Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи забезпечення самовідновлення програмного забезпечення в гетерогенних комп'ютерних системах.

(тема)

Виконав:
студент II курсу, групи КСМзм-19-1
Ткаленко О.В.
(прізвище, ініціали)

Спеціальність
123 – Комп’ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп’ютерні системи та мережі
(повна назва освітньої програми)

Керівник: проф. Волк М.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп’ютерна інженерія
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерні системи та мережі
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Ткаленко Ользі Вікторовні
(прізвище, ім’я, по батькові)

1. Тема роботи Методи забезпечення самовідновлення програмного забезпечення в гетерогенних комп'ютерних системах.

затверджена наказом по університету від “ 23 ” жовтня 2020 р. № 168 Стз

2. Термін подання студентом роботи до екзаменаційної комісії 14 грудня 2020 р.

3. Вхідні дані до роботи
Дослідження алгоритмів для забезпечення самовідновлення програмних додатків
Перелік можливих відмов
Методи забезпечення надійності та стійкості програмних систем

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз предметної області
2. Метод забезпечення самовідновлення розподілених програмних систем
3. Розробка алгоритмів самовідновлення розподілених програмних систем
4. Розробка програмних засобів самовідновлення розподілених програмних систем

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Вибір методики дослідження	27.10.20 – 02.11.20	
2	Огляд існуючих рішень	03.11.20 – 09.11.20	
3	Розробка структури та алгоритму	10.11.20 – 17.11.20	
4	Оформлення матеріалів атестаційної роботи	18.11.20 – 25.11.20	
	роботи		
5	Подання атестаційної роботи	25.11.20 – 02.12.20	
	керівникові та її попередній захист		
6	Подання атестаційної роботи на рецензування	03.12.20 –11.12.20	
	рецензування		

Дата видачі завдання 26 жовтня 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Волк М.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 69 с., 16 рис., 1 табл., 1 дод., 30 джерел.

ГЕТЕРОГЕННІ КОМПЬЮТЕРНІ СИСТЕМИ, ФУНКЦІОНАЛЬНА СТІЙКІСТЬ, НАДІЙНІСТЬ, САМОВІДНОВЛЕННЯ, МОДЕЛІ, МЕТОДИ, СЕРВЕР, КЛІЄНТ.

Метою работи є підвищення ефективності використання розподілених програмних систем за рахунок розробки та впровадження модифікованого мажоритарного методу самовідновлення, який зменшує час та знижує вартість відновлення. В процесі роботи було вирішено наступні задачі: досліджено існуючі методи та засоби забезпечення відновлення розподілених програмних систем; обрано мажорітарний метод самовідновлення, як метод з найменьшим часом виконання; розроблено модифікацію мажорітарного методу забезпечення самовідновлення програмного забезпечення в гетерогенних комп’ютерних системах, який знижує вартість процесу підтримки самовідновлення; розроблено програмні засоби підтримки забезпечення самовідновлення програмного забезпечення в гетерогенних комп’ютерних системах.

ABSTRACT

Master’s thesis: 69 pages16 figures, 1 tables, 1 appendices, 30 sources.

HETEROGENEOUS COMPUTER SYSTEMS, FUNCTIONAL STABILITY, RELIABILITY, SELF-HEALING, MODELS, METHODS, SERVER, CLIENT.

The aim of this work is to increase the efficiency of distributed software systems through the development and implementation of a modified majority method of self-recovery, which reduces time and reduces the cost of recovery. In the course of work the following tasks were solved: the existing methods and means of providing restoration of the distributed software systems are investigated; the majority method of self-recovery was chosen as the method with the shortest execution time; developed a modification of the majority method of software self-recovery in heterogeneous computer systems, which reduces the cost of the process of self-recovery support; software tools have been developed to support software self-recovery in heterogeneous computer systems.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

API – прикладний програмний інтерфейс (англ., Aplication Programming Interface)

DSA – алгоритм цифрового підпису (англ., Digital Signature Algorithm)

DLL – динамічно приєднувана бібліотека (англ., Dynamic-link library)

GUI – графічний інтерфейс користувача (англ., Graphic User Interface)

LAN – локальна обчислювальна мережа (англ., Local Area Network)

WS – компонента OS Windows, Windows Sockets

ВСТУП

Системи автоматичного бо автоматизованого самовідновлення сьогодні являють собою нову сферу наукових досліджень, яка вивчає стійкість до відмов інформаційних комп'ютерних систем, однією з складових яких є можливість програмних систем усувати помилки. Програмне забезпечення (ПЗ), яке має можливість виявляти та корегувати можливі збої та несправності, називається як правило програмним забезпеченням для самовідновлення. Подібна програмна система має особливі програмні компоненти, які перевіряють свої відмови, помилки, невідповідності функціонування та виправляють як найшвидше роботу усієї програмної системи. Програмні засоби самовідновлення зазвичай знають особливості поведінки комп'ютерної системи та динамічно перевіряють її відхилення реальної поведінки від очікуваної поведінки. Засоби самовідновлення в своєму составі мають елементи, які збільшують надійність обчислювальної системи. Методи самовідновлення відносяться до сфери підвищення надійної роботи комп’ютерної техніки, але мають свої характерні риси та особливості. До найбільш поширених особливостей самовідновлення відносяться:

- моделі відмов та надійного функціонування;
- засоби тестування та контролю стану програмних засобів;
- структурна та функціональна повнота системи;
- особливості проектування контексту системи.

Модель відмови або дефектів системи вказує, на те які несправності необхідно ліквідувати, охоплюючи тривалість несправності, походження несправності, помилки використання, невідповідність вимогам системи або помилки імплементації тощо.

Реакція системи включає виявлення дефектів, кількість відмов, вплив несправності на поточний стан системи та алгоритм відновлення чи спроби системи компенсувати відмови. Принципи щодо виявлення дефектів, які

закладені у системі самовідновлення, містять семантичні конструкції розпізнавання, що керуються прикладними програмами, періодичне тестування, дослідження вихідної інформації від обчислювальних ресурсів, порівняння мажоритарних елементів резервування, віддалене онлайн-тестування та інш. Повне відновлення програмної системи не завжди можливо засобами самовідновлення, бо така здатність потребує резервування додаткових обчислювальних ресурсів, наявність яких може бути обмежена. Для відповіді на відмови можливо використовувати різні методи, наприклад, маскування відмов, повторний запуск програмних компонентів, відкат або відновлення стану програми тощо.

Повнота гетерогенної комп'ютерної системи зазвичай стосується протиріччя між наявним та встановленим у майбутньому програмним забезпеченням та розробленому у минулому апаратно–програмним забезпеченням. Віна також стосується питання можливості самоаналізу системи, розвитку системи тощо. Врахування архітектурних недоліків, використання сторонніх програмно–апаратних компонентів або оновлення драйверів під час або після розгортання операційної системи є складною проблемою в проектуванні підсистем самовідновлення. Розробники програмних систем самовідновлення зазвичай потребують особливі знання про семантику інформаційних систем, що розробляються. Неповна інформація розробника про поведінку системи в ситуації відмови також є життєво важливим питанням розробки підсистеми самовідновлення. Дані від реальних систем дуже корисні особливо в умовах гетерогенних розподілених комп'ютерних систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття надійності функціонування програмного забезпечення

Поняття надійності гетерогенних комп'ютерних систем розглядається як комбінація надійності роботи апаратних компонент та програмного забезпечення. Можливо урахування факторів навколишнього середовища (довкілля), таких як джерела живлення, вплив стихійних явищ тощо.

Під надійністю функціонування програмного забезпечення (ПЗ) розуміють його здатність зберігати виконання своїх (або важливих) функції та основні (або вказані) характеристики (рисунок 1.1).

Рисунок 1.1 – Складові надійності програмного забезпечення

Складовими надійності програмного забезпечення є наступні властивості сучасних комп’ютерних систем:

- безвідмовність (проектна надійність системи);
- функціональна стійкість (при виявленні відмов підтримка виконання всіх або важливих функцій системи);

- самовідновлюваність (при виявленні відмов виконувати повне відновлення всіх компонентів на нових обчислювальних ресурсах).

Безвідмовність є особливістю програм підтримувати обчислювальний процес без програмних збоїв. Безвідмовність також може бути ймовірною та гарантувати роботу програмних застосувань протягом певного часу за наявних апаратних ресурсах без відмов.

Реалізація безвідмовності програмного забезпечення, програми або її компонентів дуже важлива для зберігання працездатності, особливо в складних інформаційних системах, при обробці великих обсягів інформації на розподілених комп'ютерних системах. Як вказано вище, безвідмовність ПЗ може бути вимірювана ймовірністю його роботи без відмов, або навпаки – ймовірністю відмови у певних умовах експлуатації та зовнішнього середовища на протязі усього часу виконання. Безвідмовність програмних компонентів також можна виміряти середнім часом між виникненням збоїв у функціонуванні програми. Апаратура комп'ютерної системи при цьому повинна знаходитися працездатному стані певний період часу.

Безвідмовність ПЗ у випадку самовідновлення повинна забезпечуватися самою програмною системою, а не бути наслідком роботи засобів операційної системи, стороннього програмного забезпечення, хмарних сервісів тощо. Необхідно розуміти, що механізми забезпечення безвідмовної роботи програмних та апаратних систем мають різну природу, специфічні властивості, отже відокремлених моделей та методів реалізації. Методи виникнення відмов в апаратних елементах і збої ПЗ дуже відрізняються та мають свої особливі властивості. Відмова апаратної частини зазвичай є наслідком руйнування апаратних елементів. Відмова ПЗ може мати апаратну природу (наприклад, ділення на 0), обумовлена невідповідністю вхідних даних, результатом помилок стороннього програмного забезпечення (драйверів, сервісів, брендмаузерів). Невідповідність встановленим нормам може виникати з різних причин: з провини розробника програми, який порушив вхідні специфікації та технічні

вимоги до програми, або навіть специфікація програмісту була надана неповна або не враховувала наявні апаратні та програмні ресурси для виконання. Часто кажуть, що коректність програми є її відповідність специфікації, але навіть специфікація може змінюватися під час проектування та налагоджування програми.

Однією з властивостей надійності ПЗ є його здатність до самовідновлення, яка характеризується інтервалами часу на виявлення та усунення відмови та виявлення причин та наслідків. Самовідновлення ПЗ зазвичай оцінюється середнім часом відновлення функціонування та усунення причин відмови. Самовідновлевання ПЗ є функцією великої кількості факторів: це й складність архітектури розподілених програм, обраної мови програмування, за допомогою якої написана програма, шаблонів програмування, якості стандартів та документації на програмне забезпечення.

Наведемо основні вимоги до системи, яка підтримує виявлення відмов та збоїв і здатна до самовідновлення :

- система має інформацію о своєї архітектурі;
- система є крос–платформною або реконфігурується згідно операційної системи та наявного середовища реального часу;
- система оптимізує свою роботу згідно наявного обчислювального середовища;
- система відстежує свої помилки та оперативно реагує на них;
- система відстежує та реагує на зовнішнє втручання та атаки;
- система має формацію о наявних обчислювальних ресурсах;
- система має засоби щодо відновлення функціонування окремих своїх компонентів.

Типовими ситуаціями виникнення програмних помилок є:

- закінчення виконання програми раніше з генерацією коду помилки;
- "зависання" (коли програма не відповідає);
- великий час виконання деякого програмного коду;

- втрата або спотворення даних, які потрібні для рішення завдання програмного забезпечення;

- пропуск вазову деяких програм з ланцюга або конвеєру, результатом чого є пропуск деяких необхідних програмних компонентів;

- спотворення даних (вхідних, вихідних, проміжних).

Якщо говорити про збільшення надійності програмного забезпечення, то воно зазвичай виконується за рахунок:

- удосконалення шаблонів, мов та технології програмування;

- вибору методів, не чутливих до збоїв та відмов обчислювального процесу (використання алгоритмічної надмірності);

- резервування програм та обчислювальних ресурсів, що реалізує апаратну та програмну надлишковість;

- моніторинг та тестування програмних компонентів з подальшим самовідновленням.

Вибір методів, не чутливих до різного роду проблем в обчислювальну процесі. Мірою чутливості проблем можуть бути помилки, викликані порушеннями середовища виконання програм. Помилки в результатах обчислень можуть бути викликані різними причинами:

- помилковими вихідними даними, які змінилися в ході обчислень, зберігання або передачі між обчислювальними вузлами;

- причинами округлення, використання невідповідних форматів, переповнення пам’яті, а також відмовами і помилками в тексті програми.

