Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Удосконалення методу та засобів
забезпечення відмовостійкості програмних систем

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРПЗ.2001111.01.01.ПЗ

Виконав студент 2 курсу група ІПЗм-20-1 _____ Д. Ю. Антіч
Підпис Ініціали, прізвище

Керівник канд. техн.наук, доцент _____ Д. М. Медзатий
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн.наук, доцент _____ Г. І. Радельчук
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

3 грудня 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01 09 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Антічу Дмитру Юрійовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Удосконалення методу та засобів забезпечення відмовостійкості програмних систем

Керівник проєкту (роботи) Медзатий Дмитро Миколайович, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2021 р. № 108

2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2021 р.

3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Дослідження предметної області та постановка задачі
2 Концепції, моделі та методи вирішення задачі
3 Технології вирішення задачі
4 Реалізація та тестування програмного засобу

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І. В., доцент	23.11.2021 р.	01.12.2021 р.
Нормоконтроль	Радельчук Г. І., доцент	23.11.2021 р.	01.12.2021 р.

7. Дата видачі завдання «01» вересня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження, визначення об'єкта та предмета дослідження, формування логістичної структури дипломної роботи	01.09-10.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних та Інтернет-джерел, аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі, висновки до розділу та постановка задач дослідження	11.09-25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка моделей, методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2021	
4 Робота над науковими статтями	11.10-30.10.2021	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2021	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень, висновки до розділу	27.10-17.11.2021	
7 Попередній захист дипломної роботи	Листопад (згідно графіка)	
8 Узгодження постановки задачі, отриманих результатів та висновків, написання вступу, загальних висновків, оформлення джерел посилання та додатків, оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	18.11-30.11.2021	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2021	
10 Підготовка до захисту дипломної роботи	з 01.12.2021 р.	

Студент _____ Д. Ю. Антіч
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Д. М. Медзатий
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Удосконалення методу та засобів забезпечення відмовостійкості програмних систем»

Автор роботи: Антіч Дмитро Юрійович.

Керівник роботи: Медзатий Дмитро Миколайович.

Пояснювальна записка: 128 с., 25 рис., 2 табл., 3 дод., 22 джерел.

ВІДМОВОСТІЙКІСТЬ, ПАТЕРН ВІДМОВОСТІЙКОСТІ, БАЛАНСУВАЧ, ШАРДІНГ, ВИМИКАЧ, ПРОГРАМНИЙ ЗАСІБ.

Об’єкт дослідження – відмовостійкість програмних систем.

Мета дослідження – удосконалення методу відмовостійкості програмних систем та розробка програмного засобу на його основі, що дасть змогу обробляти та зменшувати негативний вплив відмов (у тому числі часткових).

У роботі використані наступні методи дослідження та апаратура:

– спостереження, експеримент, абстрагування, аналіз та синтез, формалізація;

– інструментальні засоби проектування, програмування та тестування;

– персональний комп’ютер.

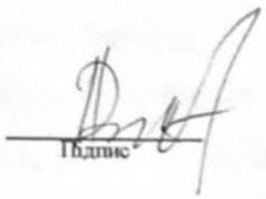
У процесі дипломного проєктування досліджено галузь відмовостійкості та сучасні методи і засоби забезпечення відмовостійкості програмних систем, визначено невирішені проблеми у галузі та методологічні підходи до їх вирішення; на базі невирішених проблем розроблено удосконалений метод забезпечення відмовостійкості програмних систем та виконано його програмну реалізацію.

Розроблений метод покращує відмовостійкість додатків, побудованих за мікросервісною або кліент-серверною архітектурою, шляхом реалізації нових підходів, переосмислення та комбінації існуючих алгоритмів відмовостійкості програмних систем, що дало змогу удосконалити працездатність та стабільність систем, збільшити їх пропускну здатність та зменшити кількість помилок.

Для програмної реалізації методу використано технології Docker, NodeJS, Express та інші, а для тестування – технології Jest та JMetter.

Проведені емпіричні дослідження доводять адекватність та ефективність розробленого методу відмовостійкості програмних систем, працездатність та функціональну придатність реалізованого на його основі програмного засобу.

Апробація отриманих результатів дослідження показала зменшення кількості помилок у програмних системах приблизно на 50% і збільшення кількості оброблених запитів на 50% у порівнянні з існуючими алгоритмами відмовостійкості. Отже, розроблений програмний засіб можна рекомендувати для використання ІТ-підприємствам, які мають на меті підвищити відмовостійкість власних розроблюваних додатків.



Підпис

01.12.2021
Дата

ABSTRACT

Master’s thesis: «Methods and Means Improvements for Ensuring Software Systems Resiliency».

Author: Antich Dmytro.

Head of research: Medzatiy Dmytro.

Master’s thesis consists of: 128 p., 25 pc., 2 tb., 3 add., 22 srs.

RESILIENCY, RESILIENCY PATTERNS, LOADBALANCER, SHARDING, CIRCUIT BREAKER, SOFTWARE.

The subject of the research is the processes of establishing and ensuring resiliency and systems reaction to failures.

The aim of the research is improvements of resiliency algorithm and development of new software on its basis which allows to handle and reduce the negative effect of failures (including partial failures).

The following approaches of the research are used during the study:

– monitoring, experiment, abstraction, analysis and synthesis, formalization;

– modern instrumental approaches of system design and development;

– personal computer.

During the study the resiliency sphere was researched and investigated, unsolved problems were detected and an approach of solving them was suggested, the new algorithm’s requirements and features were defined.

Developed method improves resiliency of software applications build on microservices and client-server architecture by implementation of new approaches, re-building and combination of existing resiliency algorithms what improved software efficiency, performance and stability, improved bandwidth, and reduced quantity of errors.

Docker, NodeJS, Express were used for the implementation of improved algorithm and Jest and JMetter were used for testing purposes.

Empirical research proves adequacy and efficiency of developed resiliency method, efficiency and functional applicability of developed on its basis software system.

Approbation of outcomes shown, that quantity of errors was decreased by approximately 50% and quantity of processed requests was increased by approximately 50% in comparison to existing resiliency patterns. Therefore, developed algorithm can be suggested for usage in IT-companies, which aim to improve resiliency of own applications.

Signature

01.12.2021
Date

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
ПЗ	–	програмний засіб
ПС	–	програмна система
СКБД	–	система керування базами даних
DNS	–	Domain Name System
HTTP	–	Hyper Text Transfer Protocol
LXC	–	LinuX Containers
MTBF	–	Mean Time between Failures
MTTF	–	Mean Time to Failure
MTTR	–	Mean Time to Recovery
NPM	–	Node Package Manager
SLA	–	Service Level Agreement
SMS	–	Short Message Service
SOA	–	Service-Oriented Architecture
SSL	–	Secure Sockets Layer
UML	–	Unified Modeling Language
URL	–	Uniform Resource Locator

ВСТУП

На сьогоднішній день складно уявити життя без мережі Інтернет. Лише за 2021 рік кількість користувачів мережі виросла на 5% (222 мільйонів нових користувачів). У середньому власник смартфону проводить онлайн 155 хвилин на день [1].

Таким чином, 89% компаній ставлять діджитал-трансформацію на перше місце. Проте, наявність веб-сайту не гарантує успішність бізнесу в Інтернет [2].

Згідно із дослідженням Kissmetrics [3], 73% користувачів Інтернет зіткнулися з веб-сайтом, який завантажувався занадто повільно, а 38% стверджували, що стикались із сайтами, які не завантажувались взагалі. Як правило, неробочий веб-сайт означає, що користувач відкриває веб-сайт конкурента. Тому стабільність ПС є основним пріоритетом компанії. Таким чином, існує потреба швидко реагувати та виправляти проблеми, щоб забезпечити максимальну стійкість веб-сайту. Одним із варіантів реагування та виправлення проблем є використання методів відмовостійкості програмного забезпечення, що дозволяють, завдяки моніторингу стану програмного продукту, вживати заходів щодо їх вирішення.

На сьогоднішній день методи забезпечення відмовостійкості ПС є ще актуальнішими, ніж вони були менш ніж п’ять років тому, через стрімку діджиталізацію проектів та бізнесів, особливо таких сфер як освіта, медицина, торгівля тощо, які змушені були стрімко адаптуватись до нових реалій у зв’язку з частими локдаунами та низками обмежень під час пандемії (оскільки клієнти не мають іншої альтернативи для використання послуг).

Проаналізувавши наукові дослідження у сфері відмовостійкості ПС, можна виділити наступні невирішені проблеми у галузі:

- відсутність врахування та обробки часткових відмов ПС;
- неспроможність зменшити кількість помилок у реальному часі;
- відсутність спроможності вирізняти першопричину помилки.

Таким чином, виникає потреба у вдосконаленні наявних методів забезпечення відмовостійкості ПС, які б надавали можливість аналізувати, реагувати та зменшувати негативний вплив часткових відмов у програмній системі.

Тому завданням магістерської роботи є:

– визначення моделі забезпечення максимальної відмовостійкості ПС з покриттям часткових відмов;

– забезпечення зменшення негативного впливу відмов шляхом розширення та комбінації існуючих патернів відмовостійкості ПС;

– аналіз результатів роботи удосконаленого алгоритму відмовостійкості ПС та визначення його ефективності у порівнянні з існуючими рішеннями.

Актуальність роботи полягає у тому, що в будь-якій програмній системі хоча б один раз траплялись відмови, тому необхідно використовувати алгоритми відмовостійкості ПС.

Об’єкт дослідження – відмовостійкість програмних систем.

Предмет дослідження – методи, алгоритми та засоби забезпечення відмовостійкості ПС.

Мета дослідження – удосконалення методу відмовостійкості ПС та розробка програмного засобу на його основі, що дасть змогу обробляти та зменшувати негативний вплив відмов ПС (у тому числі часткових).

Для досягнення поставленої мети необхідно вирішити наступні задачі:

– проаналізувати (у загальному) сферу відмовостійкості ПС;

– дослідити сучасні підходи, концепції та методи забезпечення відмовостійкості ПС з метою виявлення наявних проблем та невирішених питань;

– удосконалити наявні методи забезпечення відмовостійкості ПС для вирішення (або часткового вирішення) виявлених проблем;

– на основі удосконаленого методу відмовостійкості ПС виконати проектування відповідного програмного засобу;

– виконати програмну реалізацію удосконаленого методу відмовостійкості ПС;

– провести тестування та практичну апробацію отриманих результатів;

– дослідити ефективність запропонованих рішень;

– проаналізувати отримані результати дослідження та сформувати рекомендації щодо доцільності їх впровадження.

Для досягнення мети використано теоретичні та емпіричні методи дослідження, а саме:

а) теоретичні методи:

– абстрагування – один з важливих методів, який дозволяє відкинути несуттєві параметри (від абстрагування напряму залежить ефективність моделі);

– аналіз та синтез – декомпозиція моделі на прості складові, виявлення зв’язків між компонентами і, відповідно, синтез цих структурних елементів у єдине ціле;

– формалізація – представлення моделі у вигляді програмного коду;

б) емпіричні методи:

– спостереження (темою роботи є вдосконалення алгоритмів відмовостійкості ПС, але для того, щоб виділити корисні ознаки, які мають бути імплементовані у розроблюваних рішеннях, слід провести спостереження існуючих рішень, визначити властивості та зв’язки між ними);

– експеримент (на етапі дослідження існуючих аналогів слід відтворити певні умови, які потрібні для аналізу імплементованих алгоритмів; пізніше цей же метод використовується для аналізу ефективності результативного методу, який розроблено та імплементовано у ході роботи).

Наукова новизна отриманих результатів:

– отримав подальший розвиток метод Шардінг у напрямку його застосування для розподілених мережних застосунків, що дозволило підвищити відмовостійкість ПС завдяки лімітації негативного впливу потенційно небезпечних користувачів лише до контейнерів їх шарду;

– розроблено метод визначення та ізоляції підозрілих користувачів, що дало змогу мінімізувати їх негативний вплив на ПС та, відповідно, зменшити кількість помилок у системі;

– удосконалено метод забезпечення відмовостійкості ПС за рахунок комбінації алгоритмів Балансувач, Вимикач та Шардінг, що дало змогу отримати максимум переваг кожного з патернів та забезпечити максимальне покриття як повних, так і часткових відмов ПС.

Практична цінність отриманих результатів полягає у вдосконаленні методу відмовостійкості ПС, що дає змогу суттєво зменшити кількість помилок, оскільки алгоритм превентивно ізолює шкідливих користувачів (що, у свою чергу, збільшує пропускну здатність ПС). Удосконалений метод може бути використаний у будь-якому розподіленому додатку для покращення ефективності його роботи.

Достовірність та обґрунтованість отриманих результатів підтверджується використанням у процесі дослідження таких прийомів:

– перевірка нових та удосконалених рішень експериментальними дослідженнями за допомогою відомих процедур проектування та тестування;

– доведення ефективності вдосконаленого методу забезпечення відмовостійкості ПС та працездатності і функціональної придатності розробленого на його основі програмного засобу;

– наявність наукової публікації у рецензованому виданні.

За результатами дослідження опублікована одна стаття у фаховому науковому виданні та тези доповіді на Міжнародній науковій конференції.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, останніх досліджень та джерел

Відмовостійкість програмних систем – це новий тренд у проєктуванні та розробці ПЗ, що має на меті реалізацію надійних та стабільних ПС.

Відмовостійкість ПС – це властивість програмної системи, яка визначає, наскільки добре система може підтримувати безперебійну роботу своїх критичних модулів за наявності таких несприятливих подій, як відмова обладнання, програмного забезпечення та кібератаки. Ця точка зору охоплює три ідеї:

– деякі модулі та функціонал, що виконуються системою, є критичними, відмова яких може мати серйозні людські, соціальні чи економічні наслідки;

– деякі події є руйнівними і можуть вплинути на здатність системи надавати свої критичні послуги;

– відмовостійкість – це лише припущення (тобто немає показників стійкості, і стійкість не можна виміряти);

– відмовостійкість системи здатні оцінити лише експерти, які можуть вивчити систему та її операційні процеси.

При проєктуванні відмовостійкості ПС увага приділяється в основному обмеженню кількості системних збоїв, що виникають внаслідок зовнішніх подій, таких як кібератаки, проблеми із мережею, помилки користувача тощо.

У виявленні системних проблем та відновленні після них використовують наступні чотири підходи та активності:

– ПС автоматично або її оператори (вручну) повинні розпізнавати відмови системи на ранніх етапах;

– якщо ознаки проблеми (чи кібератаки) виявляються завчасно, тоді можна застосувати стратегії протидії, щоб зменшити вірогідність того, що система вийде з ладу;

– якщо виникає збій, операція відновлення забезпечує швидке відновлення критичних системних сервісів, щоб збій не вплинув на користувачів системи;

– у цій останній діяльності всі системні функції відновлюються, і звичайна робота ПС повинна продовжуватися.

Загалом ПС можна вважати відмовостійкою, якщо вона може здійснювати свої основні функції, незважаючи на несприятливі події. «Бути відмовостійкою» – це важлива характеристика, оскільки немає значення, як добре система розроблена та які технології використані для реалізації завдання; рано чи пізно система може зіштовхнутися з реальними подіями, які можуть призвести навіть до колапсу.

До прикладу, залишкові проблеми у програмному або апаратному забезпеченні, які не були виявлені на етапі тестування, можуть викликати часткову або повну деградацію системи або ж спричинити невідповідність системи вимогам якості (тобто до відсутності доступності, ємності, сумісності, продуктивності, надійності, безпеки, зручності використання тощо). Невідома або наявна невиправлена вразливість потенційно може закінчитись хакерською атакою. Сторонні проблеми (втрата електроенергії, перегрів серверів тощо) можуть призвести до повної недоступності сервісу [4, 5].

Зважаючи на описані умови, характеристик доступності та надійності ПС недостатньо, тому система також повинна бути відмовостійкою, а саме – мати можливість не лише виявляти часткову або повну втрату працездатності, а ще й реагувати на проблеми, продовжувати працювати у разі наявності помилок та швидко відновлюватись у разі збоїв.

Варто зазначити, що неможливо однозначно вказати, що система А є відмовостійкою, в той час як система В не є відмовостійкою. Згідно із рядом досліджень, жодна система не є відмовостійкою на 100%. Тому можна сформувати визначення, що ПС є відмовостійкою рівно до того моменту, допоки вона здатна швидко та ефективно захищати свої критичні функції від порушень, спричинених несприятливими подіями та/або умовами [6, 7].

Захист ПС включає в себе виявлення несприятливих умов та подій, реакції на них шляхом зменшення уражень, що спричиняються ними, на критичні функції та відновлення від наслідків. Попередження несприятливих подій не є частиною

відмовостійкості ПС, оскільки відмовостійкість припускає, що рано чи пізно такі події матимуть місце, та зосереджується на безперервній роботі критичних сервісів у несприятливих умовах.

Згідно з дослідженням [8] факторами, що шкодять відмовостійкості, є:

- швидкі зміни;
- розмір та складність програмної системи;
- мережева орієнтованість;
- гібридизація;
- повторне використання у різних контекстах;
- програмне забезпечення з відкритим вихідним кодом;
- взаємозалежність та взаємозв’язок;
- стрімка глобалізація компанії.

Розглянемо ці фактори детальніше.

Наприклад, поєднання двох або більше систем призводить до вищого рівня складності, ніж поєднання складності двох окремих систем. Таким чином, якщо система А має складність 5, а система С – складність 7, то комбінація систем А і С матиме складність вище ніж 12 (приблизно у діапазоні 20).

Взаємозалежність чи взаємозв’язок через все ширші мережі додає складності, тому що, коли системи стають дедалі взаємопов’язаними та взаємозалежними, досягнення стійкості стає складнішим. Ще однією з перспектив взаємозв'язку є зростання інфраструктур, що містять системи, які належать різним організаціям. Отже, стійкість системи суб’єкта господарювання все більше залежить від стійкості систем, які не належать суб’єкту господарювання.

Питання щодо мережевої орієнтованості полягає в тому, чи можна довіряти системам та мережам, які не перебувають під безпосереднім контролем організації-замовника і які докази є доступними для підтвердження такої довіри. У таких ситуаціях потрібно забезпечити надійність безпечної взаємодії компонентів ПЗ без нагляду. Слід також зазначити, що забезпечення безпеки має охоплювати стійкість та цілісність, а також конфіденційність.

В умовах стрімкої глобалізації компанії, зі зростанням дедалі розширених ланцюгів поставок для розробки ПЗ, проблема полягає у тому, що основна увага буде приділятися функціональності та низькій вартості, а не відмовостійкості.

Продукти з відкритим кодом певною мірою є програмним еквівалентом аспектів взаємозв'язку та мережевої орієнтованості, оскільки не завжди можна звернутися до конкретної групи людей, щоб забезпечити надійність і стійкість та усунути будь-які збої програми.

