image, cpu, memory):
# Ідентифікатор контейнера
        self.container_id = container_id
# Образ контейнера
        self.image = image
# Ресурси CPU
        self.cpu = cpu
# Ресурси пам'яті
        self.memory = memory
# Початковий статус контейнера
        self.status = "stopped"
# Час створення контейнера
        self.created_at = datetime.datetime.now()
# Логування створення контейнера
        logging.info("Створено контейнер %s з образом %s", container_id, image)

# Метод для запуску контейнера
    def start(self):
        self.status = "running"
        logging.info("Контейнер %s запущено", self.container_id)

# Метод для зупинки контейнера
    def stop(self):
        self.status = "stopped"
        logging.info("Контейнер %s зупинено", self.container_id)

# Метод для перезапуску контейнера
    def restart(self):
        self.stop()
        time.sleep(0.5)
        self.start()
```

```
        logging.info("Контейнер %s перезапущено", self.container_id)

# Метод для отримання інформації про контейнер
    def info(self):
        return {
            "container_id": self.container_id,
            "image": self.image,
            "cpu": self.cpu,
            "memory": self.memory,
            "status": self.status,
            "created_at": self.created_at.strftime("%Y-%m-%d %H:%M:%S")
        }

class ContainerManager:
# Ініціалізація менеджера контейнерних технологій
    def __init__(self):
# Словник для зберігання контейнерів
        self.containers = {}
# Логування ініціалізації менеджера контейнерів
        logging.info("Ініціалізовано менеджер контейнерних технологій")

# Метод для створення нового контейнера
    def create_container(self, image, cpu, memory):
        container_id = f"container_{random.randint(1000, 9999)}"
        container = Container(container_id, image, cpu, memory)
        self.containers[container_id] = container
        logging.info("Створено контейнер: %s", container_id)
        return container

# Метод для видалення контейнера
    def delete_container(self, container_id):
        if container_id in self.containers:
            container = self.containers.pop(container_id)
            logging.info("Видалено контейнер: %s", container_id)
            return container.info()
        logging.error("Контейнер %s не знайдено", container_id)
        return None

# Метод для отримання списку всіх контейнерів
    def list_containers(self):
        return [container.info() for container in self.containers.values()]

# Метод для запуску контейнера
    def start_container(self, container_id):
        if container_id in self.containers:
            container = self.containers[container_id]
            container.start()
            return container.info()
        logging.error("Контейнер %s не знайдено", container_id)
        return None

# Метод для зупинки контейнера
    def stop_container(self, container_id):
        if container_id in self.containers:
            container = self.containers[container_id]
            container.stop()
            return container.info()
        logging.error("Контейнер %s не знайдено", container_id)
        return None
```

```
#############################################################################
# ФУНКЦІЇ ДЛЯ ТЕСТУВАННЯ ФУНКЦІОНАЛУ
#############################################################################
def test_live_migration(system):
# Створення тестової VM для міграції
    test_vm = system.create_vm("LiveMigration_VM", cpu=2, memory=2048,
storage=50)
# Запуск VM
    test_vm.start()
    migration_manager = LiveMigrationManager(system)
# Імітація міграції на хост 'Host_B'
    migration_manager.migrate_vm(test_vm.id, "Host_B")
    time.sleep(1)

def test_security_manager():
    sec_mgr = SecurityManager()
# Реєстрація користувачів з різними ролями
    sec_mgr.register_user("admin", "admin_pass", role="admin")
    sec_mgr.register_user("user1", "user_pass", role="user")
# Спроби входу з правильними та неправильними даними
    sec_mgr.login("admin", "admin_pass")
    sec_mgr.login("user1", "wrong_pass")
# Перевірка прав доступу для різних дій
    sec_mgr.check_permission("admin", "migrate")
    sec_mgr.check_permission("user1", "migrate")

def test_auto_scaler(system):
    auto_scaler = AutoScaler(system, scale_up_threshold=50,
scale_down_threshold=20)
# Запуск автоматичного масштабування у окремому потоці
    scaling_thread = threading.Thread(target=auto_scaler.auto_scale,
daemon=True)
    scaling_thread.start()
# Запуск процесу масштабування на певний час
    time.sleep(15)
    auto_scaler.stop_scaling()

def test_network_manager(system):
    net_mgr = NetworkManager()
# Створення двох віртуальних мереж
    net1 = net_mgr.create_network("Net_A", "192.168.1.0/24")
    net2 = net_mgr.create_network("Net_B", "10.0.0.0/24")
# Підключення всіх VM з системи до мережі Net_A
    for vm in system.virtual_machines.values():
        net_mgr.connect_vm_to_network(vm, "Net_A")
# Отримання інформації про створені мережі
    networks = net_mgr.list_networks()
    logging.info("Інформація про мережі: %s", networks)
    time.sleep(1)

def test_container_manager():
    cont_mgr = ContainerManager()
# Створення кількох контейнерів з різними образами
    cont1 = cont_mgr.create_container("ubuntu:latest", cpu=1, memory=512)
    cont2 = cont_mgr.create_container("nginx:stable", cpu=1, memory=256)
# Запуск створених контейнерів
    cont_mgr.start_container(cont1.container_id)
    cont_mgr.start_container(cont2.container_id)
# Перезапуск одного з контейнерів
    cont1.restart()
# Отримання інформації про всі запущені контейнери
    containers_info = cont_mgr.list_containers()
    logging.info("Контейнери: %s", containers_info)
```

```
    time.sleep(1)

################################################################################
# ГОЛОВНА ФУНКЦІЯ ДЛЯ ТЕСТУВАННЯ ФУНКЦІОНАЛУ
################################################################################
def new_features_main():
# Ініціалізація системи віртуалізації (припускаючи, що клас VirtualizationSystem
визначено в основній системі)
    system = VirtualizationSystem()
# Створення кількох VM для тестування
    for i in range(3):
        vm = system.create_vm(f"Test_VM_{i+1}", cpu=2, memory=2048, storage=50)
        vm.start()
# Тестування живої міграції
    test_live_migration(system)
# Тестування менеджера безпеки
    test_security_manager()
# Тестування автоматичного масштабування
    test_auto_scaler(system)
# Тестування менеджера мережі
    test_network_manager(system)
# Тестування менеджера контейнерів
    test_container_manager()

if __name__ == "__main__":
    new_features_main()
```