1.2 Поняття функціональної стійкості програмного забезпечення

Сьогодні відбувається бурхливе зростання складності і відповідальності сучасних інформаційних комп’ютерних систем, до функцій, які виконуються програмним забезпеченням, яке є їх невід’ємною складовою, значно підвищили вимоги до їх надійності. Щоб задовольнити ці вимоги в життєвому циклі програмного забезпечення, необхідно при проектуванні

програмного забезпечення додавати в систему спеціальні засоби, які дозволяють підтримувати виконання функцій системи. Розробники повинні мати як теоретичний так і практичний досвід в рішенні таких завдань. Під час реалізації програмного проекту, необхідно реалізовувати та застосовувати спеціальні моделі, методи, методології та технології підтримки функціональної стійкості ПЗ. Необхідно мати також інструментальні засоби виявлення та попередження відмов та збоїв. Щодо реальних систем повинні проводитися дослідження потенційних джерел відмов, факторів середовища виконання, які повинні враховуватися в засобах модернізації та оновлювання. Повинні створюватися шаблони проектування, які комплексно враховують моделі, методи та засоби забезпечення надійності програмних систем великого розміру. У кожному програмному рішенні або відокремленому проекті необхідно цілеспрямовано інтегрувати їх у програмні системи. Аналіз причин, які впливають на використання ресурсів при створенні конкретної програми, повинен дозволяти оптимізувати їх використання і реалізовувати надійність ПЗ в конкретному середовищі виконання при мінімальних або розумних витратах.

Запланована надійність програмних компонент досягається як в процесі його створення, так й під час тестування та виконання. Загальні принципи розробки об’єднують зазвичай в чотири групи:

- уникнути помилки під час проектування;
- виявити помилки під час виконання;
- виправити помилки;
- підтримка роботи в умовах помилок.

Уникнення помилок створює стандарти, специфікації для розробки безпомилкових програм, або забезпечує мінімізацію помилок, які виникають під час створення ПЗ. Виявлення помилок під час виконання зазвичай базується на механізмах переривань, які призупиняють виконання програми та виконують підпрограми обробки помилок, які зазвичай інтегровані в операційну систему. Процес тестування та моніторингу можливо відобразити

такою послідовністю: програмне забезпечення розробляється як множина формально самостійних програмних компонентів. Компоненти проектуються і проходять перевірку з урахуванням виконання усіх вимог та вимог на їх створення. Якщо під час верифікації досягається задана ступінь надійності програмних компонентів, то можна робити їх інсталяцію та випробування розподіленої програмної системи у цілому. У тому випадку, коли в процесі тестування з'явилися помилки в компонентах програмного забезпечення або в системі в цілому, то виконується повторна розробка складових частин або програмної системи в цілому. Верифікація триває до тих пір, поки не буде досягнуто запланованого показника надійності.

Для того, щоб підвищувати надійність програмних систем необхідно проводити стандартизацію процесів розробки програм та їх взаємодії, приділяючи увагу і супроводу програм. У шаблонах програмування використовується досвід і результати роботи великої кількості фахівців і розроблюються рекомендації з використання найбільш ефективних сучасних моделей, методів та технологій програмування. Як наслідок відповідних узагальнень відпрацьовуються процеси розробки та методи розробки, а також методична основа для їх автоматизації. Стандарти на розробку програмних систем можуть використовуватися як специфікації, керівні або як рекомендаційні документи, а також як офіційна база при розробці засобів автоматизації технологічних етапів або проектів.

На початковому етапі розробки слід формувати повний комплект документації - план, що забезпечує регламентування для розробника усіх етапів і робіт при створенні надійних програмних систем. Для виконання цього плану та його окремих положень повинні бути обрані відповідні засоби, які спільно з операційною системою, утворюють взаємопов'язаний процес технологічної підтримки і автоматизації та у відповідь на наявні вимоги не суперечать документації розробника та розробленому плану виконання проектувальних робіт в відповідних умовах розподіленого програмного середовища.

Загалом, система підтримки обчислювального процесу паралельно з проведенням автономних обчислень повинна реалізовувати функції оптимізації, самозахисту та самовідновлення (рисунок 1.2).

Рисунок 1.2 – Система підтримки функціональної стійкості програмних систем

Для пошуку та усуненню помилок, які виниклі на етапах проектування програмної системи, усі етапи проектування та супроводу програм повинні забезпечуватися методами і засобами систематичної, автоматизованої верифікації.

З надійністю пов’язують сукупність наступних особливостей:

- цілісність програмного компонента (здатність його виконувати захист від відмов та помилок);

- живучість (здатність до перевірки обробки даних та їх трансформацій в ході функціонування програмної системи);

- завершеність (програмна система виконує тестування усіх компонентів системи);

- працездатність (властивість програмних компонент, системи, або середовища виконання відновлюватися після відмов).

Особливість поняття надійності програм у сенсі підтримки функціональної стійкості полягає в наступному: надійність програми забезпечуються не тільки самою програмою, але й середовищем виконання (апаратними засобами, операційною системою та інше). При детальному аналізі надійності програми, зазвичай допускають частку помилок в ній та вимірюють імовірність їх появи.

1.3 Поняття живучості програмного забезпечення

Живучість програмних систем є властивістю протистояти потоку критичних відмов та збоїв під час функціонування, включаючи процеси інсталяції, обслуговування та ремонту або властивість об'єкта виконувати повністю або в рамках обмежень свої функції при впливах, не передбачених заздалегідь розробниками та адміністраторами. Можливо додати здатність адаптуватися до нових умов, які змінюються постійно, нестандартних ситуацій, протистояти негативним впливам, виконуючи при цьому свою цільову функцію різними шляхами, наприклад за рахунок відповідної зміни структури, елементів, зв’язків та поведінки системи.

Говорячи при живучість, необхідно відмітити, що це комплексне поняття, що об’єднує множину дисциплін різних напрямків. Живучість зазвичай забезпечується на заданому рівні функціонування, як, наприклад, надійність системи або її відмово стійкість. Моделі, методи забезпечення живучості створюють технічні засоби, які виконують підтримку функціонування в непередбачених ситуаціях. Це надає додаткової складності в забезпеченні властивості живучості для сучасних програмних систем, що працюють в умовах гетерогенності комп’ютерних систем.

Надання властивостей самовідновлення можливо в ситуаціях, які можна об’єднати в два науково−проектувальних напрямки, що базуються на

агентах (програмних компонентах, які функціонують на об’єктно-орієнтованному програмуванню та взаємодії з операційною системою) або програмні системи самі забезпечують самоорганізацію з реалізацію самовідновлення.

1.4 Поняття збоїв та відмов

Головний підхід то понять збоїв і відмов в програмних системах – це розподіл по інтервалам часу тривалості самовідновлення після будь-якої зупинки з виконання програми, помилок даних або порушення обчислювального процесу, що фіксується як збій працездатності. У випадку, коли тривалість відновлення після відмови менше заданого апріорі порогу – помилки при функціонуванні програмної системи відносять до збоїв. У тому випадку, коли час, який було витрачено на самовідновлення, перевищує по тривалості лімітоване значення – таку аварію характеризують як відмову. З ціллю класифікації програмні збої і відмови на основі тривалості відновлення, виконується моніторинг динамічних характеристик програм, які потребують потоки даних, а також часові властивості функціонування окремих компонент програмних систем. Лімітований час відновлення працездатного стану програмного забезпечення, при перевищенні якого слід заносити відмову до журналу та реагувати на нього, зазвичай наближається до періоду вирішення задачі та видачі з боку інформаційної системи інформації відповідному користувачеві.

Одним з крокім до забезпечення функціональної стійкості є автоматичне або автоматизоване розпізнавання програмних збоїв. Стандарт IEEE 1044[3] фіксує повний список категорій та класифікацій програмних артефактів. Перелік програмних збоїв, відмов та помилок багато обговорювалися для в умовах різних типів програмних систем. Зазвичай основні параметри та визначення стосуються програмного забезпечення. Відмова елементів програмного забезпечення є наявним дефектом в системі.

Помилка програми є невідповідністю між очікуванням та реальною поведінкою програмною системи. Відмова програмного забезпечення зазвичай виникає, коли інформація не обробляється коректно або відхилення поведінки системи від плану робіт. Відмова або збій програмної системи не обов'язково може спричинити некоректне виконання програмного забезпечення[4].

Наведемо основні види несправностей програмних систем, які зустрічаються в літературі для різних видів розподілених програмних систем [5][6]:

- логічні помилки: нелогічна поведінка та не очікуванні результати;
- синтаксичні помилки: несправності інтерфейсу та конфігураційних параметрів програмної системи;
- відмова сервісу: невідповідність QoS, невідповідності, які викликані угодами щодо захисту функцій (sla), та порушень в режимі реального часу;
- відмови в взаємодії: тайм-аут та відмова в наданні послуг;
- винятки введення/виводу, та винятки, пов'язані із функціями безпеки.

Синтаксичні помилки виникають у випадках, коли повідомлення та параметри користувача не узгоджені з специфікацією замовника. І хоча компілятори зазвичай фіксують подібні види несправностей, які виникають в програмі. Такі помилки можуть статися у цільових програмних системах та веб-додатках, де клієнт і сервер можуть працювати незалежно один від іншого. Семантичні помилки представляють помилки та відмінності з запланованим проектом та наданою специфікацією. Семантичні помилки зазвичай є складнішими для діагностики та потребують спеціальних засобів для їх виявлення.

Сервіс починає відмовляти, коли продуктивність наданої послуги не відповідає заданим обмеженням, які зазначені у якості обслуговування (QoS) або відповідної угоді про рівень обслуговування (SLA). Ці види відмов призводять до погіршення характеристик к працездатності системи. Відмови

також виникають за рахунок некоректного зв’язку та взаємодії між програмними компонентами, службами, сервісами та підсистемами як в розподіленій, так і в централізованій системі. Відсутність та безвідповідальна політика хмарних сервісів та терміни виконання запитів на послуги є прикладами помилок такого спілкування та такої взаємодії.

Системні винятки є джерелом відмов програмного забезпечення. Вони часто призводять до екстремального припинення виконання програми. Зазвичай винятки, які реалізовані на мовах, таких як C++ та Java, забезпечують як стандартні, так і визначені розробником винятки. Однак вони не забезпечують надійного засобу запобігання невідомого припинення заявки під час некоректного виконання винятку. Різні рівні багатозадачності у ситуації винятку є спеціальною обробкою ситуації з відмовами, та винятки щодо відновлення в случає конкретних та наявних помилок програмної системи.

1.5 Приклади сервісів відновлення DRaaS

DRaaS (disaster recovery as a service) є сервісом, який виконує відновлення після відмов. Відмови у цієї концепції обумовлені спектром форс– мажорних ситуацій, до яких відносяться аварії, техногенні надзвичайні ситуації, природні стихійні явища та несанкціоновані дії кваліфікованих та некваліфікованих користувачів, адміністраторів, керівників та інших представників персоналу та можливих джерел впливу. Він презентує сервіс, який надає можливість хостінгу віртуальних або фізичних серверів, що функціонують для забезпечення відновлення у випадках техногенної катастрофи (рисунок 1.3).

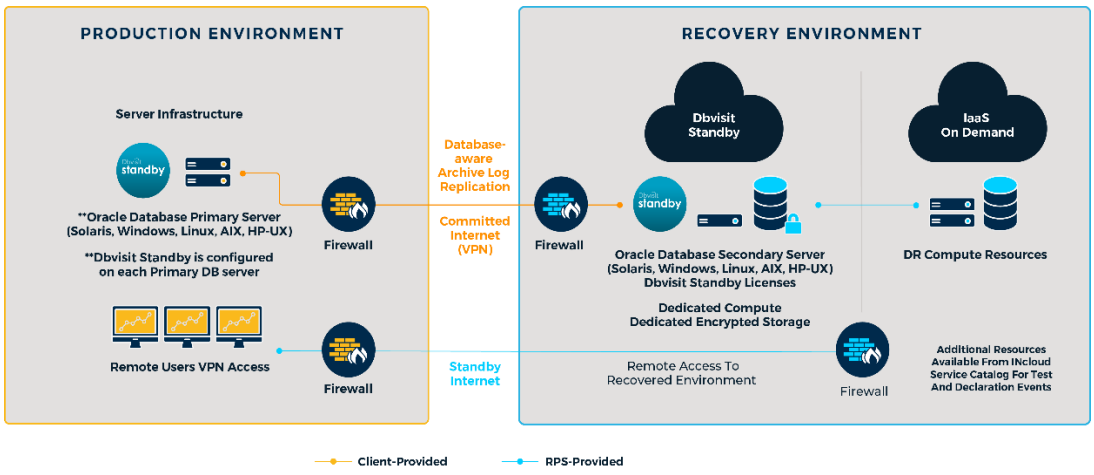

Рисунок 1.3 – Основні сервіси самовідновлення DRaaS

DRaaS зазвичай використовується фірмами, яким не вистачає кваліфікації та досвіду щодо забезпечення, виконання та верифікації порядку дій самовідновлення після відмов та збоїв.