Гібридизація – це поєднання декількох програмних систем в одну ПС. Таким чином, виникає проблема зі з'ясуванням рівня відмовостійкості даної системи. Існує думка, що в такому випадку система є лише настільки відмовостійкою, наскільки відмовостійким є її найслабший компонент. Цей аспект варто врахувати при спробі оцінити відмовостійкість комплексної багатокомпонентної ПС.

Швидкі зміни, оновлення та випуск нових функцій ПС спричиняють незліченну кількість проблем із захистом ПС та його стійкістю. Часто бракує часу на тестування однієї версії програмного продукту до появи нової, що робить тести оригінального ПС застарілими.

Оскільки будь-яка організація керується економічними та ринковими показниками, то спостерігається тенденція до збільшення використання готового ПЗ та ПЗ з відкритим кодом. Хоча такі системи можуть бути розроблені для роботи у конкретному середовищі, вони все частіше використовуються у ситуаціях, для яких не були розроблені. Як результат, системи можуть не відповідати вимогам безпеки та відмовостійкості у нових умовах.

1.2 Аналіз існуючих методів та засобів забезпечення відмовостійкості програмних систем

Одним із методів відмовостійкості ПС є перехід на резервні системи, що відбувається у випадку відмов в основній системі. Перехід на інші системи може

відбуватись на власні або зовнішні резервні системи. Перехід на інші системи у рамках власних резервів має на меті забезпечення безперервного функціонування ПС. В той же час перехід на зовнішні резервні системи може бути «гарячим», «теплим» або «холодним». У разі «гарячого» переходу резервна система працює паралельно з основною, і, при виявлення помилок в основній системі, перехід на резервну систему відбувається автоматично і миттєво. Також при такому підході до відмовостійкості ПС є певні обмеження у доступності та віддаленістю між резервною та основною системами. Ще одним ключовим обмеженням є можливість оперувати даними, використовуючи дві паралельні системи.

Комплексний підхід до відмовостійкості ПС включає у собі чотири базові методи реалізації відмовостійкості, а саме:

– реакція на помилки (як тільки виникає помилка, ПС у автоматичному режимі або розробники програми у ручному режимі повинні відреагувати на помилки);

– автоматичне логування (ПС має автоматично логувати усі проблеми та помилки, що виникають в системі, для можливості автоматичного реагування на проблеми);

– аналіз та покращення значень метрик MTBF, MTTF, MTTR (ці метрики визначають середній час між відмовами, середній час до відмови, середній час на виправлення відмови);

– автоматизація плану відновлення (необхідно автоматизувати кроки, які система здійснить для відновлення нормальної роботи) [9].

ENISA пропонує враховувати дві метрики для визначення відмовостійкості ПС згідно з формулами (1.1) та (1.2) [10].

$$Operational\ MTBF\ =\ \Sigma\ Period\ \div\ number\ of\ failures, \tag{1.1}$$

де *Operational MTBF* – час працездатності між відмовами, що вимірюється у секундах, хвилинах, годинах тощо;

Period – час між початком відмови до вирішення проблеми, що вимірюється у секундах, хвилинах, годинах тощо;

number of failures – кількість відмов.

$$Operational\ reliability\ =\ e^{\,-t\ /\ Operational\ MTBF}, \tag{1.2}$$

де *Operational reliability* – ймовірна надійність;

t – час, протягом якого система повинна функціонувати без відмов;

Operational MTBF – метрика, згадана вище, яка визначає час працездатності між відмовами.

ENISA визначає, що як тільки значення метрики ймовірної надійності менше за e^{-1} (= 0,3678 = 1/*e*), то це означає, що система безвідмовно працює довше ніж зазвичай. У разі, якщо не було зроблено жодних заходів для підвищення надійності та відмовостійкості ПС, це означатиме наступне: існує обернено пропорційна до *Operational reliability* ймовірність, що у найближчий час трапиться відмова [10].

Таким чином, вказані метрики дають можливість визначити відмовостійкість ПС та припустити ймовірність відмови у даний момент часу.

Як правило, науковці використовують ці метрики для дослідження повних відмов системи. Але на практиці трапляються випадки, коли критичні частини ПС все ще залишаються працездатними у той час, коли низькопріоритетні частини продукту можуть працювати некоректно або не працювати взагалі. Для прикладу, в Інтернет-магазині може працювати з перебоями сервіс, який відповідає за автоматичну відправку рекламних SMS-повідомлень; у той же час авторизація користувачів, кошик, пошук та інші важливі сервіси працюють безперебійно.

Також часто трапляється, коли частина системи не працює через помилки та проблеми у сторонній бібліотеці. Наприклад, сервіс авторизації використовує два способи авторизації за допомогою Gmail та за допомогою Facebook. У випадку проблем з бібліотекою Facebook сервіс авторизації буде частково непрацездатним.

Саме тому виникає потреба проаналізувати і розробити підходи та методи визначення часткової непрацездатності сервісів та, за потреби, переадресовувати користувачів на робочий сервіс. Наприклад, вивести користувачу повідомлення «Зараз є проблеми з авторизацією за допомогою Facebook. Будь ласка, авторизуйтесь за допомогою Gmail або спробуйте пізніше» або тимчасово приховати від користувачів кнопку авторизації за допомогою Facebook.

1.3 Методологічні підходи до вирішення задачі удосконалення відмовостійкості програмних систем

Важливим етапом у забезпеченні відмовостійкості ПС є правильне проєктування архітектури системи. Сучасні дослідження стверджують, що мікросервісна архітектура дає можливість проектувати системи, які здатні реагувати на часткові відмови [11], оскільки при використанні монолітного архітектурного підходу відмови спричиняють повну відмову системи. Завдяки використанню мікросервісного підходу забезпечується ізольованість кожного сервісу, і відмова одного із сервісів не блокує використання інших сервісів ПС. Також перевагою мікросервісного підходу над монолітним є те, що вирішення відмови не потребує повторного розгортання нової версії всього програмного продукту, а лише його чвстини. Таким чином, у випадку, коли розгортання ПС відбувається з очікуваним простоєм (down time), то це зачіпає роботу лише одного сервісу, а не всієї системи. В той же час, якщо після розгортання ПС з’явились відмови, то, знову ж таки, повернення до попередньої робочої версії відбуватиметься швидше за рахунок того, що повертаються лише проблемні сервіси, а не вся система.

Однак, мікросервісна архітектура ПС не гарантує її 100%-ву надійність та відмовостійкість, тому також необхідно забезпечити відмовостійкість шляхом програмної реалізації реагування на відмови, тобто використати так звані патерни (алгоритми) відмовостійкості ПС. Оскільки у будь-який момент часу може трапитись відмова у будь-якому сервісі, необхідно, щоб інші сервіси автоматично реа-

гували на наявну проблему. Тому наступним етапом є розробка стратегії реагування на відмови з використанням патернів відмовостійкості ПС. Загалом, станом на сьогоднішній день існують наступні стратегії реагування на відмову ПС:

– припинення виконання запитів у випадку появи помилок;

– повторення запиту через деякий час;

– використання даних за замовчуванням, якщо не вдалось отримати нові дані;

– лімітування кількості виконаних запитів;

– використання ідемпотентних операцій*, щоб уникнути помилок та неконсистентного стану системи.

Проте описані стратегії мають низку своїх недоліків. Наприклад, припинення виконання запитів при появі помилок може призвести до випадкового відключення працездатного сервісу. Тому помилки слід розділяти на очікувані та неочікувані. Наприклад, у випадку введення користувачем невірного паролю виведення помилки є правильною реакцією системи, тому відключення функціоналу системи є невиправданим; у той же час отримання помилки на введення правильного логіну та паролю є неочікуваною поведінкою системи. До того ж, варто враховувати, як часто трапляються помилки та скільки користувачів від них постраждали. Наприклад, користувач А завантажує на сервер файл розміром 100 МБ, ресурси системи успішно виконують запит; в той же час користувачі Б та С завантажують файли розміром 100 ГБ та 115 ГБ відповідно та отримують помилки, оскільки сервіси системи не розраховані на таке навантаження. Відключення функціоналу у випадку, коли помилки отримують лише певна група користувачів, призведе до того, що решта користувачів не зможуть використовувати ПС. Оптимальним вирішенням описаної проблеми є ізоляція тих користувачів, які отримують помилки, у так званий «карантин», щоб вони не завадили використовувати програмний продукт іншим користувачам.

Повторення запиту через деякий час не є вирішенням проблеми, оскільки,

---

* Термін «ідемпотентність» означає властивість, яка проявляється у тому, що повторна її дія над будь-яким об’єктом уже не змінює результату. Тобто повторне виконання операцій з об’єктом не змінює результату, досягнутого при першому виконанні.

якщо є проблема у модулі системи і вона досі не вирішена, то повторні запити призведуть до перевантаження сервісу та вичерпання доступних ресурсів, що, у свою чергу, призведе до повторної відмови ПС.

Використання даних за замовчуванням має на меті збереження даних в альтернативному сховищі (або у коді самої програми) інформації, яку необхідно буде використати у разі, якщо модуль системи, що повертає дані, є недоступним. Такий підхід має наступні недоліки.

1) Цей підхід працює лише для певних функцій ПС. Його не можна застосувати до таких критичних функцій ПС як, наприклад, авторизація (оскільки система не знає наперед даних користувача), отримання ціни товару тощо. Цей підхід працює лише на низькопріоритетних та статичних функціях (наприклад, виведення способу доставки), що, як правило, не реалізується окремими сервісами, запитами тощо.

2) У разі використання тимчасових сховищ даних (таких, наприклад, як кеш) зростає навантаження на пам’ять, що, у свою чергу, гальмує систему, оскільки витрачаються ресурси на ті операції та дані, які, ймовірно, не використовуватимуть користувачі системи.

Лімітування кількості запитів має на меті фіксування та обмеження кількості запитів за певний період. Наприклад, система дозволяє користувачам надіслати не більше 100 запитів на авторизацію один раз на 15 хвилин; у разі вичерпання заданого ліміту система буде повертати код відповіді 429. Розглянемо на прикладі, як буде працювати цей підхід у модульному середовищі для студентів та викладачів. Припустимо, що кілька груп студентів будуть складати іспит о 8:00. Кожен із них намагається авторизуватись на сайті, починаючи із 7:50 до 8:05 включно. У такому випадку система дозволить пройти авторизацію першій сотні студентам, а решта отримають помилку і змушені будуть чекати, доки система не оновить лічильник період. У той же час, якщо відбуватиметься хакерська атака в будь-який момент часу, система ніяким чином не буде блокувати запити, що надходять від хакерів, та ніяк не буде ізолювати їх від реальних користувачів системи. Тому можна виділити наступні недоліки такого методу:

– відсутність автоматичного реагування та адаптації у реальному часі до збільшення або зменшення навантаження;

– відсутність механізму блокування чи попередження помилок, що надходять від потенційно небезпечних користувачів.

Використання ідемпотентних операцій базується на ігноруванні повторних запитів з тими ж параметрами, якщо хоча б один з цих запитів був виконаний успішно. Наприклад, у системі існує два модулі: перший модуль відповідає за баланс користувача, а інший – за оплати та виведення коштів із системи. Якщо модуль, який відповідає за баланс користувача, використовує ідемпотентні операції, то тоді запит модуля оплати виконається лише один раз. Наприклад, користувач поповнює свій баланс, модуль оплати надсилає запит до модуля балансу, але той не доступний. Модуль оплати повторює запит з тими ж параметрами вдруге, і знову не отримує відповіді. Оскільки баланс є ідемпотентним модулем, то після відновлення він нарахує кошти лише один раз, а решту повторних запитів проігнорує. У випадку, якщо модуль балансу не є ідемпотентним, він може опрацювати всі запити, які були надіслані, та нарахувати кошти не один раз, а стільки разів, скільки запитів було надіслано.

Враховуючи вищезгадані недоліки у наявних стратегіях забезпечення відмовостійкості ПС, можна сказати, що існує потреба реалізувати новий підхід до відмовостійкості ПС, який би вирішував наявні проблеми та давав змогу зменшити негативний вплив на систему (у тому числі і від часткових відмов).

Для вирішення описаних проблем можна застосувати підхід ізоляції та тимчасового відключення непрацездатних частин системи; у такому разі негативний вплив наявної відмови буде нижчим, оскільки користувачі не зможуть продовжувати використовувати непрацюючий функціонал, що зменшить використання критично важливих ресурсів системи (таких як пам’ять та процесор).

Таким чином, модель удосконалення алгоритму відмовостійкості ПС із покриттям часткових відмов має складатися, як мінімум, з двох частин:

– визначення та тимчасова ізоляція користувачів, які отримують помилки, та потенційно небезпечних користувачів;

– визначення та тимчасова ізоляція сервісів ПС, які дали «збій».

Подальші дослідження у цьому напрямку полягають у проєктуванні та програмній реалізації відповідних методів та алгоритмів забезпечення відмовостійкості ПС.

1.4 Висновки. Постановка задачі

У зв’язку зі стрімкою діджиталізацією виникає потреба забезпечити надійність та відмовостійкість ПС. А це, у свою чергу, породжує потребу в алгоритмах відмовостійкості, які нададуть можливість підвищити працездатність ПС та зменшити негативний вплив на них після настання відмов. У зв’язку з тим, що на сьогоднішній день неможливо розробити 100%-во відмовостійкі ПС, є потреба у використанні методів, які забезпечують максимальний рівень їх відмовостійкості.

Оскільки концепція відмовостійкості ПС та алгоритмів відмовостійкості є новою, ця сфера має низку невирішених проблем, наприклад, таких як:

– відсутність врахування та обробки часткових відмов ПС;

– неспроможність зменшити кількість помилок у реальному часі;

– відсутність спроможності вирізняти першопричину помилки.

Таким чином, виникає потреба у вдосконаленні наявних методів відмовостійкості ПС, що дасть змогу аналізувати, реагувати та зменшувати негативний вплив часткових відмов у ПС.

Тому завданням магістерської роботи є:

– визначення моделі забезпечення максимальної відмовостійкості ПС з покриттям часткових відмов;

– забезпечення зменшення негативного впливу відмов шляхом розширення та комбінації існуючих патернів відмовостійкості ПС;

– аналіз результатів роботи удосконаленого алгоритму відмовостійкості ПС та визначення його ефективності у порівнянні з існуючими рішеннями.

Актуальність роботи полягає у тому, що в будь-якій програмній системі хоча би один раз траплялись відмови, тому необхідно використовувати алгоритми відмовостійкості ПС.

Об’єкт дослідження – відмовостійкість програмних систем.

Предмет дослідження – методи, алгоритми та засоби забезпечення відмовостійкості ПС.

Мета дослідження – удосконалення методу відмовостійкості ПС та розробка програмного засобу на його основі, що дасть змогу обробляти та зменшувати негативний вплив відмов ПС (у тому числі часткових).

Відповідно до мети можна виділити наступні задачі дослідження:

– проаналізувати (у загальному) сферу відмовостійкості ПС;

– дослідити сучасні підходи, концепції та методи забезпечення відмовостійкості ПС з метою виявлення наявних проблем та невирішених питань;

– удосконалити наявні методи забезпечення відмовостійкості ПС для вирішення (або часткового вирішення) виявлених проблем;

– на основі удосконаленого методу відмовостійкості ПС виконати проектування відповідного програмного засобу;

– виконати програмну реалізацію удосконаленого методу відмовостійкості ПС;

– провести тестування та практичну апробацію отриманих результатів;

– дослідити ефективність запропонованих рішень;

– проаналізувати отримані результати дослідження та сформувати рекомендації щодо доцільності їх впровадження.

Також визначено основні вимоги до вдосконаленого алгоритму забезпечення відмовостійкості ПС:

– забезпечення здійснення припущень про першопричину помилки;

– автоматичне зменшення кількості помилок, які виникають у системі.

Наступним етапом роботи є аналіз концепцій, моделей та методів вирішення поставленої задачі, визначення моделей поведінки класичних патернів відмовостійкості ПС та визначення того, які моделі і як саме буде вдосконалено.

2 КОНЦЕПЦІЇ, МОДЕЛІ ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1 Концепції відмовостійкості програмних систем

Традиційні патерни та концепції відмовостійкості ПС працюють над відновленням працездатності ПС після виявленої відмови; в той же час можна розробити превентивну модель поведінки системи (наприклад, у випадку виявлення підозрілої поведінки користувача).

Розглянемо детально суть та реалізацію сучасних патернів відмовостійкості програмних систем.

Патерн Перегородка – це тип дизайну програми, який є толерантним до відмов ПС. В архітектурі патерну Перегородка елементи програми ізолюються в пули для того, щоб, якщо один з них вийшов з ладу, інші елементи продовжували функціонувати. Цей патерн названий на честь секційних перегородок корпусу корабля (якщо корпус корабля пошкоджено, то лише пошкоджена ділянка заповнюється водою, що запобігає потопленню корабля [12]).

Контекст і проблема

Хмарний додаток може включати декілька сервісів, причому, кожен сервіс має одного або більше користувачів. Надмірне навантаження або вихід з ладу сервісу вплине на всіх споживачів сервісу.

Більше того, споживач може одночасно надсилати запити до кількох сервісів, використовуючи ресурси для кожного запиту. Коли споживач надсилає запит до сервісу, який неправильно налаштований або не відповідає на запит, ресурси, використані за запитом клієнта, можуть бути не звільнені вчасно. Оскільки запити до сервісу продовжуються, ці ресурси рано чи пізно можуть бути вичерпані (наприклад, може бути вичерпаний клієнтський пул з’єднань). У цей момент це впливає на запити користувача до інших сервісів. Зрештою, користувач більше не зможе надсилати запити не лише до оригінального сервісу (який не відповідає), а й до інших сервісів.

Те саме питання вичерпання ресурсів стосується і сервісів з багатьма користувачами. Значна кількість запитів від одного клієнта може вичерпати

доступні ресурси сервісу, унаслідок чого інші користувачі більше не зможуть використовувати сервіс, що й спричиняє каскадний ефект відмови.

Рішення

Рішенням є розділення екземплярів сервісів на різні групи, виходячи з вимог користувача щодо завантаження та доступності. Ця конструкція допомагає відокремити несправності та дозволяє підтримувати функціональність сервісу для частини користувачів навіть під час відмови.

Користувач може також розділити ресурси, щоб гарантувати, що ресурси, які використовуються для виклику одного сервісу, не впливають на ресурси, які використовуються для виклику іншого сервісу. Наприклад, користувачу, який викликає декілька сервісів, може бути призначений пул з'єднань для кожного сервісу. Якщо сервіс починає виходити з ладу, то це впливає лише на пул підключень, призначений саме для цього сервісу, дозволяючи користувачу продовжувати користуватися іншими сервісами.

Перевагами цієї моделі є наступні:

– ізоляція користувачів та сервісів від каскадних збоїв ПС (проблема, що зачіпає користувача чи сервіс, може бути відокремлена у межах її власної «перегородки», запобігаючи відмові всього рішення);

– збереження деяких функціональних можливостей у випадку відмови сервісу (інші сервіси та особливості програми продовжуватимуть працювати);

– розгортання сервісів, які пропонують іншу якість обслуговування для споживаючих програм (тобто пул користувачів з високим пріоритетом можна налаштувати на використання пріоритетних сервісів).