Багато фірм стикаються з різними формами відмов та збоїв. На приклад: віялове відключення електроенергії, апаратні збої, несанкціонована зміна файлів, людський фактор, землетрус або цунамі, повені, урагани та грози. Ці явища можуть вразити центри обробки даних. Ймовірно, через це можуть відбуватися великі перебої, наприклад, вихід з ладу сервера веб-сайту, втрата інформації згідно виконує мого обчислювального процесу або клієнтських та серверних даних. Як свідчать дослідження Інституту Понемона «Вартість недоліків центрів обробки даних», в яких вказано , що незаплановані простої апаратних комплексів, які є слідством відмов та збоїв, коштують організаціям в середньому 8,850 доларів за хвилину.

DRaaS зосереджується на відновленні в умовах реального часу. Це гарантує відновлення даних максимально наближено до поточного стану. Зазвичай на таке відновлення витрачається близько 4 годин, але клієнт націй час отримує нові машини, які географічно розташовані у віддаленому місці.

1.6 Огляд існуючих рішень

На даний час існують програмні системи, які налають функціональні можливості відновлення після збоїв та відмов, або зникнення частини даних (рисунок 1.4).

Рисунок 1.4 – Найпопулярніші сервіси аварійного відновлення після збоїв

Ряд фірм надають сервіси резервного копіювання даних, які використ овюються для синхронізації з програмними системами, щоб запобігти втраті інформації. Водночас дані зазвичай можуть бути розподілені між різними програмними системами, застосуваннями, центрами зберегання даних та хмарними сервісами. Тому необхідно підтримувати постійність роботи у випадку відмови чи збою не лише даних, але й самих програм.

В даній роботі були проаналізовані можливості найпоширеніших сервісів для автоматичного відновлення, таких як:

- Zerto IT Resilience Platform;
- Microsoft Azure Site Recovery;
- Arcserve UDP Cloud Direct;
- VMware Site Recovery Manager;
- Plan B Disaster Recovery.

Zerto IT Resilience Platform (рисунок 1.5) створювалась для захисту даних та обчислювальних ресурсів, спрощує відновлення після збоїв та зменшує витрати на їх зберігання. Розробка визначається як рішення

відновлення після збоїв, яке забезпечує автоматизацію процесу відновлення цілком та виконує цю роботу незалежно від сховища даних.

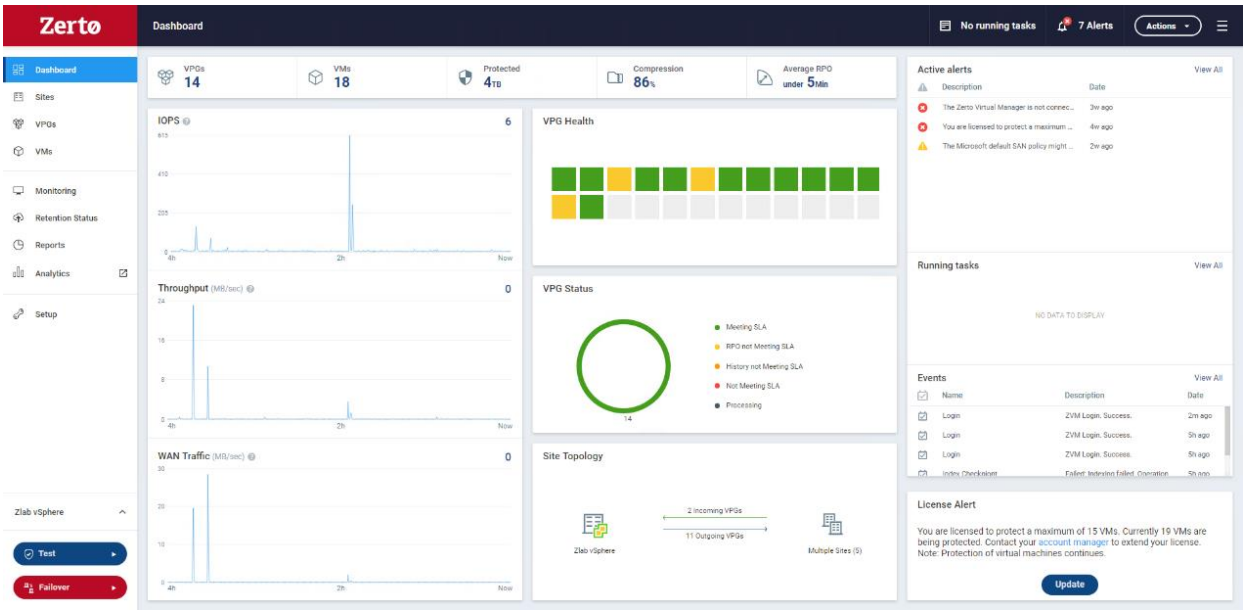

Рисунок 1.5 – Графічний інтерфейс користувача Zerto IT Resilience Platform

Продукт постачається з множиної інтегрованих опцій. Zerto має інтерфейси з відомими сервісами Microsoft Hyper-V та VMware vSphere. Рішення буде захищати мультимедійні віртуальні машини відомих баз даних, таких як Exchange, Oracle та SQL та файлові сервера. Вона також дозволяє переносити робочі навантаження та набори даних між корпоративними, приватними та гібридними хмарами.

Microsoft Azure Site Recovery допомагає захистити застосування реального часу виконання, відновлення роботи сайтів, підтримує бізнесові додатки та перерозподіляє навантаження під час несподіваних відключень.

Сервіси дозволяють автоматично копіювати IT-інфраструктуру, яка базується на обраній політиці, та може використовуватися великими корпораціями. Може виконувати захист Hyper-V, VMware та віддалені сервери, а також використовувати інші сервіси Microsoft Azure або сторонні дата центри для свого відновлення.

Засоби відновлення сайтів Azure інтегруються з SQL Server AlwaysOn та System Center. Засоби зв’язку в Azure підтримують шифрування, надають можливості розробки складних сценаріїв відновлення на віртуальному сервері Azure. На рисунку 1.6 наведено приклад інтерфейсу користувача Microsoft Azure Site Recovery.

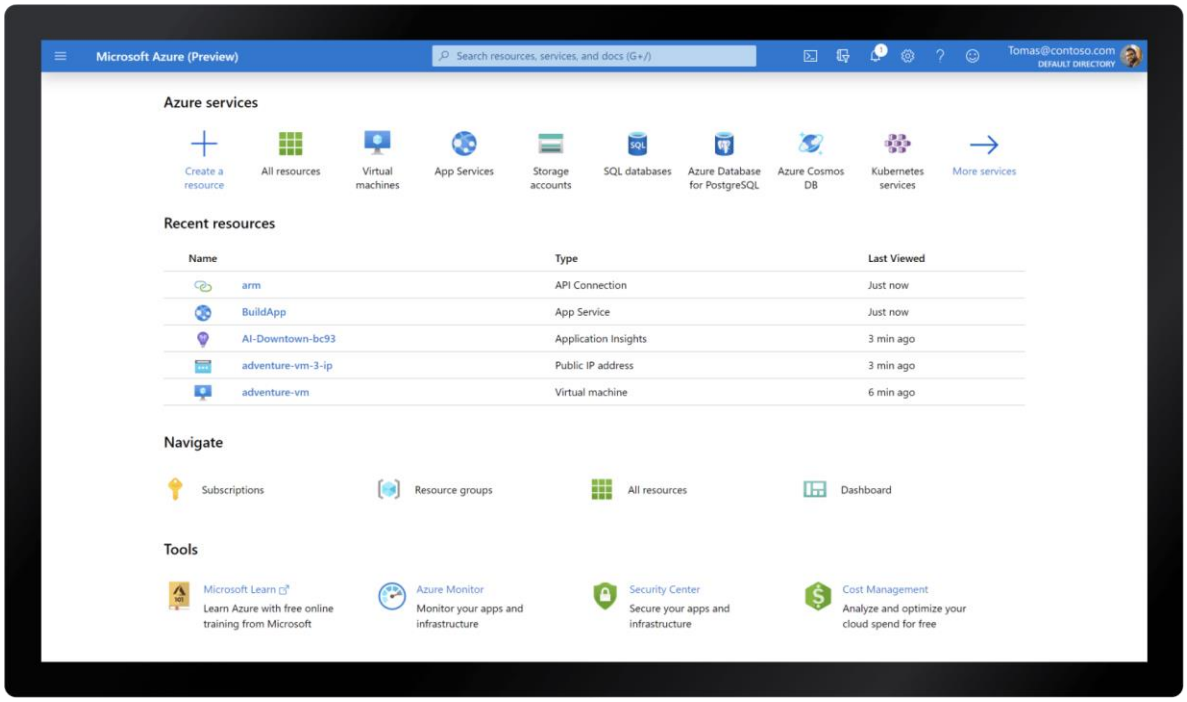

Рисунок 1.6 – Графічний інтерфейс Microsoft Azure Site Recovery

Під час роботи, ця система, для забезпечення відновлення сайту, копіює процеси, які працюють на фізичних та віртуальних машинах (VM) з оригінального сайту в інше місце. У випадку, коли на оригінальному сайті відбувається збої або відмови, всі процеси, запити, потоки даних перемикаються на інший сервер. Служба резервування Azure зберігає дані, створюючи резервну копію у хмарі.

Ще одне рішення для фірм та компаній – це VMware Site Recovery Manager. Воно забезпечує безвідмовну роботу та відновлення після збоїв. Цей пакет програм запускає віртуальні машини та робить копії захищених сайтів на базі серверів vCenter. Використовується VMware Site Recovery

Manager зазвичай для реалізації різних видів відновлення.

Дана платформа реалізує міграцію віртуальних машин та сайтів на віддалені комп'ютерні ресурси. Планова міграція може зберегти надійність функціонування програмних систем при динамічної зміні робочих навантажень. На момент міграції два сайти повинні повністю функціонувати і повторювати друг друга. При відмові нема потреби у запуску обох сайтів, але коли один з сайтів відмовляє, система автоматично розгорне його на іншому ресурсі, де він зберігався. Коли відбувається збій, система інформує власника або технічний персонал о ситуації, яка сталася. Система управління процесом відновлення сайтів відображає процес відновлення за допомогою графічного інтерфейсу (рисунок 1.7), та використовує механізми реплікації для мінімізації зникнення даних та часу простою програмної системи.

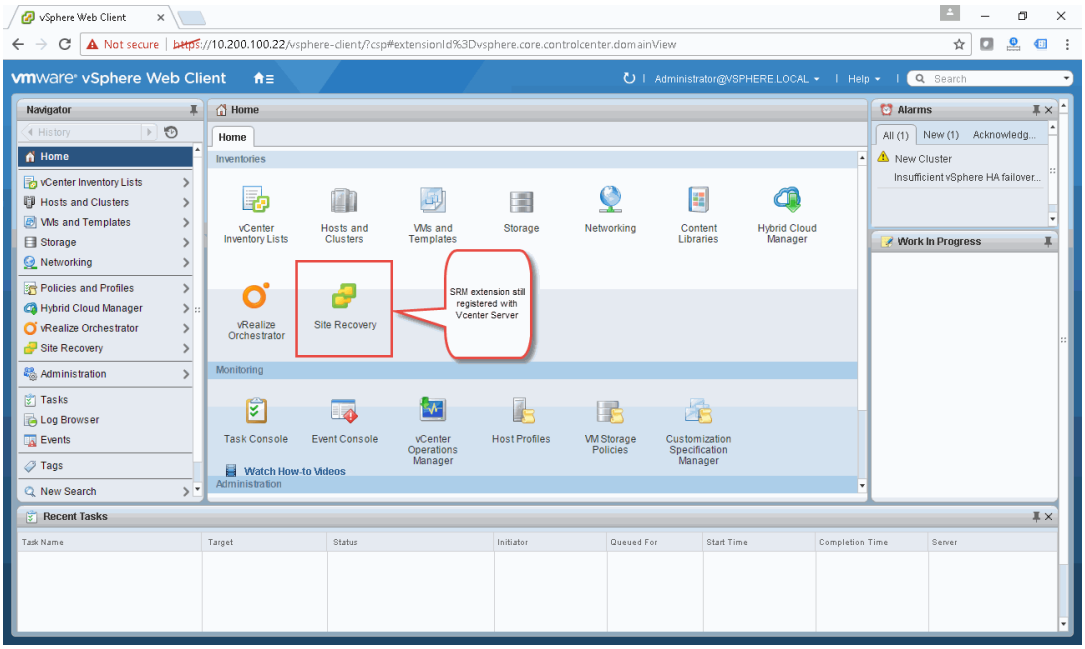

Рисунок 1.7 –Графічний інтерфейс користувача
VMware Site Recovery Manager

Множену послуг з відновлення після відмов надає Plan B Disaster Recovery Services. Система розроблялась з вимогами сбалансувати елементи архітектури системи та бюджету.

Сервіси та служби Plan B поєднують віртуальні технології резервування, підтримку сучасних мережний рішень, з мультірівневими параметрами керування. Користувачі можуть створювати свої додаткові компоненти для відновлення та інтегрувати їх у сервіси Plan B. Тому ця платформа підходить для розробки індивідуальних рішень як у приватного так і у корпаративного сегментів ринку (рисунок 1.8).

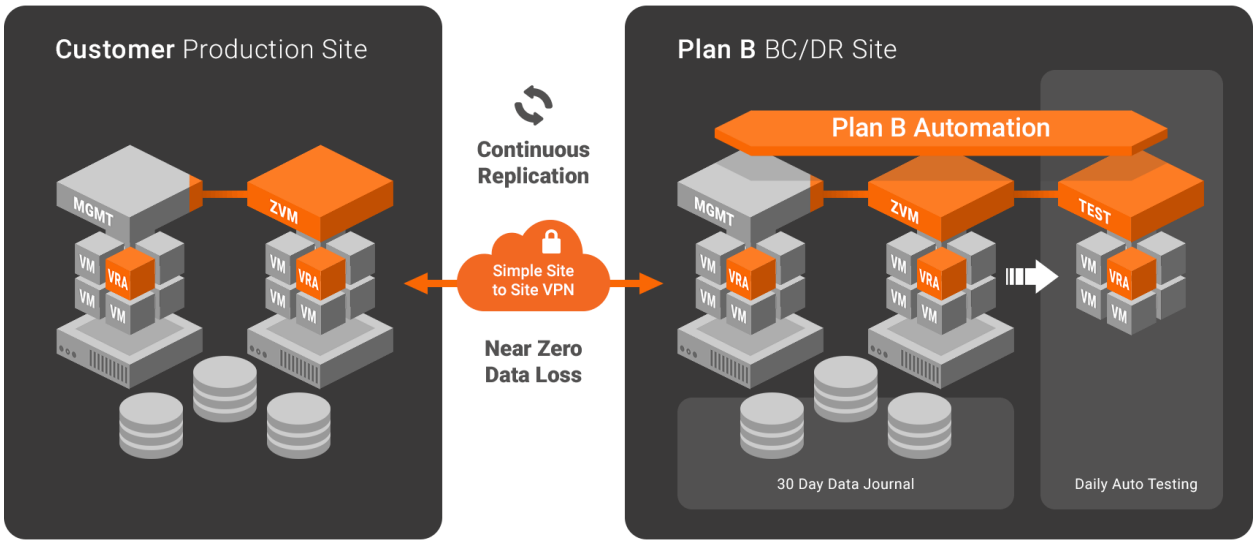

Рисунок 1.8 – Служби та сервіси Plan B Disaster Recovery Services

Відмінністю сервісів Plan B є відкритий доступ до коду, тому користувачі можуть додавати свої власні функції, організуя свої сценарії, сховища та бази даних, канали передачі інформаціїї тощо.

Disaster Recovery періодично виконує копію даних і стану систем та передає це у віртуальне середовище, де їх можуть використовувати користувачі після збоїв. Спеціальні системи постійно тестують усі зарегістровані компоненти системи та ведуть журналізацію їх станів.

Розробники ще однієї системи Arcserve UDP Cloud Direct (рисунок 1.9) розробили систему відновлення з функціями захисту даних на відповідному рівні, який потрибує фірма, корпорація або приватній кори була створена для

того, щоб забезпечити ефективність та безпеку, яку потребує підприємство. стувач. Якщо програне забезпечення зазнає збою, користувач має доступ до більшості програмних компонент, баз даних і продовжує працювати. Водночас усі дані та потоки даних шифруються та використовують протоколи безпеки.

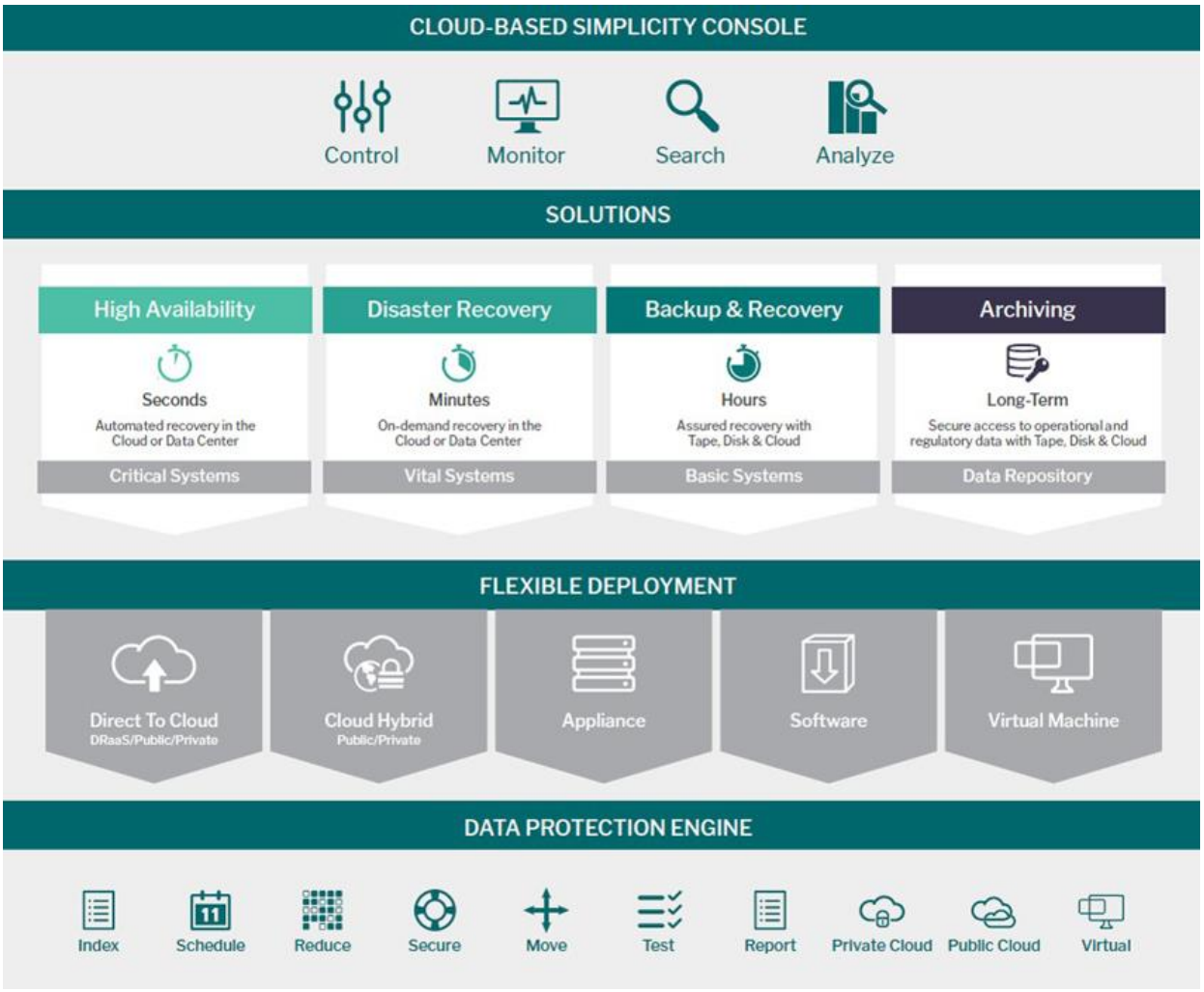

Рисунок 1.9 – Структура пакетів Arcserve UDP Cloud Direct

На підставі проведеного аналізу зробимо порівняльнний аналіз з використанням наступних характеристик часу відновлення, кількості можливих користувачів, складності користування, вартості, доступності до свободного використання та інше (таблиця 1.1).

Таблиця 1.1 – Порівняльна таблиця існуючих рішень

Продукт	Ціна	Кількість користувачів	Час відновлення	Free Trial	Складність користування	Підтримка спеціалістів
Microsoft Azure Site Recovery	25$ на місяць	~ 100.000	4 години	31 день	Необхідні спеціальні навики	Так
Zerto IT Resilience Platform	$745 на рік	> 7.000	24 години	14 днів	Необхідні спеціальні навики	Так
Arcserve UDP Cloud Direct	Від $10 на місяць	> 48.000 клієнтів	12 годин	Немає (30-днів гарантія повернення коштів)	Простий та ефективний в використанні	Ні
Plan B Disaster Recovery	$20 на місяць	No information	До 24 годин	Немає	Простий та ефективний в використанні	Так
VMware Site Recovery Manager	від $1200 на рік	~ 20.000	No information	30 днів	Необхідні спеціальні навики	Так

1.7 Постановка задачі

Аналіз готових рішень, що представлено у світі говорить о том, що є багато продуктів, які підтримують резервування та автоматичне відновлення після збоїв. Але основний недолік – це час відновлення, який за інформацією з таблиці 1.1 становить від 4 до 24 годин. Тому є необхідність пошуку інших методів відновлення, які будуть гарантувати відновлення функціонування в коротші терміни.

Одним з напрямків забезпечення надійності розподілених комп'ютерних систем є самовідновлення (self-healing) програмних компонентів [10]. Серед існуючих методів реалізації найбільш ефективним з боку мінімізації часу самовідновлення є мажоритарний метод, який

базується на паралельному виконанні на віддалених обчислювальних ресурсах однакових програмних компонентів. У нормальному стані усі компоненти видають однакові резуль-тати обчислень. Якщо один з компонентів відмовляє, результат обчислень прийметься на основі працюючих, а система підтримки самовідновлення шукає новий обчислювальний ресурс для паралельного запуску ще одного екземпляра програм-ного компонента.

Для великих складних систем може існувати багато різних типів несправностей, і їхня різна природа часто вимагає розрізнених, підібраних підходів для їхнього виявлення та виправлення. Використання зазначеного мажоритарного методу зазвичай характеризується значними витратами. Тому є потреба в розробці модифікованих методів, які, за умови виконання цілей та термінів самовідновлення, дозволять знизити вартість використання надлишкових комп'ютерних ресурсів.

Метою роботи є підвищення ефективності використання розподілених програмних систем шляхом розробки та впровадження методів самовідновлення, які зменшують час та знижують вартість відновлення.

Самими швидкими методами самовідновлення вважаються методи, які реалізують принципи мажоритарного резервування. Він передбачає надлишковість, коли паралельно запущені декілька екземплярів однієї програми. Система управління постійно виконує моніторинг стану програм, та коли один з екземплярів відмовляє, результат роботи програмної системи приймається з тих програм, що залишилися, а система управління знаходить новий обчислювальний ресурс, передає на нього екземпляр програми, запускає її та відновлює стан програми за допомогою «живого» екземпляру або збереженого стану.

Згідно встановленої мети атестаційної роботи необхідно вирішити наступні задачі:

- дослідити існуючі методи та засоби забезпечення відновлення розподілених програмних систем.

- вибрати метод самовідновлення з меньшим часом виконання.

- розробити модифікацію методу забезпечення самовідновлення програмного забезпечення в гетерогенних комп’ютерних системах, який знижує вартість процесу підтримки самовідновлення.

- розробити програмні засоби підтримки забезпечення самовідновлення програмного забезпечення в гетерогенних комп’ютерних системах.

2 МЕТОД ЗАБЕЗПЕЧЕННЯ САМОВІДНОВЛЕННЯ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

2.1 Вибір методу самовідновлення розподілених програмних систем

Методи самовідновлення можна розглядати на трьох рівнях, які заходяться в залежності від комп'ютерного ресурсу, який відстежується і який ми повинні відновити: програмні компоненти, сервіси операційних систем; Також вплив надають апаратні ресурси та локальні програмні компоненти [1-4].

Самовідновлення на рівні програмного компонента зазвичай розглядається як здатність самої програмної системи або сервісу відновити роботу самостійно, з залученням тільки стандартних засобів середовища виконання. [13,14]. Програмісти фіксують помилки та усякі внештатні ситуації за допомогою винятків, при цьому можливо вказати конкретну функцію, яка автоматично викликає обробітку збоїв та відмов. У випадку виникнення виключення програміст може реалізувати різні реакції: ігнорувати та повернутися до виконання основного алгоритму, зупинити застосування, розробити складні алгоритми відновлення, які спробують відновити програму. У нашому випадку ми будемо використовувати один з найбільш ефективних способів рішення проблеми – а саме – шаблон проектування програмних компонент, який реалізує створювати застосування з внутрішнім самовідновленням [6].

Самовідновлення може реалізовуватися як на системному рівні, так і нарівні самої програмної системи або окремого програмного компонента з застосуванням можливих та доступних службових сервісів з операційної системи або сторонніх застосувань з відкритим інтерфейсом внутрішніх елементів. Деякі види самовідновлення можливо реалізувати на рівні усієї гетерогенної комп'ютерної системи. Прикладом методу, який реалізує

відомий додаток Microsoft Word у випадку збою – перезапуск додатку з завантаженням файлу бекапа.