На рисунку 2.1 показані перегородки, структуровані навколо пулів з'єднань, що викликають окремі сервіси. Якщо сервіс А «зламано» або він спричиняє деяку іншу проблему, то пул підключень ізольований, тому це впливає лише на робочі навантаження, які використовують пул потоків, призначені для сервісу А. Робочі навантаження, які використовують сервіси В і С, є недоторканими і можуть продовжувати працювати без перерви.

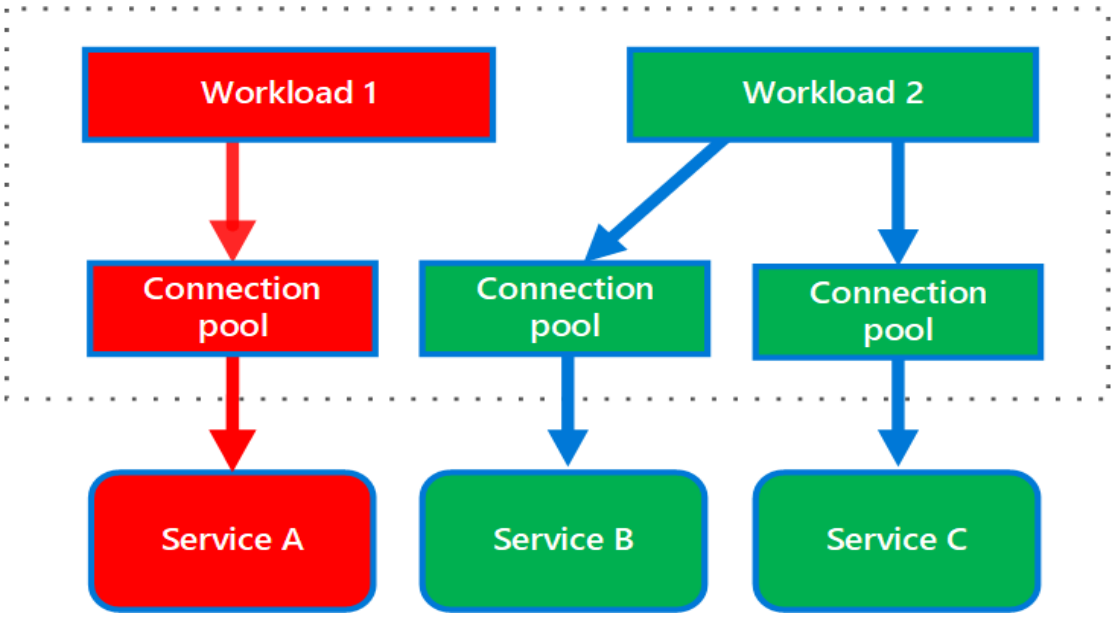


Рисунок 2.1 – Перша діаграма шаблону Перегородка

На рисунку 2.2 показано, як декілька клієнтів викликають один сервіс. Кожному клієнту присвоюється окремий екземпляр сервісу. Клієнт 1 зробив занадто багато запитів та перевантажив свій екземпляр. Оскільки кожен екземпляр сервісу ізольований від інших, то інші клієнти можуть продовжувати здійснювати запити.

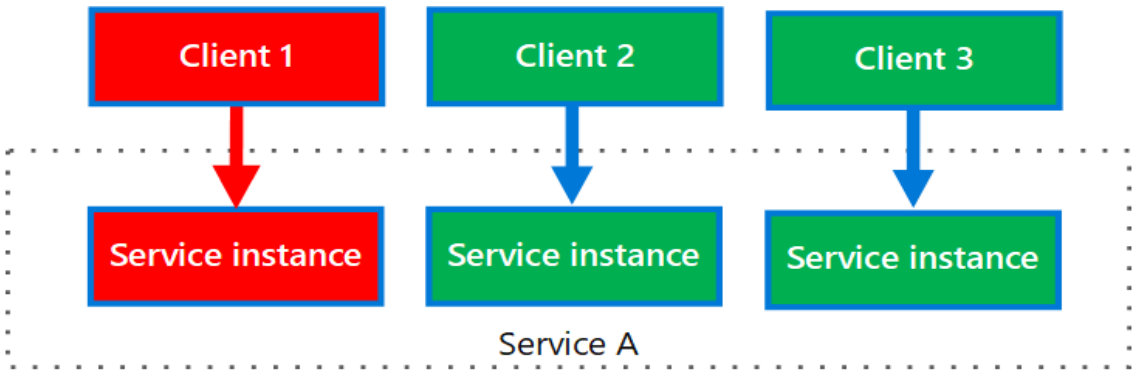


Рисунок 2.2. Діаграма запитів кількох клієнтів до одного сервісу

Шаблон Вимикач (Circuit Breaker)

Цей шаблон обробляє несправності, для вирішення яких може знадобитись різний час при підключенні до віддаленого сервісу чи ресурсу. Це може покращити стабільність та стійкість ПС [13].

Контекст і проблема

У розподіленому середовищі запити до віддалених ресурсів та сервісів можуть не відбуватися через тимчасові несправності (такі, наприклад, як повільне підключення до мережі, таймаути чи тимчасова недоступність ресурсів тощо). Ці несправності зазвичай виправляються через короткий проміжок часу, і надійний хмарний додаток повинен бути готовий обробляти їх, використовуючи таку стратегію, як патерн повторних запитів.

Однак можуть бути ситуації, коли несправності виникають через непередбачувані події, і це може зайняти набагато більше часу. Ці несправності можуть варіюватися від серйозності від часткової втрати зв’язку до повної відмови сервісу. У цих ситуаціях може бути безглуздо для програми постійно повторювати операцію, яка навряд чи буде успішною; натомість програма повинна швидко визнати, що операція не вдалася, і обробляти цю помилку відповідним чином.

Окрім того, якщо сервіс зайнятий, збій в одній частині системи може надалі призвести до каскадних збоїв. Наприклад, операція, яка викликає сервіс, може бути налаштована на реалізацію тайм-ауту та відповісти повідомленням про помилку, якщо сервіс не зможе відповісти протягом цього періоду. Однак ця стратегія може спричинити блокування багатьох одночасних запитів до тієї самої операції до закінчення періоду очікування. Заблоковані запити можуть використовувати критично важливі системні ресурси (пам’ять, потоки, підключення до БД тощо). Ці ресурси можуть вичерпатися, спричинивши вихід з ладу інших, можливо, не пов’язаних частин системи, які потребують використання тих самих ресурсів. У цих ситуаціях було б бажано, щоб операція негайно провалилася і намагалася викликати сервіс лише у тому випадку, якщо це, ймовірно, вдасться. Встановлення коротшого часу очікування може допомогти вирішити цю проблему, але час очікування не має бути настільки коротким, щоб у більшості випадків операція давала збій, навіть якщо запит до сервісу врешті-решт вдався.

Рішення

Шаблон Вимикач, який популяризував Майкл Найгард у роботі «Випустіть це!» [14], може перешкодити програмі повторно намагатися виконати операцію,

яка, ймовірно, не вдасться. Це дозволяє ПС продовжувати роботу, не чекаючи виправлення несправності та не витрачаючи ресурси системи, якщо несправність є довготривалою. Шаблон Вимикач також дозволяє програмі виявляти, чи була усунена несправність. Якщо проблема (здається) була виправлена, то програма може спробувати повторно викликати операцію.

Призначення шаблону Вимикач відрізняється від шаблону Повторної спроби. Шаблон Повторень дозволяє програмі повторити спробу операції, очікуючи, що вона успішно виконається. Шаблон Вимикач заважає програмі виконувати операцію, яка, ймовірно, не вдасться. Додаток може поєднувати ці два шаблони, використовуючи шаблон Повторень, щоб викликати операцію через автоматичний Вимикач. Однак логіка повторної спроби повинна бути чутливою до будь-яких помилок, повернутих Вимикачем, і відмовлятися від спроб повторної спроби, якщо Вимикач вказує, що несправність не є тимчасовою.

Вимикач діє як проксі-сервер для операцій, які можуть вийти з ладу. Проксі повинен відстежувати кількість останніх помилок, які сталися, і використовувати цю інформацію для того, щоб вирішити – дозволити продовження операції чи просто негайно повернути помилку.

Проксі може бути реалізований як автомат стану з наступними станами, що імітують функціональність електричного вимикача:

- закритий;
- відкритий;
- напіввідкритий.

Закритий стан означає, що запит від програми перенаправляється на сервіс. Проксі підтримує підрахунок кількості останніх помилок, і, якщо виклик операції не вдався, проксі збільшує цей лічильник. Якщо кількість останніх помилок перевищує вказаний поріг протягом певного періоду часу, проксі-сервер переходить у відкритий стан. На цьому етапі проксі запускає таймер тайм-ауту, і коли цей тайм-аут закінчується, проксі переходить у напіввідкритий стан. Призначення таймера тайм-ауту – дати системі час на усунення проблеми, яка спричинила збій, перш ніж дозволити програмі повторно спробувати виконати операцію.

Відкритий стан означає, що запит від програми негайно завершується помилкою, і до програми повертається повідомлення про помилку.

Напіввідкритий стан означає, що обмежена кількість запитів програми може проходити вдало та викликати операцію. Якщо ці запити є успішними, то передбачається, що несправність була виправлена, і вимикач перемикається у закритий стан (лічильник несправностей скидається). Якщо будь-який запит не вдається, автоматичний вимикач припускає, що несправність все-таки має місце, тому він повертається назад у стан Відкритий та перезапускає таймер очікування, щоб дати системі додатковий проміжок часу для відновлення після несправності.

Напіввідкритий стан корисний для запобігання раптовому перевантаженню сервісу запитами. Коли сервіс відновлюється, то він може підтримувати обмежений обсяг запитів, доки відновлення не завершиться, але допоки триває відновлення, потік запитів може призвести до того, що сервіс знову вийде з ладу.

Таким чином, шаблон Вимикач забезпечує стабільність тоді, як система відновлюється після «поломки», та мінімізує вплив на продуктивність ПС. Це може допомогти зберегти час відгуку системи, швидко відхиляючи запит на операцію, яка, ймовірно, не вдасться, замість того, щоб чекати, допоки операція закінчиться (або ніколи не виконається). Якщо Вимикач викликає подію щоразу, коли він змінює стан, цю інформацію можна використовувати для контролю стану системи, захищеної автоматичним вимикачем, або для попередження адміністратора, коли автоматичний вимикач переходить у відкритий стан.

Шаблон Компенсації транзакцій

Цей шаблон скасовує роботу, виконану за допомогою декількох кроків, які в сукупності визначають узгоджену (у кінцевому рахунку) операцію, якщо один або декілька кроків не вдалися. Операції, які використовують модель узгодженості в кінцевому рахунку, зазвичай, зустрічаються в розміщених у хмарі додатках, які реалізують складні бізнес-процеси чи робочі процеси [15].

Контекст і проблема

Програми, які працюють у хмарі, часто змінюють дані. Ці дані можуть поширюватися між різними джерелами даних, які зберігаються у різних

географічних місцях. Щоб уникнути суперечок та покращити продуктивність у розподіленому середовищі, програма не повинна намагатися забезпечити високу узгодженість транзакцій – радше, програма повинна забезпечувати узгодженість. У цій моделі типова бізнес-операція складається з низки окремих етапів. Поки ці кроки виконуються, загальний вигляд стану системи може бути суперечливим, але коли операція завершиться і всі кроки будуть виконані, то система повинна знову стати узгодженою.

Якщо операція, яка реалізує можливу узгодженість, охоплює декілька різнорідних сховищ даних, скасування кроків в операції вимагатиме відвідування кожного сховища даних по черзі. Робота, яка виконується у кожному сховищі даних, повинна бути надійно відмінена, щоб запобігти невідповідності системи.

Не всі дані, на які впливає операція, що реалізує можливу узгодженість, можуть зберігатися у БД. У середовищі архітектури, орієнтованої на сервіси (SOA), операція може викликати певну дію у сервісі та спричинити зміну стану, який зберігається цим сервісом. Для того, щоб скасувати операцію, цю зміну стану також потрібно скасувати. Це може включати повторний виклик сервісу та виконання іншої дії, яка скасовує ефекти першої.

Рішення

Рішення полягає у здійсненні компенсаційної транзакції. Етапи компенсаційної транзакції повинні скасувати наслідки кроків у початковій операції. Компенсаційна транзакція може бути не в змозі просто замінити поточний стан станом, у якому перебувала система на початку операції, оскільки такий підхід може замінити зміни, внесені іншими паралельними екземплярами програми. Натомість, це повинен бути розумний процес, який враховує будь-яку роботу, виконану одночасно екземплярами. Цей процес, як правило, залежить від програми, яка, у свою чергу, залежить від характеру роботи, виконаної початковою операцією.

Типовим підходом є використання робочого процесу для реалізації узгодженої у кінцевому рахунку операції, яка вимагає компенсації. У процесі початкової операції система записує інформацію про кожен крок і про те, як

роботу, виконану цим кроком, можна скасувати. Якщо в будь-який момент операція не вдається, то робочий процес повертається назад через завершені кроки та виконує роботу, що змінює кожен крок. Компенсаційній транзакції може не знадобитися відмінити роботу у точному зворотному порядку початкової операції, і, можливо, можна виконати паралельне скасування деяких кроків.

Компенсаційна транзакція також є узгодженою (у кінцевому рахунку) операцією, і вона також може провалитися. ПС повинна мати можливість відновити компенсаційну транзакцію у момент відмови і продовжувати роботу. Можливо, доведеться повторити крок, який не вдався, тому кроки у компенсаційній транзакції слід визначати як ідемпотентні команди.

Патерн Моніторингу працездатності кінцевої точки

Патерн Моніторингу реалізує функціональні перевірки у додатку, до якого зовнішні інструменти можуть отримувати доступ через відкриті кінцеві точки через рівні проміжки часу. Патерн має на меті перевірити, чи працюють програми та сервіси належним чином [16].

Контекст і проблема

Деякі програмні продукти можуть мати угоду про відсоток надійності та працездатності (SLA), тому необхідно мати можливість перевіряти працездатність системи та проактивно виправляти помилки.

Рішення

Рішення полягає у впровадженні моніторингу працездатності системи, надсилаючи запити кінцевій точці програми. Додаток повинен виконати необхідні перевірки та повернути вказівку про свій статус.

Перевірка моніторингу працездатності ПС, як правило, поєднує два фактори:

– перевірки (якщо такі є), які виконуються додатком (чи сервісом) у відповідь на запит до кінцевої точки про перевірку працездатності системи;

– аналіз результатів за допомогою інструменту або структури, що виконує перевірку працездатності системи.

Код відповіді вказує на статус програми; перевірка затримки чи часу відгуку виконується інструментом моніторингу.

Типові перевірки, які можуть виконувати інструменти моніторингу, включають наступні:

– перевірку коду відповіді (наприклад, відповідь HTTP 200 OK означає, що програма відповіла без помилок; система моніторингу може також перевіряти наявність інших кодів відповідей, щоб отримати більш вичерпні результати;

– перевірку вмісту відповіді для виявлення помилок, навіть коли повертається код стану 200 (це може виявити помилки, які стосуються лише частини повернутої веб-сторінки чи відповіді сервісу; наприклад, перевірка заголовка сторінки чи пошук певної фрази, яка вказує на правильну сторінку);

– вимірювання часу відгуку, що вказує на поєднання затримки мережі та часу, який застосував додаток для виконання запиту; зростаюче значення може вказувати на проблему, яка виникає з додатком або мережею;

– перевірка закінчення терміну дії сертифікатів SSL;

– вимірювання часу відгуку пошуку DNS для URL-адреси програми для вимірювання затримки DNS та помилок DNS;

– перевірка URL-адреси, яку повертає пошук DNS, для забезпечення правильних записів; це може допомогти уникнути зловмисного перенаправлення запиту шляхом успішної атаки на DNS-сервер.

Шардінг

Шардінг – це процес розбиття великих таблиць на менші «шматки» (шарди) або «осколки», які розподіляються на декількох серверах. Осколок – це, по суті, горизонтальний розподіл даних, який містить підмножину загального набору даних, і, отже, відповідає за обслуговування деякої частини загального робочого навантаження. Ідея полягає у тому, щоб розподілити дані, які не можуть помістится на одному вузлі, у кластер вузлів БД. Заточування також називають горизонтальною перегородкою. Різниця між горизонтальним та вертикальним розбиттям походить від традиційного табличного подання БД. БД можна розділити вертикально – зберігаючи різні стовпці таблиці в окремій базі, або горизонтально – зберігаючи рядки однієї таблиці в декількох вузлах бази [17].

Бізнес-додатки, які покладаються на монолітну СКБД, по мірі свого зростання стикаються з проблемами. В умовах обмеженого центрального процесора та обсягу пам’яті пропускна здатність запиту та час відгуку будуть страждати. Що стосується додавання ресурсів для підтримки операцій з БД, то вертикальне масштабування має власний набір обмежень і, врешті-решт, доходить до точки зменшення віддачі.

З іншого боку, горизонтальне розділення таблиці означає більшу обчислювальну здатність обслуговувати вхідні запити, і, отже, зменшується час відгуку на запити та побудови індексів. Постійно балансуючи навантаження та набір даних за додатковими вузлами, шардування також дозволяє використовувати додаткову потужність та ресурси. Більше того, мережа менших, дешевих серверів може бути економічно вигіднішою у довгостроковій перспективі, ніж підтримка одного потужного сервера.

Окрім вирішення проблем масштабування, шардінг може потенційно полегшити вплив незапланованих відключень. Під час простою усі дані у БД є недоступними, що може бути руйнівним. Якщо все зробити правильно, шардінг може забезпечити високу доступність: навіть якщо один або два вузли, на яких розміщено декілька «осколків», не працюють, решта даних БД залишиться доступною для операцій читання/запису. Загалом, шардінг може збільшити загальну ємність кластерного сховища, прискорити обробку та запропонувати вищу доступність за нижчою ціною, ніж вертикальне масштабування.

Патерн Вирівнювання навантаження на основі черги

Патерн Вирівнювання використовує чергу, яка діє як буфер між завданням та сервісом, який вона викликає, щоб згладити періодичні великі навантаження (які можуть спричинити збій сервісу або час очікування). Це може допомогти мінімізувати вплив піків попиту на доступність та швидкість реагування як для завдання, так і для сервісу [18].

Контекст і проблема

Багато рішень у хмарі включають запущені завдання, які викликають сервіси. У цьому середовищі, якщо сервіс зазнає періодичних великих

навантажень, це може спричинити проблеми з продуктивністю чи надійністю ПС.

Сервіс може бути частиною того самого рішення, що й завдання, які його використовують, або це може бути сторонній сервіс, який надає доступ до часто використовуваних ресурсів (кеш-пам’ять, сервіс зберігання тощо). Якщо один і той же сервіс використовується для низки завдань, які виконуються одночасно, то може бути важко передбачити обсяг запитів до сервісу у будь-який час.