У випадку відмови обладнання, самовідновлення виконує пошук вільних комп'ютерних ресурсів, які можуть прийняти програмний компонент з особливими характеристиками, та налагодити взаємозв'язок між оновленим програмним компонентом та іншими компонентами програмної системи. Як і у випадку операційної системи, треба періодично перевіряти та зберігати стан програмних компонент [4,6].

Усі методи самовідновлення основані на надмірності. Так збереження стану потребує значних обсягів пам'яті, передача стану – мережних ресурсів, слідкування за програмними компонентами, моніторинг та відновлення – процесорного часу. Тому вартість підтримки самовідновлення зазвичай велика. В роботі поставлена мета розробки та впровадження метода самовідновлення дозволяє зменшити час самовідновлення, але водночас не буде значно збільшувати вартість самовідновлення.

Методом, який найшвидше відновлює роботу системи, а іноді й не зупиняє обчислювальний процес, є мажоритарний метод, основна ідея якого полягає в паралельному виконанні одного програмного компонента на різних віддалених ресурсах. Під час відмови одного з них, інший (інші) аналогічні програмні елементи продовжують роботу, а система підтримки відновлення виконують пошук нового ресурсу, завантаження на нього екземпляра програмного компонента, його запуск та відновлення стану на останній збережений або робить копію функціонуючого програмного компонента.

2.2 Процес забезпечення самовідновлення

Формалізуємо процес самовідновлення програмної системи в умовах гетерогенної комп'ютерної системи.

Під самовідновленням будемо розуміти здатність програмної системи продовжувати виконання свої функцій у гетерогенному комп'ютерному

середовищі при відмові або збою одного (декількох) комп'ютерного ресурсу або втрати мережного зв'язку між програмними компонентами системи.

Підтримка мережних каналів та відновлення їх стану під час збоїв, зазвичай виконується мережними сервісами або сервісами операційної системи.

За умов відмови одного з комп'ютерів для забезпечення самовідновлення, необхідно зробити запуск програмного компоненту, який відслідковувався на окремому ресурсі, на другому обчислювачі (комп'ютері) в заданий час. Програмний компонент у цьому випадку повинен бути введено у такий стан, в якому програмний компонент був під час збою, а, якщо це можливо, у ближчій стан, який було збережено на сервері або в розподіленій базі даних в минулому.

Програмний компонент P є елементом програмної системи, який складається з дох архітектурних елементів (2.1): інструкціям програми (які виконують поведінку і є виконуються процесором або віртуальною машиною) (code), та даних (data), що є відображенням стану програмного компонента у часі. Можливо виділити в програмі дві відповідні пари $\text{code} \Leftrightarrow p^L$ і $\text{data} \Leftrightarrow p^D$ які система управління може використовувати щодо керування процесом самовідновлення програмного компонента. Іноді процес зберігання цих компонент називають процесом журналізації (запис в електронний журнал даних про стан програмного компонента в теперішній момент часу). Аналогічний процес було сформульовано [ ] для випадку збереження стану частих програмних моделей у розподіленому імітаційному моделюванні складних інформаційних систем.

Перехід програмного компоненту з одного стану в інший відображається на програмному рівні зміною даних: $\text{data} \xrightarrow{\text{code}} \text{data}'$. Ця зміна проходить в окремі моменти час у T_q, де q уявляє собою натуральне число та зазвичай має атрибут системного часу. Реалізація визначення цього атрибуту може носить апаратний або програмний характер.

В джерелах зустрічаються два найпоширеніших способи збереження стану програми – дамп пам’яті та журналізація істрії змін локальних та глобальних змінних програмного компоненту [29].

1. Дамп пам’яті програмного компонента. Алгоритм цього способу надає надійні механізми, які дозволяють системі керування обчислювальним процесом відновлювати програмні компоненти в одну з часових точок в минулому. Але для цієї точки повинен бути зроблений дамп пам’яті. Введемо для реалізації цієї функціональності підпрограму $P_{dump}(t_q)$, яка дозволить системі самовідновлення запам’ятати дані програмного компоненту в момент часу t_q, передати їх на сервер та зберегти у базі даних.

2. Журналізація зміни пам’яті програмного компоненту. У випадках, коли змінюються дані, система управління самовідновленням здійснює запис зміни даних та атрибутів, які дані та в який час змінювались. Таким чином система управління утворює ланцюжок змін, який передається на сервер та записується у базу даних. Аналогічно попередньому випадку, введемо процедуру або метод, який реалізує цей процес – $P_{change}(data^{adr,sz}, t_q)$, де атрибути adr і sz характеризують адресу даних, що змінювалися і відповідно обсяг даних, які змінювалися по даному адресу.

Кожній з цих функцій системи керування обчислювальним процесом необхідно поставити у відповідність зворотні функції, які, на основі збережених даних повертають програмні компоненти у попередній стан: $P'_{dump}(t_q)$ і $P'_{change}(data^{adr,sz}\ t_q)$. Зворотна функція P’ виконує дію $data' \xrightarrow{code'} data$, повертаючи дані програми (а значить і сам програмний компонент) в відповідний попередній стан. Так як час в системі змінюється дискретно, система управління має можливість повертати програмні компоненти в відповідні моменти у минулому.

При розробці системи самовідновлення можну обрати один з цих двох способів, або реалізувати усі з них. Кожен з них має певні недоліки та переваги, тому доцільно розробити алгоритми динамічного вибору одного з них або використовувати їх одночасно.

Факт необхідності вибору та використання обох способів підкреслюється різними ресурсними затратами, які зазвичай дуже різні для різних програмних компонент.

У випадку першого способу зазвичай спостерігається використання значних часових та ресурсних витрат, що обумовлено значним часом використання процесорних елементів та значним часом копіювання даних. У цьому випадку частіше спостерігають лінійно зростаючий обсяг пам’яті, щоб зберегти історію змін станів програмних компонент з найменшим інтервалом дискретності часу виконання.

При використанні другого способу обсяг збережених даних зазвичай менше (але не завжди). Але є й недолік – це збільшене використання процесорного часу, особливо під час повертання програмного компонента у минуле, коли необхідно прослідити в минуле увесь ланцюжок змін даних.

Як вже сказано, оптимальним є створення такої системи управління самовідновленням, яка буде підтримувати обидва способи, а додавання інтелектуальної підсистеми динамічної зміни підпрограми управління даними є найбільш перспективною.

Формалізуємо описаний процес, для чого введемо функцію, яка доречи є у стандарті мови C++ – sizeof (data), яка розрахує об’єм пам’яті (V, байт), що потрібна для розміщення data і функцію time (P), яка розраховує очікуваний час виконання циклу програми (t, секунд). Тоді sizeof (data, tq) значить, що паралельно з даними треба занести в базу даних й атрибут часу, коли відбулася зміна даних.

Нижче наведемо вирази, що можуть враховувати час відтворення алгоритмів керування програмними компонентами у різних умовах та станах середовища виконання.

У випадку, коли усі програмні компоненти виконують запис свого стану, об’єм пам’яті, що необхідно надати, буде дорівнювати об’єму пам’яті, що треба надати для збереження стану усіх програм розподіленої системи, за умови, що локальні та глобальні дані підпрограм не перетинаються:

$$\begin{array}{l}\sum_{i=1}^{N} \text{sizeof}(\text{data}_i) = \text{sizeof}(\text{data}), \\ \text{якщо } \text{data}_i \cap \text{data}_j = \varnothing \ \ \forall\, i, j \in \overline{1, N}, i \neq j\end{array} . \qquad (2.1)$$

У деяких випадках рекомендується зробити дампи пам’яті всієї програми завдання і зберегти зміну стану всієї розподіленої програмної системи з збереженням атрибутів часу.

У випадку, коли окремі програмні компоненти виконали кілька збережень пам’яті, об’єм загальної пам’яті збільшується:

$$\begin{array}{l}\sum_{i=1}^{N} \text{sizeof}(\text{data}_i) > \text{sizeof}(\text{data}), \\ \text{якщо } \exists\, \text{data}_i \cap \text{data}_j \neq \varnothing, \ \ \forall\, i, j \in \overline{1, N}, i = j\end{array} . \qquad (2.2)$$

Якщо стає питання вибору методу збереження стану, то треба враховувати не тільки об’єми пам’яті, а час, який витрачається на виконання збереження даних або навпаки відновлення стану програмного компонента. Так, коли

$$\text{time}(P_{\text{damp}}) >> \text{time}(P_{\text{change}}(\text{data}^{\text{adr,sz}}, t_q)), \qquad (2.3)$$

потрібно обрати інший метод зберігання стану. Водночас з цим, у випадку значної кількості викликів підпрограми P_{change}, об’єм пам’яті зростає лінійно, при цьому об’єм пам’яті, який треба для виконання P_{dump}, відомий, а відповідна функція зазвичай швидко реалізується засобами стандартних бібліотек або сервісами операційної системи. Вираз (2.1) говорить о том, що розрахунковий час, який витрачається програмною системою на дамп пам’яті загалом більше часу, що необхідно для збереження стану підпрограми.

Виходячи з цього факту, в доцільно проводити дослідження моделей та методів, які б дозволили змінювати динамічно методи та процедури збереження стану програм.

2.3 Розробка мажоритарного методу самовідновлення програмного забезпечення

Існують критичні застосування, які ставлять збільшені вимоги стосовно забезпечення самовідновлення. Наприклад, процеси, що виконують аналіз та обробку потоків інформації та потребують прийняття швидких рішень, (це й космічні дослідження, військові додатки, передбачення надзвичайних ситуацій, тощо). Часто для таких застосувань використовується мажоритарний метод самовідновлення розподіленого обчислювального процесу.

В основі цього методу лежить ідея запуску кожного програмного компонента не один раз, а кілька разів на різних (віддалених друг від друга комп’ютерних ресурсах). Програмний компонент запускається паралельно на декількох комп’ютерних ресурсах, потім головна програма періодично виконує перевірку стану кожного з програмних компонентів. Це можливо робити, наприклад, на основі порівняння дампів пам’яті програмних компонент або фіксуючі зміни локальних змінних. У випадку, коли дамп пам’яті одного з програмних компонент відрізняється від дампів пам’яті інших екземплярів програмних компонентів, це свідчить про збій програмного компоненту, і він перезапускається на новому комп’ютерному ресурсі.

Даний метод дозволяє опрацьовувати не тільки ситуації з програмними компонентами, які припинили функціонування у зв’язку з відмовою комп’ютерного ресурсу, але і програмні компоненти, що виконали програмну помилку або відмову каналів зв’язку, а також апаратний збій або помилками інших програм, які виконуються паралельно на гетерогенному

комп'ютерному ресурсі. Пропонується модифікація методу, який запропоновано у роботі [29]

Розглянемо етапи модифікованого методу.

1. Для кожного програмного компонента з множини P створюються додаткові екземпляри, які будуть дублювати роботу. Кількість екземплярів програмних компонентів визначається або користувачем, або системою управління $P^{ext} = \{P \cup P \cup ....\}$. Зазвичай мінімум копій програмних компонент – 3, так як тоді при поодинокій відмови програмного компонента легко встановити цій факт, якщо порівняти результати трьох паралельних програмних компонент.

2. Далі виконується схема призначення для P^{ext}:

$$Sh_k = \{P^{ext}{}_i \to R_j\}, \left| \bigcup_{i=const} \{P^{ext}{}_i \to R_j\} \cap \bigcup_{j=const} \{P^{ext}{}_i \to R_j\} \right| = 1, \qquad (2.4)$$

при цьому умова, яка присутня у виразі забороняє розміщення кількох копій програмних компонент на одному комп'ютерному ресурсі, так як у разі його відмови, буде припинення два екземпляра програмного компонента.

3. Пересилання згідно схеми призначення програмних елементів на виділені комп'ютерні ресурси. Отримання додаткових характеристик о комп'ютерному ресурсі та обчислення часу, який пострибує процес запуску програмного компонента.

4. Запуск необхідного мінімуму програмних компонентів, активація збереження початкового стану програмних компонентів $P_{damp\ i}(t_0)$ для усіх N програм (q=0). На цьому етапі запускаються програмні компоненти, час запуску яких більше, ніж інтервал часу передбаченої відмови.

5. Знаходження множини наявних комп'ютерних ресурсів R^e та їх характеристик після запуску програмної системи та отримання характеристик, які потрібні системі самовідновлення.

6. Визначення значення часу t_q, q=q+1, для якого буде виконаний дамп пам'яті програмних компонент. Перевірити умову закінчення обчислювального процесу. Якщо умова виконана, перехід до пункту 12.

7. При досягненні часу t_q виконання всіх підпрограми множини $\bigcup_{i=1}^{N} P_{dampi}^{ext}(t_q)$.