Сервіс може відчувати піки попиту, які спричиняють його перевантаження та не можуть відповідати на запити вчасно. Перевантаження сервісу з великою кількістю одночасних запитів може також призвести до збою сервісу, якщо він не в змозі обробити конфлікт, який викликають ці запити.

Рішення

Головна ідея полягає у реорганізації рішення та введення черги між завданням та сервісом. Завдання та сервіс працюють асинхронно. Завдання розміщує у черзі повідомлення, яке містить дані, необхідні сервісу. Черга діє як буфер, зберігаючи повідомлення, допоки воно не буде отримане сервісом. Сервіс отримує повідомлення з черги та обробляє їх. Запити з низки завдань, які можуть бути сформовані зі змінною швидкістю, можуть передаватися сервісу через ту саму чергу повідомлень.

Черга відокремлює завдання від сервісу, і сервіс може обробляти повідомлення у своєму власному темпі, незалежно від обсягу запитів від одночасних завдань. Окрім того, завдання не затримується, якщо сервіс є недоступним під час надсилання повідомлення у чергу.

Цей патерн забезпечує наступні переваги.

Він може допомогти максимізувати доступність, оскільки затримки, які виникають у сервісах, не матимуть негайного та прямого впливу на додаток, який може продовжувати розміщувати повідомлення у черзі, навіть якщо сервіс є недоступним або в даний час не обробляє повідомлення.

Він також може допомогти максимізувати масштабованість, оскільки і кількість черг, і кількість сервісів можуть бути різними, щоб задовольнити попит.

Патерн може допомогти контролювати витрати, оскільки кількість розгорнутих екземплярів сервісів має бути достатньою, щоб задовольнити середнє (а не пікове) навантаження.

Деякі сервіси застосовують регулювання, коли попит досягає порогового значення, за якого система може вийти з ладу. Дроселювання може зменшити доступні функціональні можливості.

Патерн Повторень

Патерн Повторень реалізує програму для обробки збоїв, коли система намагається здійснити запит до сервісу чи ресурсу, шляхом прозорого повторення невдалої операції [19].

Контекст і проблема

Додаток, який взаємодіє з елементами, що працюють у хмарі, повинен бути чутливим до тимчасових несправностей, які можуть виникати у цьому середовищі. Серед несправностей – миттєва втрата мережевого зв’язку з компонентами та сервісами, тимчасова недоступність сервісу або час очікування, який виникає, коли сервіс зайнятий.

Ці несправності зазвичай виправляються самостійно, і якщо дія, яка спричинила несправність, повторюється після відповідної затримки, швидше за все, вона буде успішною. Наприклад, сервіс БД, який обробляє велику кількість одночасних запитів, може реалізовувати стратегію регулювання, яка тимчасово відхиляє будь-які подальші запити, допоки її робоче навантаження не зменшиться. Додаток, який намагається отримати доступ до БД, може не підключитися, але якщо він спробує знову, після затримки, це може вдатися.

Рішення

У хмарному додатку тимчасові несправності є не рідкістю, і програма повинна бути розроблена для їх елегантної та прозорої обробки. Це мінімізує вплив несправностей на завдання, які виконує програма.

Якщо програма виявляє помилку під час спроби надіслати запит віддаленому сервісу, вона може впоратися з помилкою, використовуючи такі стратегії:

– скасувати запит – якщо несправність вказує на те, що вона не є тимчасовою або навряд чи буде успішною, якщо запит повторити, то програма повинна скасувати операцію та повідомити про помилку (наприклад, помилка автентифікації, спричинена введенням недійсних облікових даних, швидше за все, повториться, незалежно від того, скільки разів це буде зроблено);

– повторити спробу (якщо конкретне повідомлення про помилку є незвичним або рідкісним, це може бути спричинено незвичними обставинами, такими, наприклад, як пошкодження мережевого пакета під час його передачі; у цьому випадку програма може негайно повторити невдалий запит, оскільки та сама помилка навряд чи буде повторена, і запит, ймовірно, буде успішним);

– повторити спробу після затримки (якщо несправність спричинена одним із найпоширеніших зв'язків або збоями мережі чи сервісу, може знадобитися деякий період, поки проблеми з підключенням будуть виправлені або затримка роботи буде усунута; програма повинна почекати деякий час перед повторною спробою запиту).

Для найпоширеніших перехідних помилок слід вибрати період між повторними спробами, щоб розподілити запити з декількох екземплярів програми якомога рівномірніше. Це зменшує ймовірність того, що зайнятий сервіс продовжуватиме перевантажуватися. Якщо багато екземплярів програми постійно переповнюють сервіс запитами повторної спроби, відновлення сервісу триватиме довше.

Якщо запит все ще не вдається, програма може зачекати та зробити ще одну спробу. За необхідності цей процес можна повторити із збільшенням затримок між повторними спробами, допоки не буде зроблена спроба встановити деяку максимальну кількість запитів. Затримку можна збільшувати поступово або експоненційно, залежно від типу несправності та ймовірності того, що вона буде виправлена протягом цього часу.

Додаток має обгортати всі спроби доступу до віддаленої служби кодом, який реалізує політику повторної спроби, що відповідає одній із перелічених вище стратегій. До запитів, які надсилаються до різних сервісів, можуть застосовуватися різні правила. Деякі постачальники надають бібліотеки, які

реалізують політики повторної спроби, де програма може вказати максимальну кількість повторних спроб, час між спробами повторних спроб та інші параметри.

Якщо сервіс часто недоступний або зайнятий, то це (часто) тому, що сервіс вичерпав свої ресурси. Можна зменшити частоту цих несправностей, масштабуючи сервіс. Наприклад, якщо служба БД постійно перевантажена, то може бути корисним розділити БД та розподілити навантаження на декілька серверів.

Підсумуємо моделі поведінки класичних патернів відмовостійкості ПС.

Описані патерні мають на меті:

– під час відмови зменшити навантаження на систему шляхом тимчасового припинення запитів або вирівнювання навантаження;

– повторне виконання запитів після паузи;

– скасування запитів, які трапились після відмови, задля повернення системи до нормального стану.

Таким чином, бачимо, що наявні рішення працюють з уже виявленими проблемами та відмовами ПС, а не намагаються їм запобігти. Описані патерни спрацьовують лише після того, як деякий сервіс втратив працездатність, але ніяк не перешкоджають підозрілій поведінці ПС та не запобігають майбутнім помилкам.

Описану проблему можна вирішити за допомогою розробки нового патерну відмовостійкості ПС, який буде аналізувати поведінку користувачів та виявляти потенційно небезпечні запити, що можуть призвести до помилки.

Таким чином, відмовостійкість сервісів підвищиться, оскільки потенційно небезпечні запити будуть блокуватись та ресурси не треба буде витрачати на відновлення нормального стану ПС (оскільки стан останньої не буде порушено).

Також традиційні методи відмовостійкості ПС не передбачають автоматичну нотифікацію розробників про наявну відмову. Тому розробники можуть не здогадуватись, що є проблеми із ПС, допоки їм про це не повідомлять користувачі.

Описану проблему можна вирішити шляхом розширення класичних методів та додаванням так званих alerting (попереджень) про підозрілу активність.

Окрім того, традиційні методи забезпечення відмовостійкості ПС не передбачають засобів моніторингу, зокрема, не зберігають аналітику про час початку

відмови, кількість відхилених автоматично запитів, кількість повторів тощо.

Таким чином, класичні методи не дають можливості детально проаналізувати та з’ясувати, що спричинило проблему, тому ці методи не дають можливості вирішити не симптом, а проблему. До того ж, неможливо провести Post Mortem – процес або зустріч, яка має на меті з’ясувати причини, наслідки та методи попередження проблеми. Внаслідок цього є великий ризик, що проблема буде повторюватись нескінченну кількість разів.

Цю проблему можна вирішити шляхом доопрацювання класичних методів забезпечення відмовостійкості ПС та додавання автоматичного моніторингу, що буде зберігати наступну інформацію:

- час спрацьовування патерну;
- користувачі, чиї запити були заблоковані;
- запити, що були заблоковані, та вхідні дані;
- кількість запитів;
- ip-адреса користувача.

2.2 Моделі та методи забезпечення відмовостійкості програмних систем

На базі описаних у підрозділі 2.1 проблем розробимо модель системи, що вирішить проблеми, які не вирішують традиційні методи відмовостійкості ПС.

Оскільки розроблювана модель має сприяти запобіганню появи відмов ПС (у тому числі часткових), то у першу чергу слід з’ясувати та вирішити, які з існуючих патернів відмовостійкості ПС потрібно доопрацювати та як.

Розглянемо способи попередження виявлених проблем. Для цього потрібно визначати появу «спайків» навантаження; отже, слід розробити алгоритм, який буде визначати причину збільшення навантаження та, у залежності від причини, обирати спосіб її вирішення. Алгоритм буде наступним:

- призначення всім користувачам унікального ідентифікатора;

– моніторинг навантаження (якщо збільшення навантаження надходить від різних ідентифікаторів, слід розподілити навантаження, щоб не допустити відмову);

– якщо збільшення навантаження надходить від одного або групи ідентифікаторів та значно перевищує середню кількість запитів, які здійснюють інші користувачі, то необхідно ізолювати цих користувачів та виділити їм окремі контейнери, щоб не спричинити відмову та недоступність сервісу для інших користувачів.

Таким чином, будуть доопрацьовані патерни Балансування навантаження та Шардінгу, що даватиме змогу не просто збільшувати ресурси, а й ізолювати потенційно небезпечних користувачів, що, в кінцевому рахунку, має забезпечити вищу відмовостійкість ПС.

Далі розглянемо способи виявлення проблем у ПС. Вирішити цю задачу можна двома способами:

– за допомогою використання патерну Моніторингу працездатності кінцевої точки;

– розробивши алгоритм, який буде, на основі виконаних користувачем запитів та на основі відповіді системи, визначати, чи є проблема.

Перший спосіб не є досконалим, оскільки метою вказаного патерну є переконатись, що кінцева точка повертає статус код «200», а не з’ясувати, як система буде поводити себе у випадку різних вхідних даних. Цей патерн не має на меті вникати у деталі реалізації, навпаки, його мета – виконати простий запит, який має успішно завершитись.

З огляду на це, слід розробити алгоритм, який буде збирати дані про:

– користувача, який здійснює запит;

– вхідні параметри запиту;

– відповідь на запит.

Таким чином, алгоритм зможе автоматично з’ясувати, де існує проблема, та застосовувати наступні способи її вирішення:

– якщо повертаються помилки лише деяким користувачам, то, отже, проблема є локальною та стосується лише цих користувачів; відповідно, слід

проаналізувати, що є спільного у групи користувачів та тимчасово ізолювати цих користувачів, надавши їм лімітовану кількість контейнерів;

– якщо повертаються помилки усім користувачам ПС, які здійснюють деякий запит, то, отже, проблема – з кінцевою точкою або сервісом; відповідно, слід тимчасово відключити доступ до сервісу за допомогою патерна Вимикач.

Для вирішення поставленої задачі необхідно реалізувати зберігання даних про користувачів та логування їх запитів, тому новий метод відмовостійкості збиратиме та аналізуватиме дані, представлені у таблиці 2.1.

Таблиця 2.1 – Вхідні дані алгоритму

Вхідний параметр	Приклад даних
IP-адреса користувача	188.123.19.208
Операційна система користувача	macOS 10.15.7
Браузер користувача	Chrome 91.0.4472.106
Запит, що здійснюється	somesite.com/login

Вхідні параметри запиту	{
	"username": “dmytro”,
	"password": "sometext"
	}

Відповідь системи на запит користувача	{
	"code": 300,
	"message": "Login failed"
	"meta info": "Uncaught PHP Exception Ramsey\Uuid\Exception\InvalidUuidStringException: "Invalid UUID string: " at /var/www/project/vendor/ramsey/uuid/src/Codec/StringCodec.php line 146"
	}

На основі даних таблиці 2.1 формуватиметься унікальний анонімний хеш-ключ, який слугуватиме як ідентифікатор користувача. Аналізуючи дані, алгоритм може автоматично припустити, де існує проблема, та спробувати автоматично ізолювати причину проблеми.

У випадку, якщо система надсилатиме помилки усім користувачам, які виконують певний запит, то тоді алгоритм здійснить припущення, що проблема є локальною та стосується лише одного сервісу чи однієї кінцевої точки, тому алгоритм автоматично вимкне даний сервіс за допомогою патерна Вимикач. Наприклад, сторінка авторизації не працює, користувачі здійснюють запити на авторизацію, проте отримують помилки; у цьому випадку алгоритм аналізує помилки і дані користувачів та робить припущення, що проблема стосується конкретного модуля системи та блокує усі подальші запити до нього, допоки проблема не буде вирішена. Схематичне відображення роботи алгоритму представлено на рисунках 2.1 та 2.2.

Рисунок 2.1 – Робота алгоритму: відкритий стан

Рисунок 2.2 – Робота алгоритму: закритий стан

Описаний підхід дозволить ізолювати проблемні частини програмної системи та дасть можливість користувачам продовжувати використовувати інші частини програми, які все ще залишаються працездатними.

У той же час алгоритм відмовостійкості та моніторинг на основі даних про користувачів можуть здійснити припущення про потребу в ізоляції користувачів та ізолювати їх від інших у випадку, якщо система почне повертати помилки лише деяким користувачам (чи групі користувачів).

Для реалізації нового підходу до балансування на основі користувачів та навантаження було вирішено переосмислити підходи шардінгу та скомбінувати їх з патерном Балансувач. Зазвичай, такий алгоритм використовується у БД для збереження даних у різних таблицях, проте, завдяки доопрацюванню, його можна використати у сучасних додатках, побудованих за кліент-серверною або мікросервісною архітектурою.

Таким чином, розроблено алгоритм, який ізолює потенційно небезпечного користувача або групу користувачів, що сприяє зниженню кількості потенційних помилок і повних відмов системи, та надає можливість іншим користувачам ПС продовжувати використовувати додаток без перешкод.

На рисунку 2.3 представлена схематична робота алгоритму. У даному випадку при розподілі навантаження та виділення користувачам конкретних сервісів A, B, C, користувачі 1 та 2 зможуть безперебійно використовувати програмний продукт.

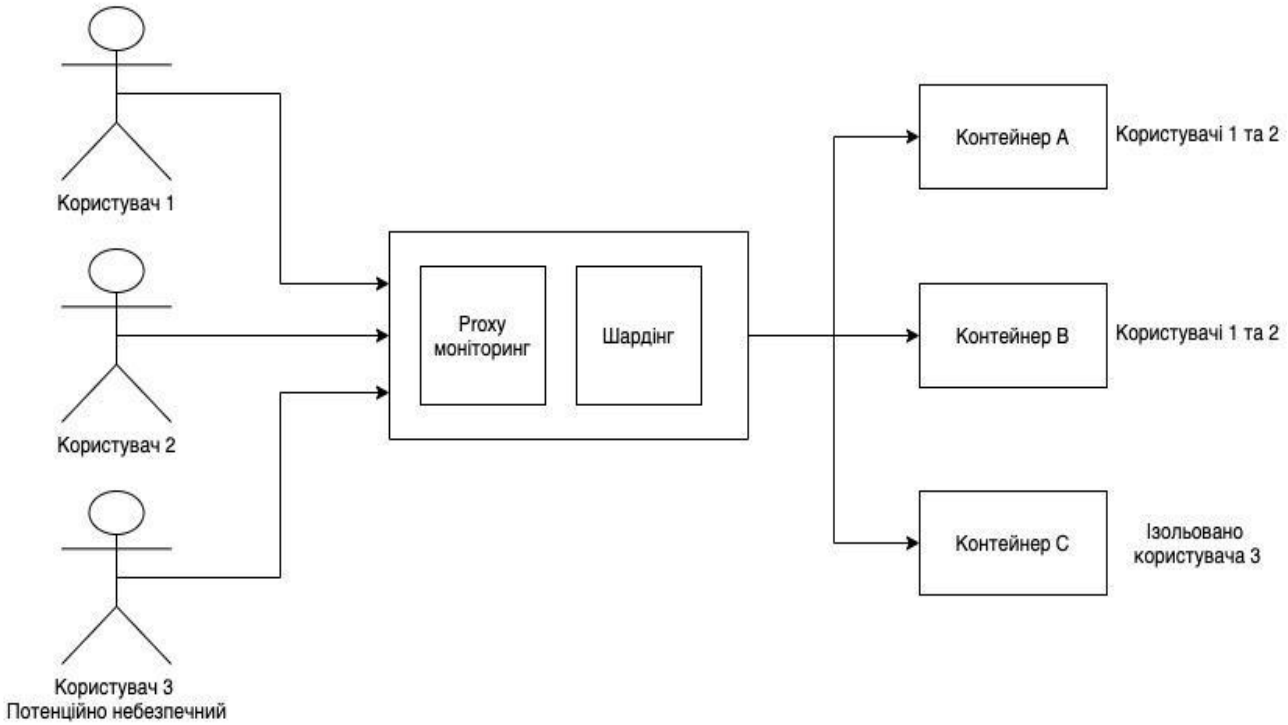


Рисунок 2.3 – Розподіл користувачів та сервісів

2.3 Висновки

У розділі здійснено аналіз моделей та методів вирішення проблем відмовостійкості. Наприклад, класичні методи відмовостійкості представляють собою прості вузькоспеціалізовані рішення, що не здатні відрізняти помилки, які сталися через несправності системи, від помилок, викликаних активністю користувача. Цю проблему запропоновано вирішити шляхом доопрацювання класичних патернів відмовостійкості ПС. Таким чином, новий алгоритм буде використовувати комплексний підхід і визначити різні типи відмов та, за необхідності, ізолювати або окремі елементи ПС, або потенційно небезпечні запити користувачів. Даний алгоритм підвищить відмовостійкість системи, оскільки система буде здатна самостійно визначити тип потенційної проблеми та реагувати відповідно.

Було розглянуто вплив інтеграції підходу Вимикача до класичного патерну Балансувача, що дозволяє зменшити кількість помилок системи за рахунок тимчасового відключення частини ПС, яка найбільше схильна до появи помилок. Це дозволяє вивільнити системні ресурси, що були задіяні на обробку запитів, які часто завершуються невдачею, що, у свою чергу, може підвищити пропускну здатність ПС.

На основі підходу шардінгу розроблено модель програмного засобу, що вирішує проблеми, які не вирішують традиційні методи відмовостійкості ПС. Це дає змогу зменшити потенційний негативний вплив одних користувачів на інших, адже користувач, який виконує дії, що призводять до помилок, не може отримати доступ до усього додатку, тож його вплив обмежується виділеним йому шардом. Також варто додати, що при активності користувача, яка призводить до постійних помилок у ПС, розмір шарду може бути зменшено таким чином, щоб ізолювати користувача, а його негативний вплив звести до мінімуму.

Наступним етапом роботи є аналіз детальних вимог до програмного засобу, його проєктування та декомпозиція, підбір та обґрунтування засобів реалізації.