8. Перевіряємо прогноз або факт настання відмови програмного компонента із системи на основі трьох умов:

- програмний компонент не функціонує (наприклад, відповідь від нього не одержано);
- дані, отримані з екземплярів програмних компонент $P_i^1, P_i^1, P_i^1, ...$ не збігається з іншими;
- отримано прогноз о відмові програмного компонента.

9. Визначення підмножини програмних компонентів системи, для яких існує прогноз відмови, або від яких не отримано даних, або дані не співпадають з відповідями інших екземплярів програмних компонент $P^{Err} \subset P$.

10. Для підмножини P^{Err} виконується перерозподіл ресурсів з та отримується нова схема призначення: $Sh^{Err} = \{P^{Err} \to R^e\}$.

11. Виконання схеми призначення програмних компонентів на комп'ютерні ресурси Sh^{Err}. Запуск програмних компонентів, повертання програмного компонента (компонентів) в стан, який був до відмови $P'_{damp\ i}(t_q)$ для всіх програмних компонентів з множини P^{Err}. Перехід до пункту 5 цього методу.

12. Завершення обчислювального процесу, закриття служб підтримки самовідновлення на усіх комп'ютерних ресурсах.

3 РОЗРОБКА АЛГОРИТМІВ САМОВІДНОВЛЕННЯ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

3.1 Загальні відомості про розроблені програмні засоби

Програмні засоби, які описані в роботі, розроблені у інтегрованому середовищі Microsoft Visual Studio 2017 з використанням мови програмування C# і стандартних функцій операційної системи Windows. Модулі, які реалізують алгоритми самовідновлення написані на мові C++, тому можуть використовуватися в інших операційних системах. Для використання їх у відмінних операційних системах достатньо під нову платформу розробити графічний інтерфейс та інтерфейсі модулі обміну даними в мережі. Для виконання програмних елементів не потрібні додаткові апаратні пристрої або програми. Частина найбільш використаних функцій оформлена у динамічні бібліотеки (dll).

Розроблена версія програмної системи працює в усіх операційних системах на яких установлено платформу з мінімальною версією .Net Framework 4.5.

Мінімальні вимоги до системи:

- 80 Mb вільного місця на жорсткому або мережному диску для серверної частини;
- 25 Mb вільного місця на жорсткому або мережному диску для клієнстької частини;
- комп’ютерна мережа з двох чи більше комп’ютерів з підтримкою протоколів TCP та UDP;
- 512 Mb оперативної пам’яті;
- бібліотеки підтримки мережної взаємодії на основі сокетів;
- операційна система з .Net Framework 4.5.

3.2 Особливості реалізації клієнтської програми

Як вказано у розділі 3.1, програмне забезпечення включає клієнтську та серверну частини. Програмні компоненти клієнту реалізують функції пошуку та з’єднання із сервером, завантаження, запуск та моніторинг стану програмного компоненту розподіленої системи, передачу даних на сервер та інші компоненти, відновлення програмного компонента із збереженими на сервері даними у разі відмови або збою.

Розглянемо докладно алгоритм роботи клієнта (рисунок 3.1), етапи якого наведено представлено нижче:

1. Запуск підсистеми клієнту системи (Start) на кожної робочої станції, яка може прийняти програмний компонент до виконання.

2. Пошук та підключення до серверу (Search and Connect to server), передача інформації про апаратні та програмні ресурси клієнта на сервер, обмін ключами для шифрування.

3. Завантаження та відкриття програмного компоненту (Download and Open application). У випадку, коли є інформація про попередній стан програмного компоненту, клієнт виконує відновлення роботи додатку до цього стану.

4. Цикл стеження та передачі на сервер стану програмного компонента (Work loop) – передача даних на сервер з даними про стан відстежуваного додатку та обчислювальних процесів на клієнту протоколом TCP.

Під час виконання циклу виконуються наступні завдання:

- шифрування даних для серверу за необхідністю (Encrypt data);
- відсилання даних на сервер (Send data to the server);
- моніторінг виконання завдання та перевірка завершення циклу.

5. Аналіз причини виходу з циклу:

- припинення виконання програми користувачем (оператором, сервером, програмною ситемою), після якого не відбувається відновлення програмного компонента (Closed by user);

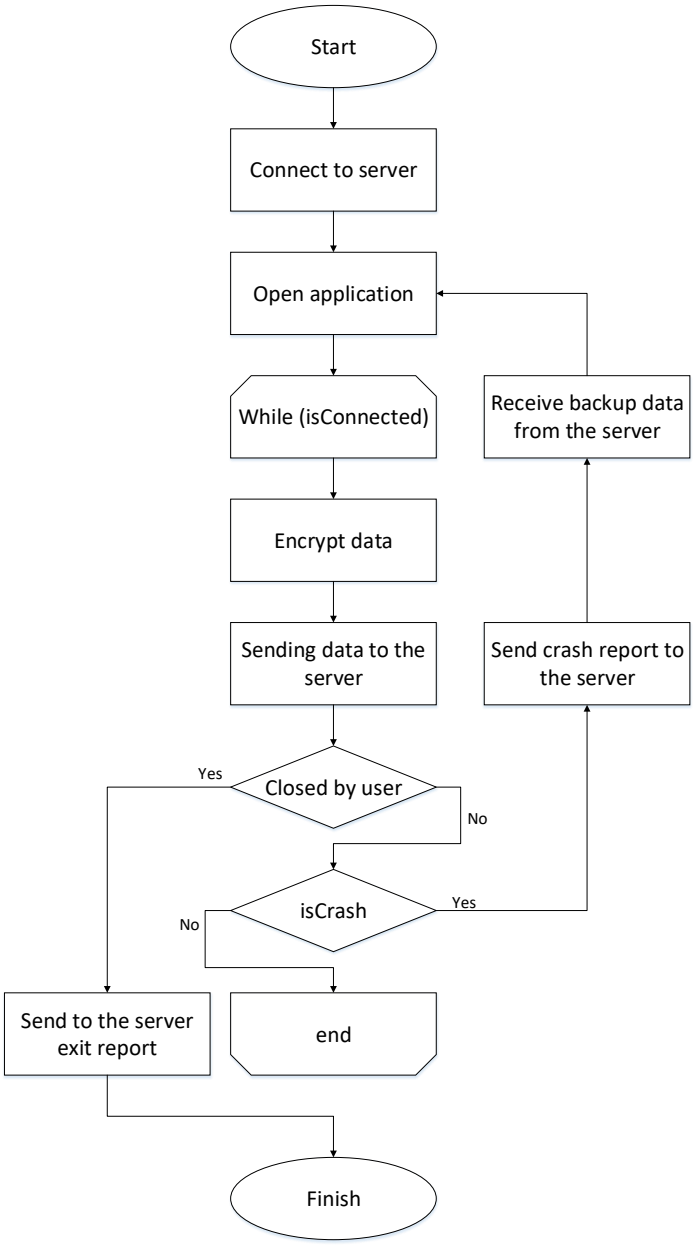

Рисунок 3.1 – Блок-схема алгоритму клієнтської частини

- припинення виконання програми як результат збою ібо відмови (Is crash) – у цьому випадку запускається процедура самовідновлення;
- відправка даних на про збій на сервер. (Send a crash report to the server);
- відправка даних на сервер у випадку припинення користувачем, яке потребує відновлення (Send an exit report to the server);

- отримка повідомлення від сервера про останній стан відстежуваного додатку.

6. У випадку мажоритарного відстеження (Receive backup data from the server) .

3.3 Особливості реалізації серверної програми

На рисунку 3.2 наведено блок-схему алгоритму функціонування серверу підтримки самовідновлення.

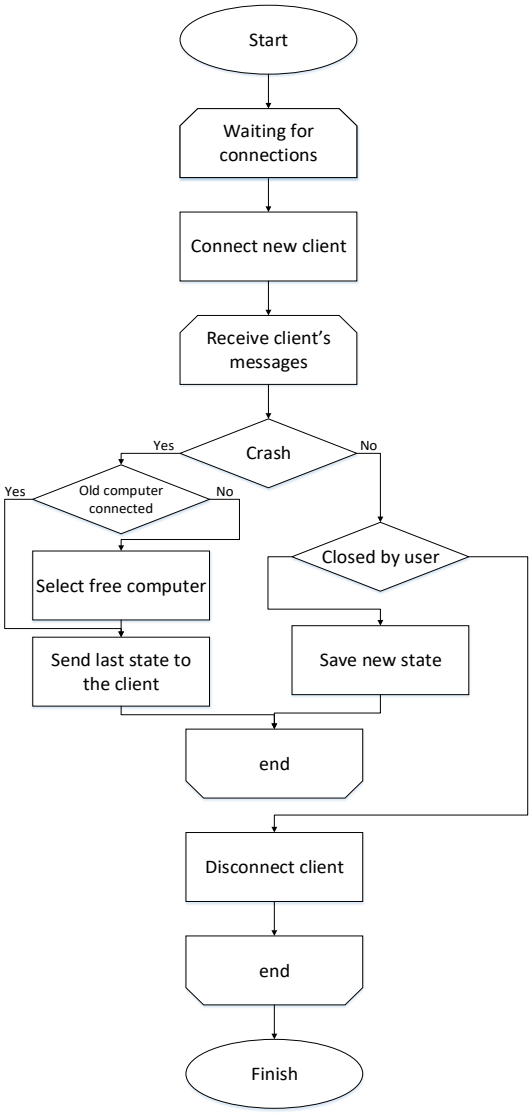

Рисунок 3.2 – Блок-схема алгоритму функціонування серверної частини

Основна ідея серверної частини системи самовідновлення полягає в готовності з’єднань з клієнтськими програмами, отримання даних о стані від них, збереження даних у доступній базі даних, відповідь на запити про відновлення програмних компонент, відправка останньої збереженої контрольної точки з ціллю скорішого самовідновлення після відмови на тому самому клієнті, або пошук вільного обчислювального ресурсу з відповідними характеристиками, ініціалізація та відновлення на новому клієнті програмного компонента. Розглунимо основні шаги алгоритму.

1. Запуск серверної частини (Start).

2. Сервер прослуховує мережу в очікуванні підключень клієтьских застосувань. Це відбувається протягом усього часу роботи серверу та реалізується засобами операційної системи (Waiting for connections).

3. Підключення при наявності запиту нового клієнта (Connect new client).

4. Ініціалізація циклу пересилки даних між серверною та клієнтскою частинами системи (Receive client’s messages). У рази включення системи шифрування (Receive and decrypt message).

Під час виконання циклу виконуються наступні дії з аналізу ситуації, яка відповідає запиту з клієтської частини.

5. Ситуація збію або відмови (Crash). Це може бути інформація від клієнта про збій, інформаація від операційної системи втрату з’єднання з клієнтом, перевищено час отримання інформації про поточній стан програмного компонента клієнської частини.

6. Перевірка умови розірвання з’єднання з клієнтською частиною (Old computer connected).

7. У випадку, коли з’єднання з комп’ютером втрачено, виконується пошук іншого (Select free computer).

8. Посилання збереженого стану програмного компоненту на клієнтську машину, ініціація відновлення функціонування програмного компонента (Send last state to the client).

9. Корректне завершення виконання програмного компонента на боці клієнта (Closed by user).

10. Збереження стану програмного компонента (Save new state) в бізі даних).

11. Відключення клієнта (Disconnect client).

12. Припинення роботи серверу (Finish).

3.4 Взаємодія клієнта з програмним компонентом

Взаємодія клієнта з програмним компонентом визначається тим, на якому етапі проектування програмного забезпечення проектуються засоби відновлення. При самовідновленні самий оптимальний час – це розробка засобів відновлення під час реалізації прикладного програмного зебезпечення.

На рисунку 3.3 наведено різні способи розробку самовідновлювального застосування.

Рисунок 3.3 – Способи розробки застосування

З нашої точки зору найбильш ефективними є два напрямки такої розробки: це інтеграція засобів самовідновлення в ісходну програму або використовування динамічно приєднуваних бібліотек.

Перший спосіб зазвичай приводить к программі, код якої неможливо змінити без перекомпіляції. Розробка сервера та клієнта розробляється програмистами однієї команди. У даному випадку нема обмежень на антегроване програмне забезпечення, мову програмування, інформаційні технології, архітектури, мерені протоколи. Даний напрямок надає можливість до оптимізації коду, модифікації засобів в подальшому. Можливо закладувати можливість оновлення програмних компонент в подальшому.

У випадку, коли клієнт та програмні компоненти приклодної задачі розробляються різними колективами програмістів, є можливість використання динамічно приєднувану бібліотеку (dll), яка проектується відокремлено для реалізації взаємодії двох застосувань. Ця можливість дозволяє клієнту та програмним компонентам використовувати одину й теж бібліотеку DLL.