3 ТЕХНОЛОГІЯ РЕАЛІЗАЦІЇ УДОСКОНАЛЕНОГО МЕТОДУ ВІДМОВОСТІЙКОСТІ ПРОГРАМНИХ СИСТЕМ

3.1 Аналіз вимог до програмного засобу

На основі аналізу предметної області, розробленого методу та алгоритмів, необхідно сформувати та описати вимоги до створюваного програмного засобу, метою розробки якого є імплементація удосконаленого методу відмовостійкості ПС. Оскільки удосконалений метод має не лише оптимізувати підходи до забезпечення відмовостійкості ПС, а й вирішити наявні проблеми, то можна виділити наступні характеристики продукту:

- реагування на повні та часткові відмови системи;
- автоматичний аналіз та сканування помилок, які з’являються у ПС;
- автоматичне висування гіпотези про причину помилок та спроби автоматичного усунення помилок.

1) Класи користувачів та їх характеристики

Система матиме лише один клас користувачів. Користувачі можуть виконувати лише вхідні запити (решта роботи виконуватиметься автоматично на основі рішень алгоритму).

2) Середовище функціонування

Середовищем функціонуванням є додаток, який побудований на основі мікросервісної або клієнт-серверної архітектури; при цьому розроблюваний програмний засіб має бути незалежним модулем.

3) Характеристики системи

Балансування навантаження

Система має автоматично визначати навантаження на додаток, використовуючи дані кожного окремого контейнера. На основі отриманих даних система має автоматично приймати рішення про збільшення чи зменшення кількості активних контейнерів.

Послідовності дія/відгук:

а) автоматична перевірка працездатності контейнерів;

б) якщо активний контейнер з деяких причин перестає відповідати на запити про активний стан, то такий контейнер вилучається зі списку доступних, а замість нього буде додано новий контейнер;

в) отримання даних про використання системних ресурсів кожним окремим контейнером;

г) якщо використання ресурсів наближається або перевищує максимальне граничне значення, то тоді відбувається додавання вказаної кількості додаткових контейнерів (граничне значення навантаження та кількість контейнерів визначаються за допомогою конфігурації);

д) якщо використання ресурсів наближається або перевищує мінімальне граничне значення, то тоді відбувається вилучення вказаної кількості активних контейнерів (граничне значення навантаження та кількість контейнерів, які вилучаються, визначаються за допомогою конфігурації);

е) запити, які надходять до додатку, розподіляються за допомогою алгоритму циклічного планування (Round-robin) поміж активними контейнерами.

Ідентифікація користувача

Дані з таблиці 2.1. мають автоматично зберігатись, і на їх основі має формуватись унікальний анонімний хеш-ключ, який слугує ідентифікатором користувача.

Послідовності дія/відгук:

- користувач здійснює запит;
- система видає анонімний хеш-ключ користувачу;
- запис отриманого хеш-ключа до HTTP-cookie.

Шардінг

Система автоматично виділяє для кожного користувача деякий перелік активних контейнерів (Shard), до яких будуть надходити усі наступні запити користувача. Кількість контейнерів у шарді може варіюватися від одного до 30% від усіх доступних контейнерів, у залежності від загальної кількості контейнерів та рейтингу надійності користувача.

Послідовності дія/відгук:

а) система зберігає доступні дані про запит та результат його виконання для кожного окремого користувача;

б) на основі даних користувача відбувається виділення доступних для нього контейнерів;

в) запити, які здійснює користувач, розподіляються між контейнерами за допомогою алгоритму циклічного планування (Round-robin), що включений у шард даного користувача.

г) якщо один з виділених користувачу контейнерів перестає функціонувати – його місце займає інший активний контейнер;

д) якщо загальна кількість доступних контейнерів зменшується, то також автоматично зменшується розмір шарду кожного користувача (але цей розмір не може бути меншим ніж два контейнери);

е) якщо загальна кількість доступних контейнерів збільшується, то також автоматично збільшується розмір шарду кожного користувача (але цей розмір не може перевищувати 30% від усіх контейнерів);

ж) на основі даних про активність користувача відбувається аналіз його надійності (якщо користувач здійснює багато запитів, які повертають помилку, та ці запити виконувалися на багатьох різних контейнерах, тоді такий користувач отримує статус «підозрілого»);

з) якщо користувач є «підозрілим», тоді кількість доступних контейнерів у його шарді зменшується до одного (це потрібно для того, щоб такий користувач якнайменше перетинався з іншими користувачами);

и) якщо повторні запити «підозрілого» користувача не викликають помилки додатку протягом деякого часу, то такий користувач знову вважається звичайним.

Тимчасове відключення функціоналу

Система повинна призупиняти виконання запитів до функціоналу, який тимчасово працює несправно, та робити запити для перевірки працездатності ПС через деякий час. У разі, якщо помилки припинились, то слід відновити нормаль-

ну роботу додатку, а якщо ні – припинити виконання запитів та здійснити повторну перевірку через деякий час.

Послідовності дія/відгук:

а) при появі помилок система має призупиняти виконання запитів до функціоналу, який працює несправно;

б) через деякий час система має зробити тестовий запит до функціоналу з метою перевірки його працездатності;

в) у разі, якщо працездатність функціоналу відновлена, то система починає поступово виконувати запити та частково повертати помилки задля уникнення настання повторної відмови;

г) через деякий час ПС починає виконувати усі запити, які надходять.

Профайлер

Система повинна записувати у журнал подій інформацію про всі виконані запити та їх результати.

Послідовності дія/відгук:

- система отримує дані про кожен запит та його результат;
- інформація про запит та його результат записується у журнал подій.

3.2 Проектування програмного засобу

3.2.1 Розробка структури програмного засобу

Розроблюваний програмний засіб має працювати як окремий самостійний модуль, тобто буде складовою будь-якої ПС. Таким чином, користувачі будуть взаємодіяти з ним не напряму, а опосередковано (тобто будь-яка ПС буде здійснювати запити на алгоритм відмостійкості).

У зв’язку з цим розроблюваний програмний засіб буде мати монолітну архітектуру та окремі модулі, які відповідатимуть за логіку роботи описаних характеристик системи:

- модуль балансувача;

- модуль шардінгу;
- модуль вимикача;
- модуль ідентифікатора;
- модуль профайлера.

Опишемо детальніше кожен з компонентів програмного засобу.

Модуль балансувача відповідає за розподіл навантаження системи, шляхом роботи з контейнерами. Цей модуль динамічно змінює кількість активних контейнерів.

Модуль шардінгу має на меті створення так званих шардів для користувача (чи групи користувачів) шляхом виділення їм окремих контейнерів, моніторинг активності користувачів та виявлення потенційно небезпечних користувачів, ізоляцію користувачів (після виявлення підозрілої активності) та роботу зі сховищем даних.

Модуль вимикача має на меті автоматично виявляти проблеми у додатку, блокувати/вимикати кінцеву точку, якщо з’являються помилки, з якими стикаються користувачі системи, та роботу зі сховищем даних.

Модуль ідентифікатора має на меті видачу унікального ідентифікатора кожному користувачу системи.

Профайлер відповідає за логування всіх опрацьованих запитів та результатів їх виконання.

Далі виконаємо декомпозицію модулів на компоненти, кожний з яких відповідатиме за конкретну логічну функцію.

На рисунку 3.1 представлена логічна структура програми у вигляді діаграми компонентів.

На рисунку 3.2 представлена діаграма послідовності, яка показує взаємодію модулів (класів) системи. На діаграмі послідовності представлений приклад обробки запиту користувача та послідовність виклику модулів системи.

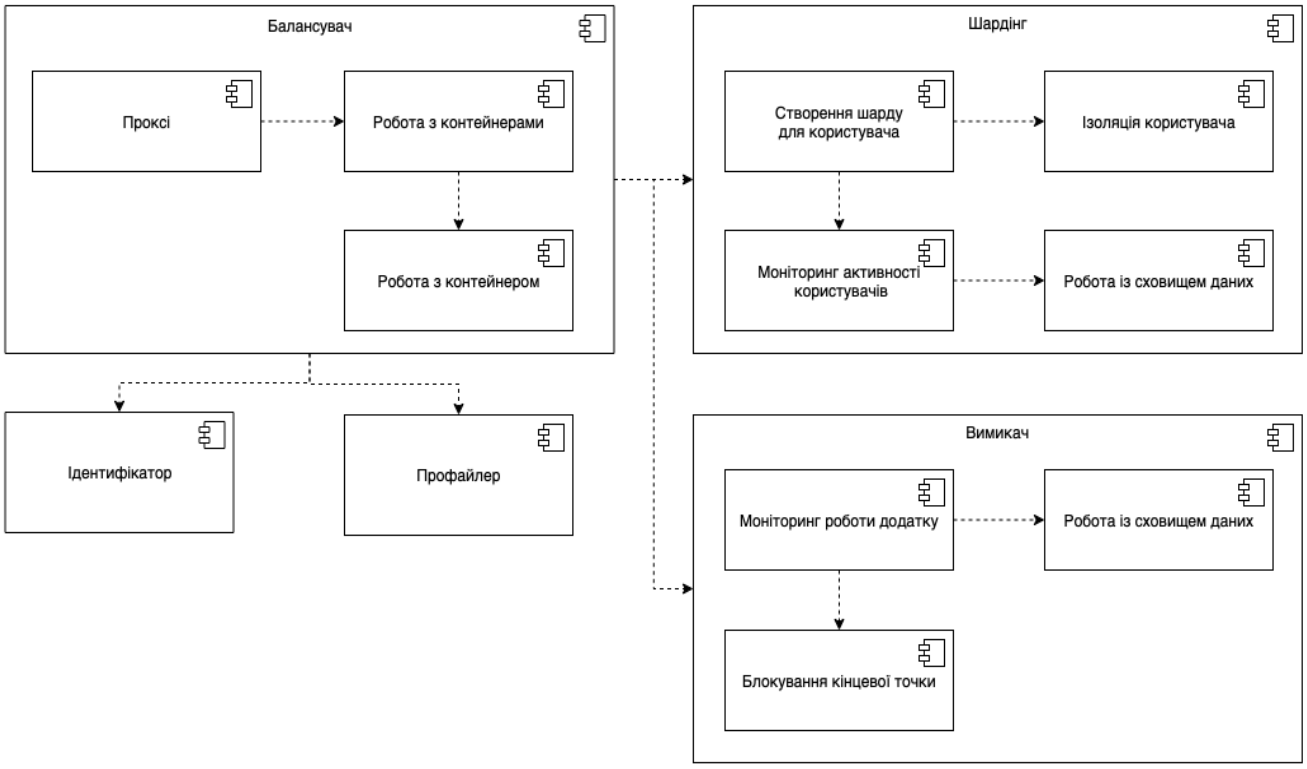


Рисунок 3.1 – Діаграма компонентів

3.2.2 Проектування структури даних

Програма оперує даними про користувача та його запити. Оскільки дані користувача та інформація стосовно його запитів належать до різних сутностей, було вирішено сформувати два окремих зв’язаних між собою об’єкти.

Дані користувача – це зашифрований анонімний хеш, що складається з наступних атрибутів:

– IP-адреса користувача;

– операційна система користувача;

– браузер користувача.

Дані запитів користувача містять наступні атрибути:

– запит, що здійснюється;

– вхідні параметри запиту;

– відповідь системи на запит користувача (запит успішно виконано чи повернулась помилка).

Схема даних представлена на рисунку 3.3.

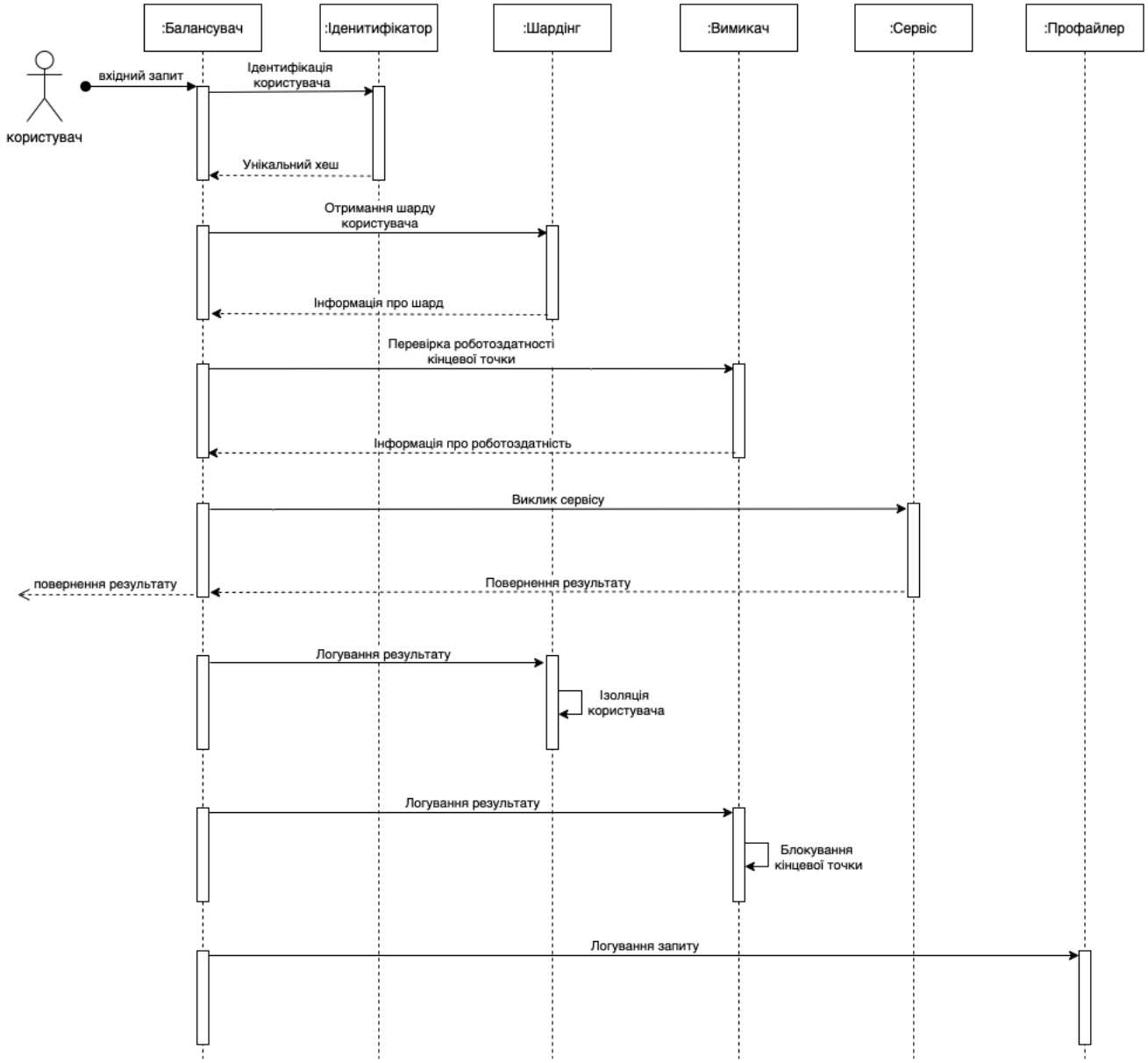


Рисунок 3.2 – Діаграма послідовності

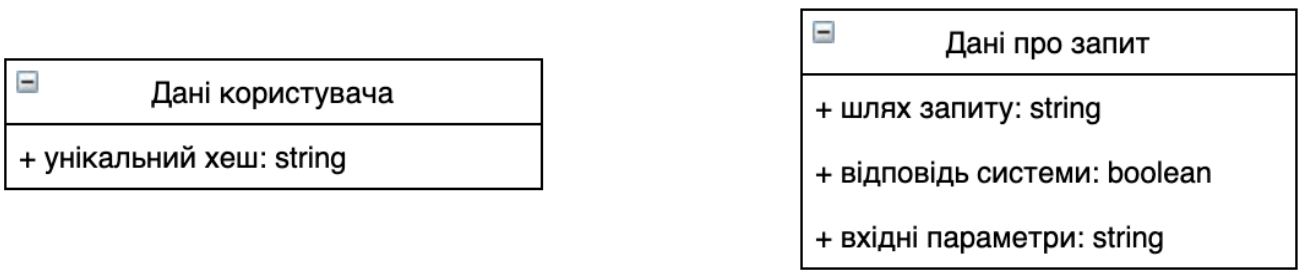


Рисунок 3.3 – Схема даних

3.3 Аналіз та вибір засобів програмної реалізації методу

Під час розробки ПЗ слід врахувати додаткові вимоги:

– засіб має бути гнучким (а саме: легко розширюватись);

– засіб має легко інтегруватись у будь-яке середовище;

– дані користувачів мають бути захищеними, анонімними та відповідати чинному законодавству про обробку та збереження персональних даних;

– засіб має обробляти запити у реальному часі та без затримок.

З огляду на це, було вирішено використати мову програмування JavaScript та середовище Node.js для реалізації алгоритмів (оскільки вказане середовище має низький рівень споживання ресурсів пам’яті, процесора та обчислювальної потужності апаратного забезпечення, що, у свою чергу, задовольняє вимогу про обробку даних у реальному часі та без затримок). Це також забезпечить можливість використовувати програмний засіб у будь-якому додатку.

Також доцільно використати гнучкий фреймворк Express, який дозволяє стандартизувати створення веб-сервера (проксі), легко створювати прикладний програмний інтерфейс, а також пришвидшити створення обробника подій.

Для обробки бібліотек доцільно використати менеджер пакетів NPM, що дасть змогу керувати бібліотеками, які необхідні для функціонування додатку, легко встановлювати додаток у новому середовищі та автоматично оновлювати бібліотеки до останньої версії.

Для зберігання даних оптимальним варіантом є використання In-Memory Cache – це технологія короткотривалого, проте найшвидшого зберігання даних в оперативній пам’яті комп'ютера. Оскільки дані користувачів необхідно обробляти у реальному часі, то, задля уникнення затримок та постійного запису/зчитування у БД, було обрано саме варіант використання кешування.

Для управління контейнерами та ізоляції користувачів був обраний інструментарій Docker, оскільки Docker дає змогу:

– ізолювати ресурси (навантаження пам’яті, процесу), які доступні

користувачу, задля зменшення ризиків відмови для інших користувачів;

– ізолювати файлову систему;

– запускати контейнери незалежно один від одного в будь-який момент часу.

Docker доповнює інструментарій LXC більш високорівневим прикладним програмним інтерфейсом, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє, не переймаючись вмістом контейнера, запускати довільні процеси у режимі ізоляції і потім переносити та клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

3.4 Висновки

У розділі проведено аналіз вимог до програмного засобу та визначено, що програма має реагувати на часткові та повні відмови, автоматично аналізувати і сканувати помилки, а також висувати гіпотези про причини помилок та автоматично виконувати спроби усувати помилки.

Було визначено вимоги до компонентів ПЗ, зокрема, базовим модулем буде Балансувач. Він буде виконувати роль проксі-сервера та відповідати за адаптацію системи до навантаження, використовуючи можливості роботи з контейнерами.

Важливою частиною ПЗ має бути ідентифікація користувачів, що дасть змогу аналізувати активність кожного окремого користувача, а це, у свою чергу, дозволить гнучкіше реагувати на помилки.

Шардінг дасть можливість зменшити вплив користувачів на ПС та на інших користувачів. Також визначено вимоги до створення шардів, зокрема, визначено: яким способом будуть розподілятись запити користувача між контейнерами, що включені у шард даного користувача; за яких умов може відбуватися зміна кількості контейнерів у шарді; за яких умов користувач може бути ізольований.

Були сформовані вимоги до роботи модуля Вимикача, що має на меті тимчасово вимикати певні частини ПС при втраті їх функціональності.

У розділі також обрана архітектура ПЗ, визначено модулі ПЗ та описано призначення кожного модуля, здійснено їх декомпозицію на компоненти, побудовані діаграми UML (зокрема діаграми компонентів та послідовності).

Обрано та обгрунтовано перелік технологій, які є необхідними для вирішення задачі (JavaScipt, Node.JS, Express, NPM та Docker).

Наступним етапом є реалізація спроектованого ПЗ та його тестування, а також доведення ефективності удосконаленого методу відмовостійкості ПС.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

4.1 Програмна реалізація

4.1.1 Структура та призначення модулів програми, їх взаємозв’язок

Програмний засіб складається з наступних модулів.

LoadBalancer (Балансувач) – це модуль управління, який відповідає за створення необхідної кількості контейнерів додатку та маршрутизацію трафіку. Його головна мета – слідкувати за станом контейнерів та за тим, щоб кількість робочих контейнерів відповідала поточному навантаженню.

CircuitBreacker (Вимикач) – це модуль, який відповідає за те, щоб тимчасово відключити частину програмної системи у випадку, якщо частота помилок перевищує задане порогове значення.

Sharding (Шардінг) – це модуль, який відповідає за виділення користувачам декількох контейнерів (шардів), до яких вони мають доступ при роботі з ПС.

xUuid (Ідентифікатор) – модуль, який видає кожному користувачу унікальний хеш-ідентифікатор.

Profiler (Профайлер) – модуль, який логує всі події.

Модулі складаються з компонентів, які взаємодіють між собою.

LoadBalancer (Балансувач) складається з наступних компонентів:

– Proxy (проксі) – модуль перенаправлення запитів;

– Container – модуль роботи з контейнером;

– ContainersStore – модуль роботи з контейнерами.

Модуль Proxy передає вхідний запит на контейнер і повертає відповідь.

Модуль Container створює контейнер, слідкує за його станом та використанням ресурсів у реальному часі. Моніторинг стану та споживання ресурсів контейнера відбувається у строго визначений час з інтервалом в одну секунду, що дає змогу точно оцінювати працездатність контейнера у реальному часі.

Модуль ContainersStore відповідає за автоматичну зміну кількості контейнерів у залежності від навантаження на систему.

Модуль CircuitBreacker складається з наступних компонентів:

- моніторинг роботи додатку;
- робота зі сховищем даних;
- блокування кінцевої точки.

Моніторинг роботи додатку аналізує працездатність системи та виявляє помилки у системі.

Компонент роботи зі сховищем даних відповідає за ведення статистики запитів до кожного окремого контейнера та їх результатів.

Компонент блокування кінцевої точки відповідає за автоматичне відключення частини системи у разі настання критичної маси помилок.

Модуль Sharding складається з наступних компонентів:

- створення шарду для користувача;
- робота зі сховищем даних;
- ізоляція користувача;
- моніторинг активності користувачів.

Створення шарду для користувача відповідає за виділення окремих контейнерів для нього. Також цей компонент забезпечує актуальний стан шарду, тобто оновлює його наповнення у разі відмови одного з контейнерів у його складі та додає або видаляє контейнери при зміні навантаження на систему.

Компонент роботи зі сховищем даних відповідає за ведення статистики запитів до кожного окремого користувача та їх результатів.

Компонент ізоляції користувача відповідає за виділення обмеження шарду користувача до лише одного окремого контейнера.

Компонент моніторингу активності користувача відповідає за аналіз поведінки користувача і його запитів та висуває гіпотези про рівень потенційної небезпеки користувача для системи.

4.1.2 Розробка програмних модулів

Для початку було реалізовано модуль LoadBalancer, який, як було зазначено вище, складається з компонентів Container, ContainersStore та Proxy.

Фрагмент програмного коду, який відповідає за управління контейнером (компонент Container):

```
class Container {
      create = async () => {
         try {
             const instance = await docker.createContainer({
               Image: this.image,
               AttachStdin: false,
               AttachStdout: true,
               AttachStderr: true,
               ExposedPorts: {
                   [`${this.containerPort}/tcp`]: {}
               },
               HostConfig: {
                   CpuShares: 1024,
                   Memory: 128 * 1048576,
                   PortBindings: {[`${this.containerPort}/tcp`]: [{HostPort:
`${this.hostPort}`}]}
               },
               Tty: true,
           });
             this.id = instance.id;
             this.instance = instance
         } catch (e) {
             console.error('Container creation failed:', e)
         }
      }

      start = async () => {
         try {
             await this.instance?.start();
         } catch (e) {
             this.startAttemptsFailed = this.startAttemptsFailed + 1
             console.error('Container start failed:', e)
         }
      }
      stop = async () => {
         /*istanbul ignore else*/
         if (this.timer) {
             clearInterval(this.timer);
         }
         try {
             await this.instance?.remove({force: true})
         } catch (e) {
             console.error('Container stop failed:', e)
         }
      }

      getStatus = async () => {
         try {
             const data = await this.instance?.inspect();
             this.status = data?.State?.Status
             this.health = data?.State?.Health?.Status
         } catch (e) {
             console.error('Container status detection failed:', e)
             this.health = CONTAINER_HEALTH.unhealthy;
         }
      }
}
```

Далі необхідно реалізувати модуль роботи з контейнерами, що дасть змогу динамічно змінювати кількість контейнерів у залежності від навантаження на систему. Цей модуль має забезпечувати безперебійне створення нових та видалення вже запущених контейнерів. Завдяки підходу автоматичної зміни кількості контейнерів система зможе самостійно справлятись з навантаженнями.

Фрагмент коду, що відповідає за управління контейнерами (ContainersStore):

```
class ContainersStore {
      createContainer = async () => {
         const container = new Container(this.image, this.lastUsedPort,
this.containerPort)
         await container.create()
         await container.start()
         container.monitor()
         this.lastUsedPort++;

         return container;
      }

      setContainersToDesired = async () => {
         const containersDiff = this.desiredContainers -
this.getContainersCount();
         if (containersDiff > 0) {
             for (let i = 0; i < containersDiff; i++) {
                 const container = await this.createContainer();
                 this.addContainer(container.id, container);
             }
         } else if (containersDiff < 0) {
             const keys = this.getContainersKeys()
             for (let i = 0; i < containersDiff * -1; i++) {
                 const id = keys[i];
                 const cont = this.getContainer(id)
                 cont.setStatus(CONTAINER_STATUS.removing)
                 cont.stop();
                 this.removeContainer(id)
             }
         }
      }
        isUnhealthy = ({health, status, startAttemptsFailed = 0}) => health ===
CONTAINER_HEALTH.unhealthy
         || status === CONTAINER_STATUS.dead
         || status === CONTAINER_STATUS.removing
         || startAttemptsFailed >= 2

      killUnhealthy = async () => {
         const containerEntries = Object.entries(this.containers);
         await Promise.all(containerEntries.map(async ([id, container]) => {
             if (this.isUnhealthy(container)) {
                 console.log('removing', id, container.status, container.health)
                 await container.stop();
                 this.removeContainer(id)
             }
         }))
      }

      scale = () => {
```

```
        if (this.lastScale + (this.scaleTimeOut * 1000) < Date.now()) {
            const containerValues = Object.values(this.containers)
            const sumMemoryUsage = containerValues.reduce((sum, {memoryUsage}) =>
sum + (memoryUsage ?? 0), 0)
            const sumCPUUsage = containerValues.reduce((sum, {cpuUsage}) => sum +
(cpuUsage ?? 0), 0)
            const averageMemoryUsage = sumMemoryUsage / this.desiredContainers;
            const averageCPUUsage = sumCPUUsage / this.desiredContainers;
            // console.log({averageCPUUsage, averageMemoryUsage},
this.desiredContainers)
            if ((averageMemoryUsage > 30 || averageCPUUsage > 35) &&
this.desiredContainers < this.maxContainers) {
                this.lastScale = Date.now();
                this.desiredContainers =
                    this.desiredContainers + Math.min(this.scaleUpStep,
this.maxContainers - this.desiredContainers)
            } else if ((averageMemoryUsage <= 15 && averageCPUUsage <= 20) &&
this.desiredContainers > this.minContainers) {
                this.lastScale = Date.now();
                this.desiredContainers =
                    this.desiredContainers - Math.min(this.scaleDownStep,
this.desiredContainers - this.minContainers)
            }
        }
    }
}
```

Наступним етапом реалізації є створення модуля Proxy, який передає вхідний запит на контейнер і повертає відповідь.

Фрагмент програмного коду:

```
const request = require('request');
const getNextServer = require("../utils/getNextServer");
const getContainerUrl = require("../utils/getContainerUrl");

const handler = (store, cb) => async (req, res) => {
    const url = req?.locals?.proxyUrl ?? getContainerUrl(getNextServer(store))
    if (!url) {
        console.log('503: Service temporary unavailable')
        return res.status(503).send('Service temporary unavailable');
    }
    const isValid = cb?.validatePath(req.url);
    if (!isValid) {
        console.log('504: Service temporary unavailable')
        return res.status(504).send('Service temporary unavailable');
    }

    const proxy = request({url: url + req.url}).on('error', error => {
        res.status(500).send(error.message);
    });
    req.pipe(proxy).pipe(res);
};
module.exports = handler;
```

Далі реалізуються xUuid (Ідентифікатор) та Profiler (Профайлер).

xUuid відповідає за присвоєння унікального ідентифікатора користувачам, який дає змогу ідентифікувати користувачів, зберігаючи при цьому анонімність. Функція, яка відповідає за це, представлена нижче:

```
const xUuidMiddleware = (req, res, next) => {
   let xUuid = req?.cookies?.xUuid
   if (!xUuid){
       const ip = req.connection.remoteAddress;
       const browser = req.headers["user-agent"];
       const language = req.headers["accept-language"];
       xUuid = createUUID(`${ip}-${browser}-${language}`)
       res.cookie('xUuid', xUuid, {maxAge: 900000})
   }

   if (!req.locals) {
       req.locals = {};
   }
   req.locals.xUuid = xUuid
   next();
};
```

Profiler логує всі події, які відбуваються у системі:

```
const profilerMiddleware = (req, res, next) => {
   const start = Date.now();
   res.on('finish', () => {
       console.log('Completed', req.method, req.url, 'status:', res.statusCode,
'time:', Date.now() - start,);
   });
   next();
};
```

Далі реалізується компонент Sharding, який відповідає за динамічний розподіл користувачів.

Фрагмент програмного коду:

```
class Sharding {
      defineShard = (userUuid, isMalicious) => {
         const keys = this.containersStore.getHealthyContainersKeys()
         const userShards = keys.map((value) => ({value, sort: Math.random()}))
                                 .sort((a, b) => a.sort - b.sort)
                                 .map(({value}) => value).slice(0,
                                                                isMalicious ? 1 :
Math.max(2, Math.ceil(keys.length / 3))
             );
         this.sharding[userUuid] = {
             userShards,
             current: 0,
             containersCount: keys.length,
             isMalicious
```

```
        }
    }

    isUserMalicious = (userUuid) => {
        const userRequests = this.getUserRequests(userUuid);
        if ((userRequests?.length ?? 0) > 25) {
            const errors = userRequests.filter(req => !req.result);
            const serversCalled = errors.map(req => req.serverCalled);
            const serversAffected = [...new Set(serversCalled)].filter(res =>
res);
            const errorsNumber = errors.length;
            const successNumber = userRequests.length - errorsNumber;
            const successRate = successNumber / userRequests.length * 100;
            return successRate < 80 && serversAffected.length > 3;
        }
        return false;
    }
    checkAndReplaceUnhealthyUserShards = (userUuid) => {
        /*istanbul ignore else*/
        if (this.sharding?.[userUuid]?.userShards?.length) {
            const containersKeys =
this.containersStore.getHealthyContainersKeys();
            const allAlive = this.sharding[userUuid].userShards.every(shardId =>
containersKeys.includes(shardId));
            /*istanbul ignore else*/
            if (!allAlive) {
                this.sharding[userUuid].userShards =

this.getNRandContainerKeys(this.sharding[userUuid].userShards.length);
            }
        }
    }
    getUserShards = (userUuid) => {
        const isMalicious = this.isUserMalicious(userUuid);
        if (isMalicious) {
            console.warn('Isolating malicious user', userUuid);
        }
        /*istanbul ignore else*/
        if (!this.sharding?.[userUuid]?.userShards.length || isMalicious !==
this.sharding?.[userUuid]?.isMalicious) {
            this.defineShard(userUuid, isMalicious);
        }

        this.checkAndReplaceUnhealthyUserShards(userUuid);
        this.useNextShard(userUuid)
        this.updateLastAccessedShard()
        return this.sharding[userUuid];
    }
}
```

Далі реалізується компонент CircuitBreacker, який відповідає за відключення непрацюючих частин системи:

```
class CircuitBreacker{
    addURLResult = (path, serverId, result) => {
        const url = this.sanitizeUrl(path);
        if (!this.getURLResults(url)) {
            this.URLs[url] = {calls: []};
        }
```

```
        this.URLs[url].calls.push({result, timeStamp: Date.now(), serverId})

        if (this.URLs[url]?.state === CB_STATE.HALF_OPEN) {
            if (result) {
                console.log('CB is closing', url);
                this.URLs[url].state = CB_STATE.CLOSED;
                this.URLs[url].closedTimeStamp = Date.now();
            } else {
                console.log('CB is re-opening', url);
                this.URLs[url].state = CB_STATE.OPEN;
                this.URLs[url].openTimeStamp = Date.now();
            }
        }
    }
    shouldCBOpen = (path) => {
        const results = this.getURLResults(path);
        if (results?.calls?.length > 50) {
            const successCalls = results?.calls.filter(({result}) => result);
            return (successCalls.length / results.calls.length * 100) < 25
        }
        return false;
    }

    validatePath = (path) => {
        const url = this.sanitizeUrl(path);
        const results = this.getURLResults(url);
        if (results?.state === CB_STATE.CLOSED
            && results?.closedTimeStamp
            && results?.closedTimeStamp + 30 * 1000 > Date.now()) {
            console.log('paused', url)
            return true;
        }

        if (results?.state === CB_STATE.OPEN
            && results?.openTimeStamp
            && results?.openTimeStamp + 30 * 1000 < Date.now()) {
            this.URLs[url].state = CB_STATE.HALF_OPEN;

            console.log('CB is setting half open', url);
            return true;
        }

        if (results?.state !== CB_STATE.OPEN
            && this.shouldCBOpen(url)) {
            this.URLs[url].state = CB_STATE.OPEN;
            this.URLs[url].openTimeStamp = Date.now();
            console.log('CB is opening', url);
        }

        return this.URLs?.[url]?.state !== CB_STATE.OPEN
    }
}
```

Для коректної роботи алгоритму відмовостійкості ПС необхідно визначати першопричину відмови. Для цього слід зберігати дані про запити, які надходять у систему, та про користувачів, щоб зрозуміти, де знаходиться проблема: у користувачах чи у самій системі. Оскільки дані користувачів необхідно обробляти

у реальному часі (задля уникнення затримок та постійного запису/зчитування у БД), було обрано варіант використання кешування як оптимальний.

Згідно із концептом кешування дані, які зберігаються, можуть бути записані у форматі ключ-значення.Формати даних представлено нижче.

Формат даних про запити користувачів:

```
{
      [userUuid]: {
      result:true,
      timeStamp: 000000000,
      serverCalled: “1”
}
```

Формат даних запису шардів користувачів:

```
{
       [userUuid]: {
          userShards:[“1”, “2”, “3”],
          current: 0,
          containersCount: 2,
          isMalicious:false
        }
}
```

Формат даних запису даних вимикача:

```
{
      [url]: {
         calls:[{result:true, timeStamp: 000000000, serverId: “1”}],
         state: 'CLOSED',
         closedTimeStamp: 000000000,
      }
}
```

Програмний код наведено у додатку А.

4.1.3 Реалізація методів поліпшення технічних характеристик системи

Завдяки переосмисленню підходу шардінгу та його перенесенню на розподілені додатки вдалось покращити технічні характеристики методу відмовостійкості ПС.

Новий підхід до шардінгу має на меті створення так званих шардів (певного масиву контейнерів, на які надходять усі запити деякого користувача). Цей підхід дає можливість лімітувати кількість доступних користувачу контейнерів, та, у разі підозрілої діяльності, ізолювати потенційно небезпечних користувачів, виділивши їм лише певні контейнери. Якщо користувач навмисно або ненавмисно продовжить наносити шкоду системі шляхом використання надмірної кількості ресурсів, то, оскільки він знаходитиметься в ізоляції, шкода буде заподіяна лише певним контейнерам, а решта користувачів продовжать використовувати ПС без перешкод.

Даний підхід дає змогу зменшити негативний вплив шкідливих користувачів ПС, оскільки помилки виникатимуть лише в одному з контейнерів.

Цей підхід вдалось реалізувати за допомогою автоматичного створення шардів. Шардінг відбувається для усіх користувачів ПС, незалежно від їх поточного статусу. Спершу усім користувачам системи автоматично видається доступ лише до певних контейнерів. Наприклад, є активних дев’ять контейнерів та 30 користувачів. Першим десятьом користувачам будуть доступні лише перші три контейнери, друга десятка використовуватиме четвертий, п’ятий та шостий контейнери, а решта – три останніх.

Однак самих лише шардів недостатньо для запобігання помилок у системі, тому необхідно реалізувати алгоритм, який дасть змогу рівномірно розподіляти навантаження між контейнерами у шарді. Для цього реалізовано розподіл запитів між контейнерами шляхом циклічного планування. Розподіл запитів між контейнерами дає змогу запобігти перевантаженню та виходу з ладу контейнерів. Якщо контейнер перестав відповідати на запити або не функціонує як слід, тоді користувачу виділяється або активний контейнер, який уже запущений, або створюється новий. Це демонструє наступний фрагмент програмного коду:

```
checkAndReplaceUnhealthyUserShards = (userUuid) => {
   /*istanbul ignore else*/
   if (this.sharding?.[userUuid]?.userShards?.length) {
       const containersKeys = this.containersStore.getHealthyContainersKeys();
       const allAlive = this.sharding[userUuid].userShards.every(shardId =>
containersKeys.includes(shardId));
       /*istanbul ignore else*/
       if (!allAlive) {
```

```
            this.sharding[userUuid].userShards =
this.getNRandContainerKeys(this.sharding[userUuid].userShards.length);
        }
    }
}
```

Окрім цього, розмір шарду має бути динамічним та залежати від кількості активних контейнерів. Наприклад, якщо на деякий момент часу є дев'ять активних контейнерів, то шард не повинен складатися більше ніж з трьох контейнерів. Якщо кількість контейнерів змінилась через зміну навантаження, тоді і кількість контейнерів у шарді має змінитись автоматично. Це дасть змогу ефективно забезпечувати доступність додатку для користувачів.