Ще один з способів – завершена програмна система з використанням закритого програмного коду. Цей спосіб реалізує усі функції відстеження програмного компоненту на клієнта. У цілому – можливості відстеження меньше ніж у попередніх випадках і можливість їх розштрення відсутня. А у випадку, коли розробники не надали програмний інтерфейс користувача (API), який надає можливість отриманя інформації про стан програмного компонента, сберігати можливо мабуть елементи графічного інтерфейсу (GUI), або дані взаємодії програмного компонента з операційною системою (наприклад з файловою системою).

Для кожного нового застосування із закритим програмним кодом треба розробляти свого клєнта. Водночас такий спосіб можливо реалізувати для любої програми, яка розроблялась раніше та не має своїх засобів самовідновлення.

4 РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ САМОВІДНОВЛЕННЯ РОЗПОДІЛЕНИХ ПРОГРАМНИХ СИСТЕМ

4.1 Проектування архітектури програмної системи

Розробка програмного забезпечення починається з архітектурного рівня. Представимо розроблену систему самовідновлення архітектурного рівня в виді об'єктної моделі, яка відображає базові класи, класи–спадкоємці та зв'язок між ними.

Для проектування моделей програмних систем існує безліч підходів, які не мають силу стандартів, загальних правил, методів, які гарантують до того ж правильний результат. Є деякі характеристики, під які спробують підігнати програмні компоненти. На моделі програм також впливають час розробки, налагоджування та подальшого використання. Деякі значні параметри використання програмних засобів, наприклад, здатність до масштабування, можливо оцінити на етапі проектування. Усі великі програмні системи після впровадження мають супровід – під час цього періоду продовжується роботи по програмуванню. Тобто, модель програмного забезпечення може змінюватись, доповнюватись та розвиватися вже під час експлуатації системи [6].

Розглянемо класи, які були розроблено в даній роботі:

- клас Client – у своєму складі має методи для з'єднання з сервером, також виконує моніторинг програмних компонент, відправляє на сервер інформацію про стан програмного компоненту, а також виконує відновлення у разі відмові або збою;
- клас Remote Computer працює на сервері, зберігає інформацію про стан програмного компонента, підтримує зв'язок з клієнтом, під час відмови аба збою активує процес відновлення та передачу останнього стану програмного компонента;

- клас Server – містить множину усіх клієнтів, підтримку керування зв’язком з ними, підключає, якщо необхідно, шифрування.

- класс State використовується для обробки інформації про поточний програмних компонент.

На рисунку 4.1 представлено UML діаграму класів розробленого програмного забезпечення.

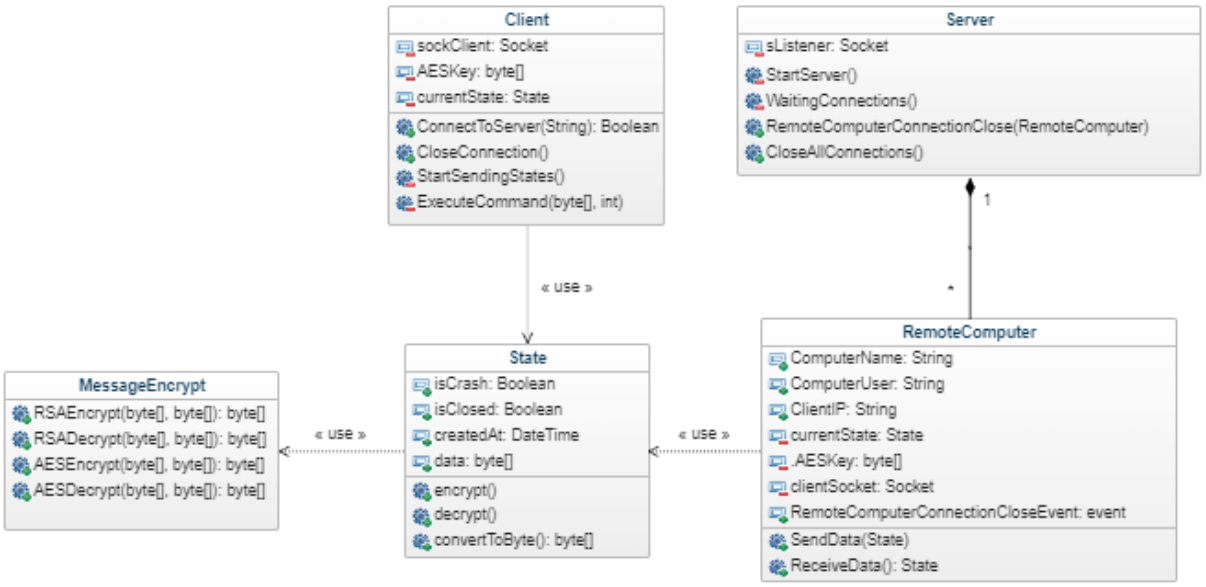

Рисунок 4.1 – UML діаграма класів

Зазвичай UML-діаграми використовуються щодо архітектурного рівня моделі розробленої програмної системи. Вони надають розуміння структури програми, її елементв, а головне – взаємовідносин між елементами програмної системи. Для цього ця стандартизована нотація має широкий вибір графічних засобів [7].

Методи першого класу Client, наведені у прикладі 4.1, зберігають стан програмного компонента, мережний адрес серверу, сонет з’єднання з сервером, ключі шифрування та потоки виконання, які дозволяють реалізувати багато поточний режим обробки даних та паралельну роботу з сервером з прийому та передачі даних.

```
private Socket sockClient;
private string serverIp;
private Thread SendThread;
private Thread ReceiveThread;
private byte[] AESkey;
private State currentState;
```

Приклад 4.1 – Приватні поля базового класу Client

Приватний метод ConnectToServer робить з'єднання з сервером за допомогою сокету. Єдиний параметр методу містить ip-адресу сокету. У даному випадку використовується протокол TCP. Після з'єднання метод передає до серверу інформацію про комп'ютер клієнту. Функція повертає true, якщо з'єднання виконалось, та false в іншому випадку (приклад 4.2).

```
private bool ConnectToServer(String serverIp)
{
    IPHostEntry ipHost = Dns.GetHostEntry(serverIp);
    foreach (var address in ipHost.AddressList)
    {
       try
       {
          var ipe = new IPEndPoint(address, port);
          var tempSocket = new Socket(ipe.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
          tempSocket.Connect(ipe);
          if (tempSocket.Connected)
          {
                 sockClient = tempSocket;
                 serverIp = address.ToString();
                 Commands.SendInfoToServer(sockClient,
Environment.MachineName, Environment.UserName,
Dns.GetHostByName(Environment.MachineName).AddressList[0].ToString());
                 return true;
           }
       }
       catch (SocketException e) { }
    }
    return false;
}
```

Приклад 4.2 – Функція підключення до серверу

Після цього клієнт переходить у стан очікування. Щоб не припиняти виконання основного потоку програми, метод, який відповідає за очікування, виконується в окремому потоці. Таким чином функція очікування не

припиняє виконання основної програми та виконується паралельно головному потоку програми. Функція наведена у прикладі 4.3.

```
private void ReceiveCommand()
{
     byte[] commandBuffer = new byte[1500];
     while (true)
     {
          try
          {
               int length = sockClient.Receive(commandBuffer);
               ExecuteCommand(commandBuffer, length);
          }
          catch (Exception e)
          {
               ReceiveThread.Abort();
               return;
          }
     }
}
```

Приклад 4.3 – Функція очікування команд від серверу

Для отримання команд від серверу в локальній пам’яті функції виділяється буфер даних розміром 1,5 кВ. Цикл функції періодично перевіряє наявність даних від серверу. Якщо дані отриманні, виконується виклик функції виконання команди серверу, після обробки якої, цикл очікування подовжується. Розмір команди не повинен перевищувати розмір локального буферу даних. В іншому випадку генерується виключення, яке не приводить к зупинці програми, але цикл очікування припиняється. В останньому випадку головний потік програми може викликати функцію очікування команд з серверу ще раз.

Якщо сервер дає команду запуску програмного компоненту, клієнт активізує метод передачі стану програмного компонента на сервер. Активація здійснюється функцією, наведеною у прикладі 4.4.

```
private void StartSendingStates()
{
     if (!isSendingStates)
     {
          isSending = true;
          SendThread = new Thread(SendState);
          SendThread.Start();
     }
}
```

Приклад 4.4 – Функція ініціалізації контролю та передачі стану програмного компонента

Надана функція перевіряє коректність початку контролю стану (можливо, стан вже контролюється) та створює окремий потік, для періодичного надсилання інформації про стан програмного компонента серверу. Таким чином, паралельно основному потоку програми виконуються потоки очікування команд та передачі даних серверу.

З боку сервера зв’язок з усіма клієнтами також підтримується постійно. Сервер приймає дані про програмні компоненти та зберігає їх у базі даних. Якщо відбувається відмова або збій, сервер використовує ці дані для відновлення роботи програмного компонента.

Якщо є запит від нового клієнта, створює екземпляр класу RemoteComputer, який буде відповідати за поведінку серверу в усіх діях з клієнтом, включаючи отримання станів програмних компонент, посилання команд клієнту, реакцію на відмови та інше.

В екземплярі класу присутні всі дані про комп’ютерні ресурси клієнта, адреса, порт, процесор, пам’ять. Якщо є необхідність, сервер може отримати доступ до відео пам’яті та транслювати зображення з екрану. У майбутньому є можливість розширити можливості серверу, але ці зміни потребують програмування як з боку серверу, так і з боку клієнта.

Поля класу, серед яких є методи, які виконують роль делегатів, які реагують на повідомлення з боку клієнта, властивості, які відповідають параметрам ресурсу клієнта та сокет зв’язку з клієнтом (приклад 4.5).

```
public string ComputerUser { get; private set; }
public string ClientIP { get; private set; }
public string ComputerName { get; private set; }
public State ComputerState { get; private set; }
private Socket clientSocket;

public delegate void RemoteComputerConnectionDelegate(RemoteComputer r);
public event RemoteComputerConnectionDelegate
RemoteComputerConnectionCloseEvent;
```

Приклад 4.5 – Поля класу клієнта з боку сервера

Конструктор даного класу має 4 аргументи, які ідентифікують комп'ютер та застосування клієнта в комп'ютерній мережі (приклад 4.6). Після збереження усіх параметрів клієнта, конструктор створює відокремлений потік, який буде постійно приймати стан програмного компонента з клієнта.

```
public RemoteComputer(Socket _clientSocket, string _computerName = "Unknown",
string _computerUser = "Unknown", string _ip = "Unknown")
{
      ComputerName = _computerName;
      ComputerUser = _computerUser;
      ClientIP = _ip;
      clientSocket = _clientSocket;
      Thread ReciveStateThread = new Thread(ReciveState);
      ReciveStateThread.IsBackground = true;
      ReciveStateThread.Start(); //запускаємо потік
}
```

Приклад 4.6 – Конструктор класу взаємодії з клієнтом

Таким чином, ми розглянули основні класи клієнтського та серверного застосувань, які підтримують процес самовідновлення програмних компонент та складають основу архітектури розроблених програмних засобів.

4.2 Тестування програмної системи

Тестування програмної системи необхідно виконати як з боку сервера, так і з боку клієнта. Тестування було проведено для усіх класів, методів, функций з обох боків. Більш того, серверна частина пройшла перевірку у випадку підключення кількох клієнтів.

У зв'язку з відсутністю доступу до комп'ютерних мереж під час написання атестаційної роботи, була використана віртуальна (программна) машина (Oracle VirtualBox) для симуляції повідінки розробленого програмного забезпечення. Було змодельовано середовище, яке має три компьютери, два з яких виконували клієнтську частину програмного забезпечення, та останній комп'ютер – серверну частину.

Тестування розпочалося з сценарію підключення декількох клієнтів до серверу та їх синхронної роботи (рисунок 4.1). На рисунку можно бачити сервер, який розташовано на віртуальному ресурсі з адресом 192.168.229.126, та трех клієнтів, один з яких розташовано на тому же ресурсі, два інших – вікористовуют реальний та віртуальний комп'ютери.