Фрагмент програмного коду:

```
defineShard = (userUuid, isMalicious) => {
    const keys = this.containersStore.getHealthyContainersKeys()
    const userShards = keys.map((value) => ({value, sort: Math.random()}))
                           .sort((a, b) => a.sort - b.sort)
                           .map(({value}) => value).slice(0, isMalicious ? 1 :
Math.max(2, Math.ceil(keys.length / 3))
        );
    this.sharding[userUuid] = {
        userShards,
        current: 0,
        containersCount: keys.length,
        isMalicious
    }
}
```

Незважаючи на те, що шардінг дає змогу надати користувачам доступ лише до певних контейнерів, все ще є необхідність ізолювати підозрілих користувачів серед інших (оскільки може виникнути ситуація, що один з користувачів вичерпає усі системні ресурси контейнерів у його шарді, що, у свою чергу, викличе обмеження у використанні ПС для інших користувачів).

Ізоляція підозрілого користувача відбуваться за наступною логікою: якщо протягом деякого періоду часу запити підозрілого користувача не викликають помилок, то тоді цей користувач знову вважається звичайним.

Фрагмент програмного коду:

```
isUserMalicious = (userUuid) => {
    const userRequests = this.getUserRequests(userUuid);
```

```
    if ((userRequests?.length ?? 0) > 25) {
        const errors = userRequests.filter(req => !req.result);
        const serversCalled = errors.map(req => req.serverCalled);
        const serversAffected = [...new Set(serversCalled)].filter(res => res);
        const errorsNumber = errors.length;
        const successNumber = userRequests.length - errorsNumber;
        const successRate = successNumber / userRequests.length * 100;
        return successRate < 80 && serversAffected.length > 3;
    }
    return false;
}
```

Наступним етапом є тестування покращеного методу відмовостійкості ПС та аналіз його ефективності.

4.2 Результати тестування програмного засобу та їх аналіз

4.2.1 Вибір методів тестування

Після проектування та програмної реалізації методу відмовостійкості ПС необхідно провести тестування і переконатись, що система успішно реагує на відмови та може автоматично відновлюватись після настання відмови.

Одним із методів тестування відмовостійкості є так зване «мавпяче тестування», яке було запропоновано компанією Netflix [20]. Цей метод має на меті хаотичне виключення будь-якого із сервісів (БД, сервер, відключення інтеграцій тощо) та запуск тестів усієї системи з подальшим генеруванням детального звіту. Наприклад, якщо досліджувана ПС є Інтернет-магазином, то «мавпяче тестування» виглядатиме наступним чином чином.

1) Випадковим чином вибираємо сервіс для відключення (наприклад, БД товарів).

2) Запуск автоматичних тестів. Наприклад, перевіряємо, що користувачі можуть авторизуватись на сайті, відкрити головну сторінку та, у залежності від реалізації сховищ даних (тобто замовлені товари беруться безпосередньо з непрацюючої БД чи містяться в іншій функціонуючій базі), будуть або не будуть бачити свої замовлення.

3) Перевіряємо реакцію сервісів, які безпосередньо взаємодіють з непрацюючим сервісом. Наприклад, запускаються тести на пошук товарів. Звісно, є

очікуваним, що певні функції системи не будуть працювати зовсім або працюватимуть в обмеженому режимі, проте цей етап потрібен не лише для визначення, ЩО саме перестане працювати, а й для з'ясування, ЯК система буде себе поводити у режимі часткової працездатності. Як реагує вона на те, що певна БД не відповідає? Чи пробує ПС в автоматичному режимі зчитати дані з резервної БД? Чи надсилає вона надлишкові запити до непрацюючого сервісу? Чи здійснює система повторні запити до непрацюючого сервісу? Чи з'являються поломки, затримки у інших сервісів через непрацюючий сервіс?

Таким чином, цей етап дає можливість не лише визначити часткову або повну непрацездатність ПС, а й її поведінку, та визначити можливі області для покращень.

4) Формування та аналіз звіту. На цьому етапі формується звіт, де зазначається, які сервіси втратили працездатність та як повела себе ПС через це. Під час аналізу звіту перевіряється, чи це є очікуваною поведінкою системи, чи є якісь місця для покращень. Наприклад, якщо у випадку із вимкненням БД товарів користувачам відображається «білий екран» замість головної сторінки через те, що компонент «Останні придбані товари» почав здійснювати нескінченну кількість запитів (що призвело до «поломки» всієї сторінки), то це означає, що система невірно відреагувала на «поломку». У даному випадку система мала б за допомогою паттерну Circuit breaker визначити, що сервіс є недоступним та вивести користувачу повідомлення про помилку.

Для емпіричного дослідження розробленого ПЗ використовуються два види тестування: модульне та навантажувальне.

Модульне тестування має на меті окреме тестування кожної функції та модуля програмного засобу.

Навантажувальне тестування має на меті довести, що система залишається стійкою, незважаючи на різку зміну навантаження.

Оскільки алгоритм працюватиме вірно лише, якщо всі функції системи працюють правильно, то кожна функція системи протестована за допомогою модульного тестування.

Для проведення тестування необхідно використати допоміжні бібліотеки, які дають змогу працювати з модульними тестами.

Jest дає змогу писати модульні тести для додатків, які розроблені мовою JavaScript. Його перевагами є:

– простота;

– велика кількість документації;

– ізольований запуск тестів.

Завдяки перевагам цієї бібліотеки тести можна розробити легко і швидко.

Також для модульного тестування використано бібліотеку Sinon, яка дає змогу імітувати відповідь системи.

Окрім модульного тестування, необхідно також провести навантажувальне тестування, щоб переконатись у тому, що система може:

– динамічно управляти контейнерами;

– змінювати розмір шарду;

– ізолювати користувачів при виявленні підозрілої активності;

– блокувати запити до певного функціоналу ПС при появі помилок.

Для проведення навантажувального тестування було обрано інструментарій JMeter. Його превагами є:

– простота використання;

– багатопоточність;

– емуляція значної кількості користувачів

– презентація результатів тестування різними способами.

Також під час навантажувального тестування для перевірки усіх можливих сценаріїв було створено простий додаток, який має декілька різних кінцевих точок, які емулюють навантаження на центральний процесор та оперативну пам’ять. Так, було створено кінцеві точки:

– / (home);

– /posts;

– /login;

– /comments;

– /health.

Варто відзначити, що кінцева точка /comments була створена як емуляція складної операції (яка потребує значної кількості обчислювальних ресурсів та часу для виконання), що не функціонує правильно, та повертає помилку. Кінцева точка /health повертала помилку відносно швидко, але призводила до повної втрати працездатності додатку.

4.2.2 Розробка тестових сценаріїв

Метою модульного тестування є емпіричне дослідження ПЗ на його відповідність вимогам, наведеним у специфікації вимог. У таблиці 4.1 наведено перелік тестових випадків та описано очікуваний результат роботи кожної функції програмного засобу.

Таблиця 4.1 – Тестові випадки модульного тестування

Ч. ч.	Модуль/Функція	Вихідні дані	Очікуваний результат
1	2	3	4
1	Службові програми/ Створити UUID	Унікальний ідентифікатор за стандартом UUID версії 5.	Має повертати правильне значення
2	Профайлер	Створення запису логу	Має створити запис логу
3	Шардінг/Обрати контейнер, який буде виконувати запит	Ідентифікатор контейнера та шлях до нього	Має повертати ідентифікатор контейнера та шлях за яким буде здійснено запит
4	Балансувач/Циклічне планування	Ідентифікатор контейнера та шлях до нього	Має циклічно повертати ідентифікатор контейнера та шлях, за яким буде здійснено запит
5	Статистика про користувачів (stats)	Статистика запитів користувачів та відповіді системи на запити користувачів	Збереження та обробка даних про користувачів та відповіді системи на запити користувачів

Продовження таблиці 4.1

1	2	3	4
6	Службові програми/ Побудувати шлях до контейнеру	Шлях до контейнеру	Має повернути повний шлях до контейнеру
7	Ідентифікатор (створення)	Унікальний ідентифікатор користувача	Створення унікального ідентифікатора користувача
8	Ідентифікатор (запис)	Запис унікального ідентифікатора користувача в кеш-пам’яті	Записує унікальний ідентифікатор користувача в кеш-пам’ять
9	Робота із контейнером/ Створення контейнера	Результат виконання операції	Створення нового контейнера або повернення помилки, якщо щось пішло не так
10	Робота із контейнером/ Старт контейнера	Результат виконання операції	Запуск нового контейнера або повернення помилки, якщо щось пішло не так
11	Робота із контейнером/ Вимкнення контейнера	Результат виконання операції	Вимкнення зайвого контейнера
12	Робота із контейнером/ Отримання статусу контейнера	Результат виконання операції	Отримує дані, чи контейнер є працездатним
13	Робота із контейнером/ Отримання кількості використаних ресурсів контейнера	Результат виконання операції	Отримує дані про кількість використаної оперативної пам’яті та завантаження центрального процесора
14	Робота із контейнером/ Запуск моніторингу контейнера	Ідентифікатор таймеру	Створюється обробник, який виконується щосекунди
15	Робота із контейнерами/ Створення екземпляра контейнера	Контейнер	Створюється екземпляр контейнера
16	Робота із контейнерами/ Додавання контейнера до пулу	Оновлений список контейнерів	Додає контейнера до списку
17	Робота із контейнерами/ Отримання пулу контейнерів	Список контейнерів	Отримує дані усіх контейнерів у списку
18	Робота із контейнерами/ Перевірка контейнера на працездатність	Результат виконання операції	Повертає інформацію про працездатність контейнера
19	Робота із контейнерами/ Отримання списку працездатних контейнерів	Список контейнерів	Отримує список працездатних контейнерів

Кінець таблиці 4.1

1	2	3	4
20	Робота із контейнерами/ Видалення контейнера	Результат виконання операції	Видаляє вказаний контейнер
21	Балансувач/Встановити бажану кількість контейнерів	Оновлений список контейнерів	Повертає оновлений список, що відповідає бажаній кількості контейнерів (тобто створює нові контейнери або видаляє старі)
22	Балансувач/Масштабування	Бажана кількість контейнерів	Повертає кількість контейнерів, яка відповідає поточному навантаженню
23	Шардінг/Запис даних про запит користувача	Дані про запити користувача	Записує до кешу дані про останній запит користувача та повертає його результат
24	Шардінг/Отримання даних про запити користувача	Дані про запити користувача	Отримує дані про останні запити користувача
25	Шардінг/Створення шарду користувача	Інформація про шард користувача	Створює шард для даного користувача та повертає його; у разі, якщо активність користувача є підозрілою, – ізолює користувача у шарді з одного контейнера
26	Шардінг/Отримання шарду користувача	Інформація про шард користувача	Повертає шард для даного користувача
27	Шардінг/Перевірка підозрілості користувача	Результат виконання операції	Повертає інформацію про підозрілість активності користувача
28	Шардінг/Перевірка шарду на працездатність та оновлення шарду	Інформація про шард користувача	Видаляє непрацездатні контейнери з шарду та додає нові контейнери
29	Вимикач/Збереги результати запиту до кінцевої точки	Результат виконання операції	Записує в кеш дані про останній запит до кінцевої точки
30	Вимикач/Перевірити, чи можливий доступ до кінцевої точки	Результат виконання операції	Перевіряє, чи є безпечним доступ до даної кінцевої точки
31	Вимикач/Перевірити, чи потрібно відкривати Вимикач	Результат виконання операції	Перевіряє, чи є потрібним відключити дану кінцеву точку
32	Вимикач/Перевірити, чи потрібно закривати Вимикач	Результат виконання операції	Перевіряє, чи є потрібним відновити доступ до даної кінцевої точки

Також було проведене навантажувальне тестування, що дало змогу переконатись у працездатності удосконаленого методу відмовостійкості ПС.

У процесі навантажувального тестування було емульовано навантаження у 500 користувачів, які циклічно виконують декілька послідовних запитів до додатку (ці дії повторювались сім разів).

Одна з кінцевих точок додатку працювала повільно та повертала помилку. Також було додано три шкідливих користувачі, які виконували запити до неробочої кінцевої точки, запит до якої призводив до відмови роботи контейнера додатку.

4.2.3 Аналіз результатів тестування

Результати тестування службових модулів та модуля ідентифікатора наведено на рисунку 4.1. На цьому рисунку показано, що тестові випадки, наведені у таблиці 4.1, пройдені успішно.

```
PASS __tests__/middlewares/stats.test.js
stats
  ✓ should add data (5 ms)

PASS __tests__/utils/getNextServer.test.js
getNextServer
  ✓ should return proper server (4 ms)
  ✓ should return proper value if some servers are invalid  (2 ms)

PASS __tests__/utils/getContainerUrl.test.js
getContainerUrlTest
  ✓ should return proper value (3 ms)
  ✓ should return null on empty server

PASS __tests__/middlewares/xUuid.test.js
xUuidMiddleware
  ✓ should store uuid to locals (4 ms)
  ✓ should create new uuid and set it to locals (1 ms)
```

Рисунок 4.1 – Результати тестування службових модулів та модуля ідентифікатора

Результати тестування модуля, який відповідає за роботу з контейнером, представлені на рисунку 4.2. Як бачимо, тестові випадки, описані у таблиці 4.1, виконались успішно. Середній час тестування склав близько однієї мілісекунди.

```
PASS __tests__/utils/Container.test.js
  Container
    ✓ should create instance with values set (4 ms)
    ✓ should set status
    ✓ should create docker container (2 ms)
    ✓ should return error on docker container create (1 ms)
    ✓ should start docker container
    ✓ should return error on docker container start (1 ms)
    ✓ should get docker container status (4 ms)
    ✓ should return error on docker container status
    ✓ should get docker container consumption (1 ms)
    ✓ should set consumption to previous value if no data available (1 ms)
    ✓ should return error on docker container status (1 ms)
    ✓ should start timer on monitoring (1 ms)
    ✓ should start timer on monitoring (1 ms)
    ✓ should stop docker container an clear timer
    ✓ should return error on docker container stop (1 ms)
```

Рисунок 4.2 – Результати тестування модуля роботи з контейнером

Результати тестування профайлера та проксі представлений на рисунку 4.3. На цьому рисунку продемонстровано, що всі функції модулів проходять тести успішно, і немає розбіжності між тестовим випадком та результатом.

```
PASS __tests__/middlewares/profiler.test.js
  profilerMiddleware
    ✓ should log data (5 ms)

PASS __tests__/handlers/proxy.test.js
  proxy
    ✓ should return error if url is not available (5 ms)
    ✓ should return error if url is not valid (1 ms)
    ✓ should call proxy (5 ms)
```

Рисунок 4.3 – Результат тестування модулів профайлера та проксі

Результат тестування шардінгу представлений на рисунку 4.4. На цьому рисунку також представлені усі функції, які були протестовані у модулі шардінгу.

```
PASS __tests__/utils/Sharding.test.js
  Sharding
    ✓ should start polling on creation (4 ms)
    ✓ should add and get user requests (2 ms)
    ✓ should not add request if user id is not set (1 ms)
    ✓ should not add request if user id is not set
    ✓ should return false on malicious user identification when has no requests logged (1 ms)
    ✓ should return false on malicious user identification when has <25 requests logged
    ✓ should return false on malicious user identification when has >25 requests logged (1 ms)
    ✓ should return false on malicious user identification when has >25 failed requests logged (1 ms)
    ✓ should identify malicious user
    ✓ should define shard (1 ms)
    ✓ should define shard
    ✓ should define malicious user shard (1 ms)
    ✓ should chose next shard (1 ms)
    ✓ should chose first shard
    ✓ should chose not change current shard (1 ms)
    ✓ should set last accessed date
    ✓ should change list of hosts (1 ms)
    ✓ should change list of hosts (1 ms)
    ✓ should get user shards (1 ms)
    ✓ should get malicious user shards (1 ms)
    ✓ should get malicious user shards (1 ms)
    ✓ should get clear user shards
```

Рисунок 4.4 – Результати тестування модуля шардінгу

Результати тестування вимикача представлено на рисунку 4.5. Розбіжностей між очікуваною та реальною поведінкою системи не виявлено; усі функції у модулі працюють так, як і очікується.

Результати тестування модуля роботи з контейнерами представлено на рисунку 4.6. На цьому рисунку видно, що тестові випадки, описані у таблиці 4.1, виконались успішно. Середній час тестування склав близько однієї мілісекунди.