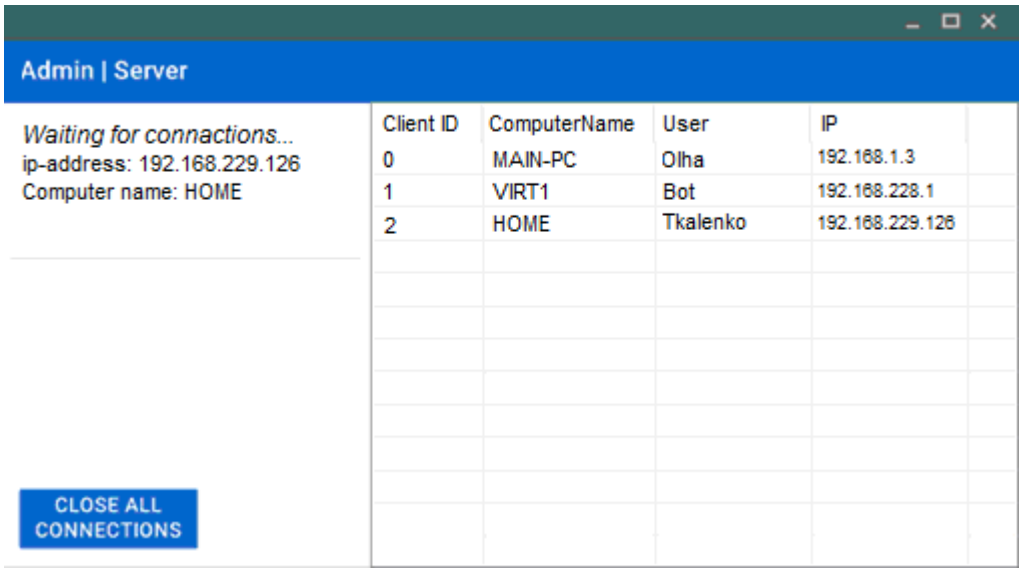

Рисунок 4.1 – Клієнти, що зв'язані з сервером

Вікно моніторингу сервера фіксує з’єднання з усіма клієнтами. Клієнти отримали програмні компоненти, самовідновлення яких забезпечував сервер на основі даних, які постійно передавалися з боку клієнтів. Таким же чином можливо організувати тестування на реальних сегментах комп’ютерної мережі.

Тестування режиму самовідновлення відбулося шляхом припинення роботи однієї з віртуальних робочих станцій, після чого сервер відновив роботу клієнта на тому же комп’ютері, на якому знаходився сам. Час відновлення склав 20 секунд. Після цього усі програмні компоненти продовжили свою роботу.

Інший тест було проведено для відновлення програмного компоненту на іншому віртуальному комп’ютері. Самовідновлення у цьому випадку включало передачу клієнту програмного компонента, його останнього стану з серверу, запуск програмного компонента на боці нового клієнта та синхронізацію роботи з іншими програмними компонентами. У цьому випадку час самовідновлення склав 153 сек. Час самовідновлення був більш, ніж очікувалось, що викликано ситуацією, коли віртуальній комп’ютер резервного клієнта, перейшов у сплячий режим, і потрібен був час для виходу з цього режиму.

4.3 Інструкція користувача

Було створено загрузочний файл, який розгортає програмну систему на окремому ресурсі. Під час інсталяції користувач (або адміністратор) відповідає на запитання – яку частину требі інсталювати – серверну або клієнтську. Можливий варіант розміщення обох частин на одному ресурсі. Більш того, кількість клієнтів на одному ресурсі не обмежена, але на одному ресурсі можливо установити тільки один сервер. Після інсталяції усі компоненти встановлено.

Для запуску системи відкривається діалогове вікно (рисунок 4.2).

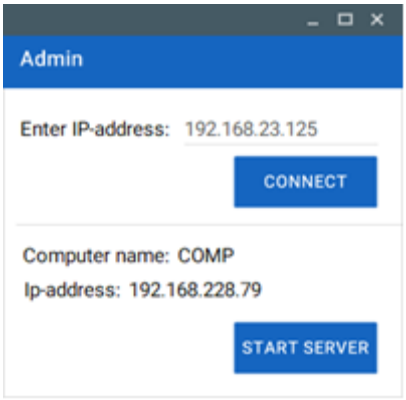

Рисунок 4.2 – Стартове вікно системи

У розділі Enter IP–address необхідно вказати адресу серверу, у низу – адрес наявного комп'ютеру. Якщо даний ресурс буде працювати в режимі серверу, необхідно обрати опцію "Start server".

Клієнт функціонує у скритому режимі (Hide), у той час коли сервер має графічний інтерфейс, який відображає усіх клієнтів, стартовий вид якого представлено на рисунку 4.3.

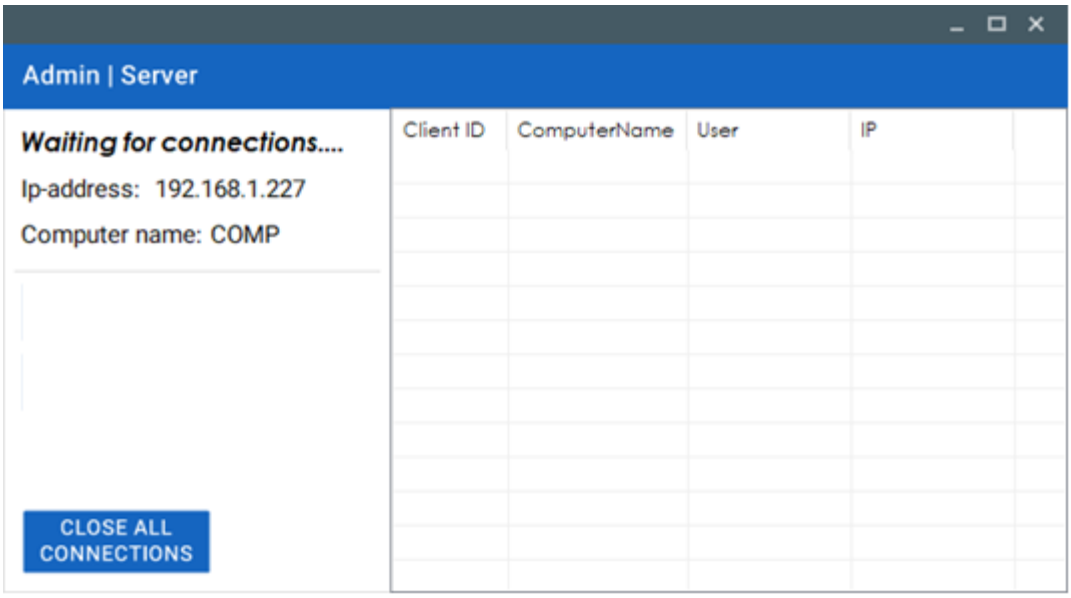

Рисунок 4.3 – Вікно серверної частини

У вікні програми серверу вказується ір-адрес та ім’я комп’ютера на якому запущено програму. Тут же вказується стан серверу. На рисунку сервер знаходиться у стані очкування. Після підключення клієнтів, інформація о них розміщується у таблиці.

Сторона клієнта може запускати програмні компоненти тільки після підключення до серверу. В подальшому планується розширити графічний інтерфейс серверу для відображення поточного стану клієнтів та програмних компонент, процесу відновлення програмних компонент тощо.

ВИСНОВКИ

В процесі роботи було досягнуто підвищення ефективності використання розподілених програмних систем за рахунок розробки та впровадження модифікованого мажоритарного методу самовідновлення, який зменшує час та знижує вартість відновлення.

Для досягнення встановленої мети в процесі роботи було необхідно вирішити наступні задачі:

- досліджено існуючі методи та засоби забезпечення відновлення розподілених програмних систем;

- обрано мажорітарний метод самовідновлення, як метод з найменьшим часом виконання;

- розроблено модифікацію мажорітарного методу забезпечення самовідновлення програмного забезпечення в гетерогенних комп'ютерних системах, який знижує вартість процесу підтримки самовідновлення;

- розроблено програмні засоби підтримки забезпечення самовідновлення програмного забезпечення в гетерогенних комп'ютерних системах.

Спираючись на особливості наявних на ринку програмних продуктів, було сформульовано вимоги, спроектовано та розроблено програмне забезпечення для самовідновлення після відмов та збоїв.

Додаток було спроектовано на основі «клієнт-серверної» архітектури, для обміну даними використано технологію Windows Socket на основі протоколу передачі даних TCP.

Програмна система підтримки самовідновлення розроблена з використанням об'єктно-орієнтованої мові C#, у середовищі Microsoft Visual Studio 2017. Програмна система має додатковий інтуїтивно-зрозумілий графічний інтерфейс.

Тестування програмних засобів показало, що розроблені програмні засоби задовольняют завданню на аттестаційну роботу, усі реалізовані функції роботають без збоїв.

Програмні засоби можуть застосовуватися для підтримки надійності програмного забезпечення веб серверів, інформаційних систем, та на відмінну від аналогічних продуктів, в розроблених засобах є можливість наділити властивістю самовідновлення будь-які програмні системи на стадії розробки, або вже під час використання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. John Strassner and Jerey O. Kephart, \Autonomic Systems and Networks: Theory and Practice, Network Operations and Management Symposium (NOMS), 2016.

2. Jerey O. Kephart and David M. Chess, The Vision of Autonomic Computing, IEEE Computer, Vol. 36(1), January 2013.

3. IEEE Standard Classication for Software Anomalies, IEEE Std 1044-1993, 1993. A Modeling Framework for Self-Healing Software Systems 9.

4. John Musa, Software Reliability Engineering, McGraw-Hill, 1999.

5. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, \Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, 2014, pp. 11-33.

6. Leonardo Mariani, A fault taxonomy for component-based software, Proceedings of International Workshop on Test and Analysis of Component-Based Systems (TACoS), Electronic Notes in Theoretical Computer Science, Vol. 82, Elsevier Scence, 2013.

7. Gregor Kiczales, et al, Aspect-Oriented Programming, European Conference on Object-Oriented Programming, Finland, June 2011.

8. Salim Hariri, Alok Choudhary, and Behcet Sarikaya, Architectural Support for Designing Fault-Tolerant Open Distributed Systems. IEEE Computer, 1992.

9. Richard Soley and the OMG Sta Strategy Group, Model-Driven Architecture, [Електронний ресурс] ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

10. Markus Voelter, A Collection of Patterns for Program Generation, Eighth European Conference on Pattern Languages of Programs, Irsee, Germany, June 2013..

11. Didonet Del Fabro Marcos, Bzivin Jean, Jouault Frdric, Breton Erwan, Gueltas Guillaume, AMW: A Generic Model Weaver, In Proceedings of the 1re Journesur l'Ingnierie Dirige par les Modles (IDM05), Paris, France, June-July 2015.

12. Uwe Amann, Invasive Software Composition, Springer-Verlag, 2003.

13. Patrick Francis, David Leon, Melinda Minch, Andy Podgurski, Tree-Based Methods for Classifying Software Failures, Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, November 2014.

14. Object Management Group, UML Pro le for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, [Електронний ресурс] http://www.omg.org/cgibin/doc?formal/06-05-02, 2016.

15. David Garlan, Bradley Schmerl and Jichuan Chang, Using Gauges for Architecture-based Monitoring and Adaptation, Working Conference on Complex and Dynamic Systems Architecture, Australia, 2011.

16. Janak Parekh, et al, Retrotting autonomic capabilities onto legacy systems, Journal of Cluster Computing, 2005, pp. 141-159.

17. A. Reza Haydarlou, et al, A Self-healing Approach for Object-Oriented Applications, 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems, 2005.

18. M. Muztaba Fuad and Michael J. Oudshoorn, Transformation of Existing Programs into Autonomic and Self-healing Entities, 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), 2007.

19. Michael E. Shin and Daniel Cooke, Connector-Based Self-Healing Mechanism for Components of a Reliable System, Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), St. Louis, Missouri, May 21, 2005.

20. M. Xue, L. M. Hitt, P. Y. Chen, The Determinants and Outcomes of Internet Banking Adoption, Management Science (Forthcoming), 2011.

21. H. A. Al-Zu’bi, A. M. Ahmad, E-banking Functionality and Outcomes of Customer Satisfaction: An Empirical Investigation, International Journal of Marketing Studies Vol. 3, pp. 50-65 February 2011.

22. F. Calisir, C. A. Gumussory, Internet Banking versus Other Banking Channels: Young Consumers’ View, International Journal of Information Management, pp 215-221, 2018.

23. B. Goodwin-Jones, Emerging technologies – accessibility and web design why does it matter? Language Learning and Technology, pp. 11-19, 2001.

24. M. O. Hilari, Quality of Service (QoS) in SOA Systems: A Systematic Review, 2009.

25. D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self adaptation with reusable infrastructure, IEEE Computer, vol. 37, pp. 46-54, October 2014.

26. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, Using architecture models for runtime adaptability, IEEE Software, vol. 23, pp. 62-70, 2006.

27. E. M. Dashofy, A. v. Hoek, R. N. Taylor, Towards architecture-based self-healing systems, in WOSS'02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, pp. 21-26, 2012.

28. M. E. Shin, J. H. An, Self-reconfiguration in self-healing systems, in EASE'06: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems, Washington, DC, USA, pp. 89-98, 2006.

29. Волк М. О. Моделі, методи та інформаційна технологія управління розподіленим обчислювальним процесом в гетерогенних комп’ютерних системах: автореф. дис. д-ра техн. наук. Харків: ХНУРЕ, 2019. 43 с.

30. M. M. Fuad, M. J. Oudshoorn, Transformation of existing programs into autonomic and self-healing entities, in ECBS'07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Washington, DC, USA, pp. 133-144, 2007.