Звіт про покриття тестами усього додатку представлено на рисунку 4.7. На рисунку представлено відсоткове співвідношення покритих функцій та рядків до непокритих. Як бачимо, 100% функцій та рядків протестовано та нема жодного рядка, не покритого тестами.

```
PASS __tests__/utils/createUUID.test.js
  createUUID
    ✓ should return proper value (2 ms)

PASS __tests__/utils/CB.test.js
  CB
    ✓ should start polling on creation (4 ms)
    ✓ should sanitize url (1 ms)
    ✓ should return added logs (1 ms)
    ✓ should clear url calls (1 ms)
    ✓ should open CB (1 ms)
    ✓ should not open CB (1 ms)
    ✓ should not open CB when too few requests logged
    ✓ should return false on validatePath (38 ms)
    ✓ should return true on validatePath (1 ms)
    ✓ should set state on half open (5 ms)
    ✓ should return true if cb was just closed (2 ms)
    ✓ should set closed on adding new result (1 ms)
    ✓ should set closed on adding new result (1 ms)
```

Рисунок 4.5. Результат тестування модуля вимикача

```
PASS __tests__/utils/ContainersStore.test.js
  ContainersStore
    ✓ should create instance with values set (4 ms)
    ✓ should set and get container by id (1 ms)
    ✓ should return null if container is not available
    ✓ should return all containers (1 ms)
    ✓ should return healthy containers
    ✓ should return containers keys (1 ms)
    ✓ should return healthy containers keys (1 ms)
    ✓ should return containers count
    ✓ should remove container (1 ms)
    ✓ should create container (1 ms)
    ✓ should increment port on create container (1 ms)
    ✓ should create containers to desired state (1 ms)
    ✓ should not do anything if containers are at desired state (1 ms)
    ✓ should remove containers to desired state (1 ms)
    ✓ should identify unhealthy container
    ✓ should identify dead container (4 ms)
    ✓ should identify removing container
    ✓ should identify container that not able to start (1 ms)
    ✓ should kill unhealthy container (47 ms)
    ✓ should scale if memory consumption is too high
    ✓ should scale if cpu usage is too high  (1 ms)
    ✓ should scale down if memory and cpu usage is low
    ✓ should not scale down if memory and cpu usage is not low
    ✓ should not scale if scaling happened recently  (1 ms)
    ✓ should call all needed functions on monitor
    ✓ should call all needed functions on monitor  (1 ms)

PASS __tests__/middlewares/urlChooser.test.js
  urlChooser
    ✓ should return container (3 ms)
```

Рисунок 4.6 – Результати тестування модуля роботи з контейнерами

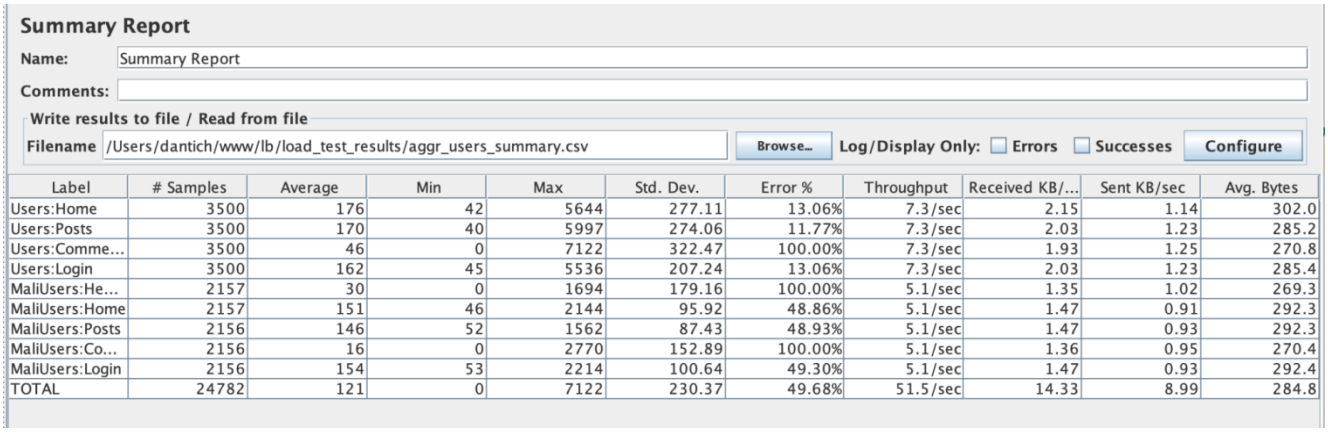
```
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |     100 |      100 |     100 |     100 |
 handlers            |     100 |      100 |     100 |     100 |
  proxy.js           |     100 |      100 |     100 |     100 |
 middlewares         |     100 |      100 |     100 |     100 |
  profiler.js        |     100 |      100 |     100 |     100 |
  stats.js           |     100 |      100 |     100 |     100 |
  urlChooser.js      |     100 |      100 |     100 |     100 |
  xUuid.js           |     100 |      100 |     100 |     100 |
 utils               |     100 |      100 |     100 |     100 |
  CB.js              |     100 |      100 |     100 |     100 |
  Container.js       |     100 |      100 |     100 |     100 |
  ContainersStore.js |     100 |      100 |     100 |     100 |
  Sharding.js        |     100 |      100 |     100 |     100 |
  configs.js         |     100 |      100 |     100 |     100 |
  createUUID.js      |     100 |      100 |     100 |     100 |
  getContainerUrl.js |     100 |      100 |     100 |     100 |
  getNextServer.js   |     100 |      100 |     100 |     100 |
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 12 passed, 12 total
Tests:       89 passed, 89 total
Snapshots:   0 total
Time:        1.19 s, estimated 2 s
```

Рисунок 4.7 – Звіт про повне покриття тестами удосконаленого методу

Результати навантажувального тестування представлені на рисунку 4.8. Система успішно впоралася з навантаженням у 24782 запитів та показала зменшення кількості помилок у порівнянні з таким же тестуванням, проведеним на додатку без удосконаленого методу (що представлено на рисунку 4.9).

4.3 Оцінка ефективності удосконаленого методу вирішення задачі

У дипломній роботі було розроблено удосконалений метод забезпечення відмовостійкості ПС. Було імплементовано наступні зміни.



Summary Report

Name: Summary Report

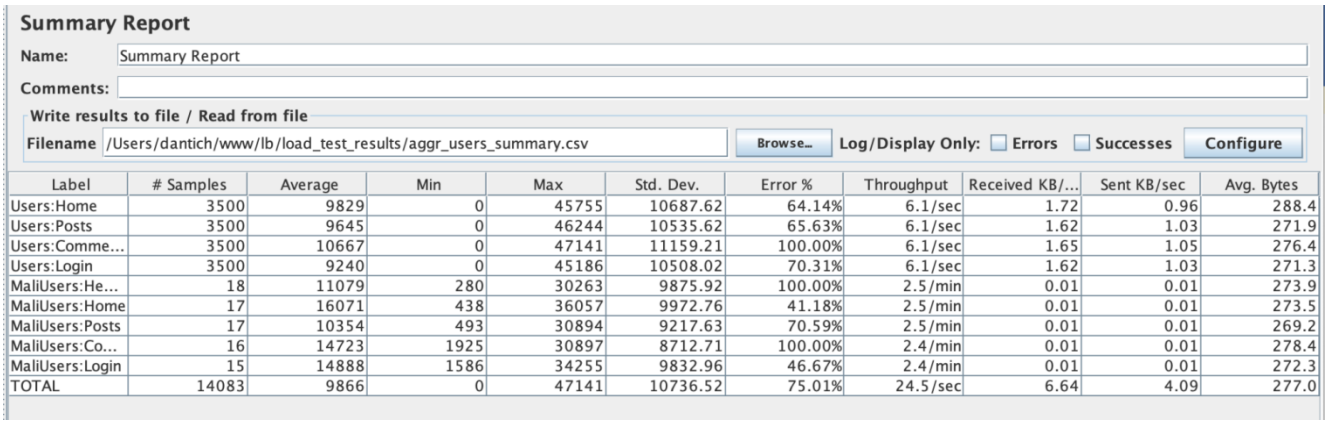
Comments:

Write results to file / Read from file

Filename /Users/dantich/www/lb/load_test_results/aggr_users_summary.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Users:Home	3500	176	42	5644	277.11	13.06%	7.3/sec	2.15	1.14	302.0
Users:Posts	3500	170	40	5997	274.06	11.77%	7.3/sec	2.03	1.23	285.2
Users:Comme...	3500	46	0	7122	322.47	100.00%	7.3/sec	1.93	1.25	270.8
Users:Login	3500	162	45	5536	207.24	13.06%	7.3/sec	2.03	1.23	285.4
MaliUsers:He...	2157	30	0	1694	179.16	100.00%	5.1/sec	1.35	1.02	269.3
MaliUsers:Home	2157	151	46	2144	95.92	48.86%	5.1/sec	1.47	0.91	292.3
MaliUsers:Posts	2156	146	52	1562	87.43	48.93%	5.1/sec	1.47	0.93	292.3
MaliUsers:Co...	2156	16	0	2770	152.89	100.00%	5.1/sec	1.36	0.95	270.4
MaliUsers:Login	2156	154	53	2214	100.64	49.30%	5.1/sec	1.47	0.93	292.4
TOTAL	24782	121	0	7122	230.37	49.68%	51.5/sec	14.33	8.99	284.8

Рисунок 4.8 – Загальні результати навантажувального тестування з удосконаленим методом



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename /Users/dantich/www/lb/load_test_results/aggr_users_summary.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Users:Home	3500	9829	0	45755	10687.62	64.14%	6.1/sec	1.72	0.96	288.4
Users:Posts	3500	9645	0	46244	10535.62	65.63%	6.1/sec	1.62	1.03	271.9
Users:Comme...	3500	10667	0	47141	11159.21	100.00%	6.1/sec	1.65	1.05	276.4
Users:Login	3500	9240	0	45186	10508.02	70.31%	6.1/sec	1.62	1.03	271.3
MaliUsers:He...	18	11079	280	30263	9875.92	100.00%	2.5/min	0.01	0.01	273.9
MaliUsers:Home	17	16071	438	36057	9972.76	41.18%	2.5/min	0.01	0.01	273.5
MaliUsers:Posts	17	10354	493	30894	9217.63	70.59%	2.5/min	0.01	0.01	269.2
MaliUsers:Co...	16	14723	1925	30897	8712.71	100.00%	2.4/min	0.01	0.01	278.4
MaliUsers:Login	15	14888	1586	34255	9832.96	46.67%	2.4/min	0.01	0.01	272.3
TOTAL	14083	9866	0	47141	10736.52	75.01%	24.5/sec	6.64	4.09	277.0

Рисунок 4.9 – Загальні результати навантажувального тестування без удосконаленого методу

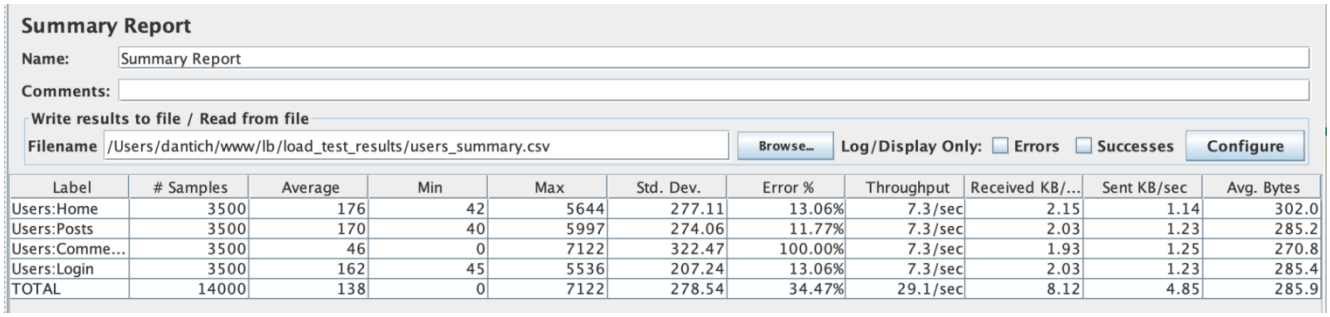
1 Перенесення підходу шардінгу на розподілені мережні застосунки. Ця зміна дала змогу обмежувати доступ користувачів до різних контейнерів.

2 Визначення підозрілих користувачів та їх ізоляція від решти користувачів. Ця зміна дала змогу зменшити кількість помилок у системі та зменшити негативний вплив підозрілих користувачів на ПС, оскільки помилки виникають лише в окремих контейнерах та не заважають іншим користувачам використовувати ПС.

3 Поєднання алгоритмів навантаження, вимикача та шардінгу. Дана зміна дала змогу отримати максимум переваг кожного з методів та забезпечити максимальне покриття всіх типів відмов.

Практична апробація отриманих результатів показала, що удосконалений алгоритм відмовостійкості ПС дає змогу системі обробляти більшу кількість запитів та зменшує кількість помилок, які виникають у системі.

Завдяки реалізації нового підходу до шардінгу система може самостійно визначити потенційно небезпечного користувача та ізолювати його; таким чином на 50%-60% зменшується кількість помилок для звичайних користувачів (що можна побачити на рисунку 4.10).


Summary Report

Name: Summary Report

Comments:

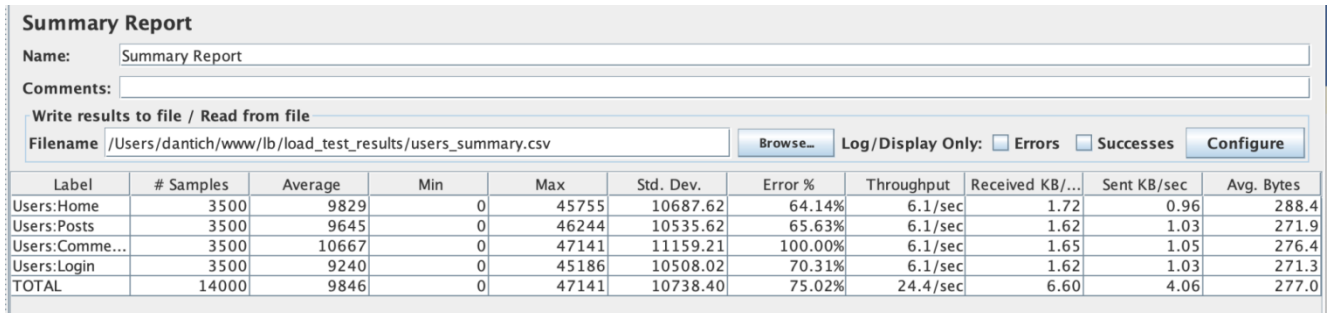
Write results to file / Read from file

Filename /Users/dantich/www/lb/load_test_results/users_summary.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Users:Home	3500	176	42	5644	277.11	13.06%	7.3/sec	2.15	1.14	302.0
Users:Posts	3500	170	40	5997	274.06	11.77%	7.3/sec	2.03	1.23	285.2
Users:Comme...	3500	46	0	7122	322.47	100.00%	7.3/sec	1.93	1.25	270.8
Users:Login	3500	162	45	5536	207.24	13.06%	7.3/sec	2.03	1.23	285.4
TOTAL	14000	138	0	7122	278.54	34.47%	29.1/sec	8.12	4.85	285.9

Рисунок 4.10 – Результати запитів користувачів під час навантажувального тестування з використанням удосконаленого методу

У той же час та сама система без використання удосконаленого методу (з тими ж самими налаштуваннями тестування) має значно вищий відсоток помилок, що представлено на рисунку 4.11.


Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename /Users/dantich/www/lb/load_test_results/users_summary.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Users:Home	3500	9829	0	45755	10687.62	64.14%	6.1/sec	1.72	0.96	288.4
Users:Posts	3500	9645	0	46244	10535.62	65.63%	6.1/sec	1.62	1.03	271.9
Users:Comme...	3500	10667	0	47141	11159.21	100.00%	6.1/sec	1.65	1.05	276.4
Users:Login	3500	9240	0	45186	10508.02	70.31%	6.1/sec	1.62	1.03	271.3
TOTAL	14000	9846	0	47141	10738.40	75.02%	24.4/sec	6.60	4.06	277.0

Рисунок 4.11 – Результати запитів користувачів під час навантажувального тестування без використання удосконаленого методу

Дуже значною є різниця у графіках повернення кодів результату запиту. Так, на рисунку 4.12 представлено графік роботи системи з увімкненим

алгоритмом шардінгу та вимикача. Код «200» означає, що додаток повернув результат без помилок, «500» – що відбулася помилка при виконання запиту, «504» – спрацював алгоритм вимикача та заблокував кінцеву точку, яка працювала з помилками. Тоді як на рисунку 4.13 представлено графік роботи без використання удосконаленого методу, на якому кількість помилок значно переважає кількість успіхів.

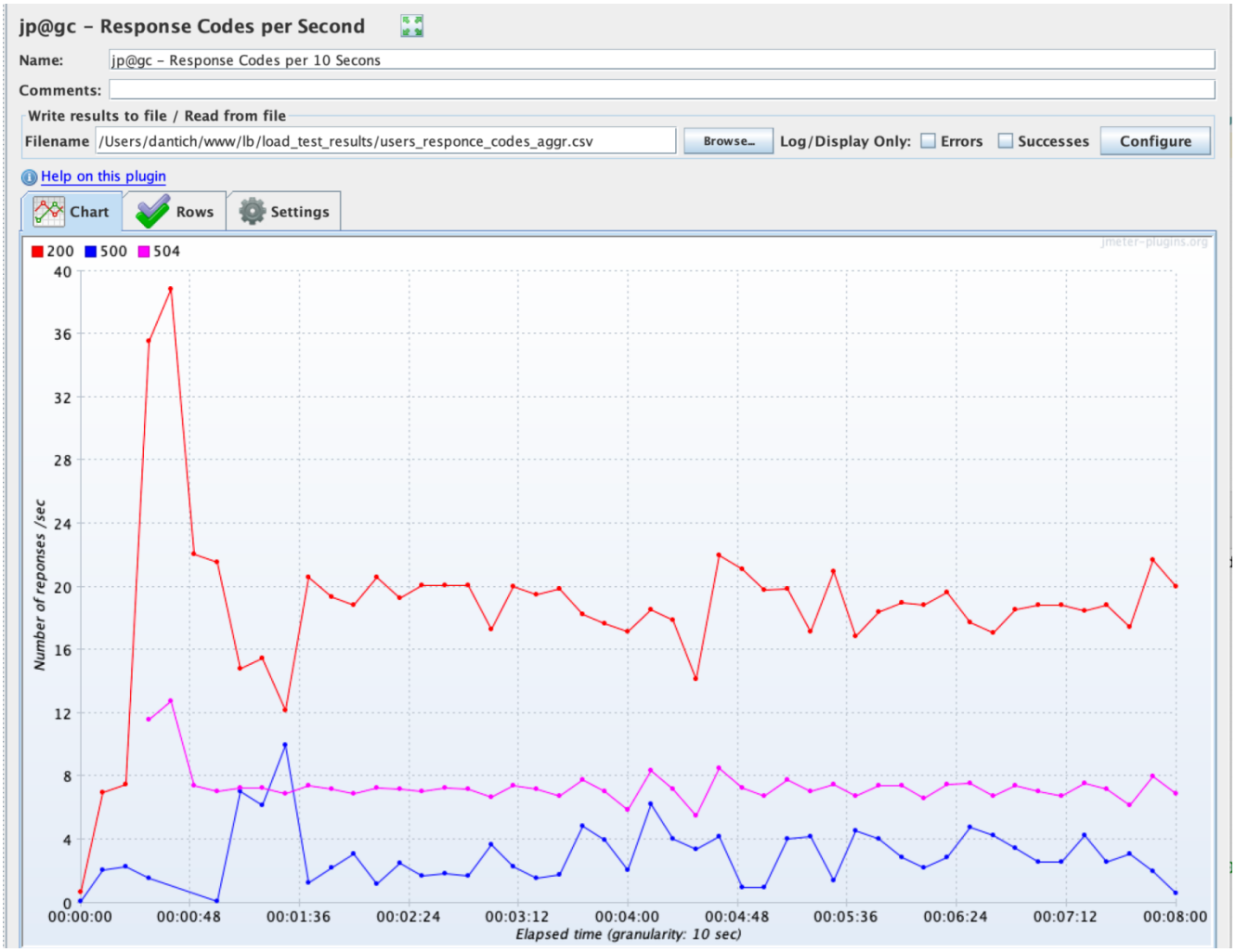


Рисунок 4.12 – Графік результатів запитів користувачів під час навантажувального тестування з використанням удосконаленого методу

Окрім цього, не можна не відмітити, що використання удосконаленого методу зробило систему більш стійкою до відмов. Наприклад, результат тестування показав, що система без використання нового методу здатна виконати 14083 запитів за 9.5 хвилин (представлено на рисунку 4.14), в той час як

удосконалений метод дав змогу системі виконати 24782 запитів за 8 хвилин (представлено на рисунку 4.15), що на 56% більше.

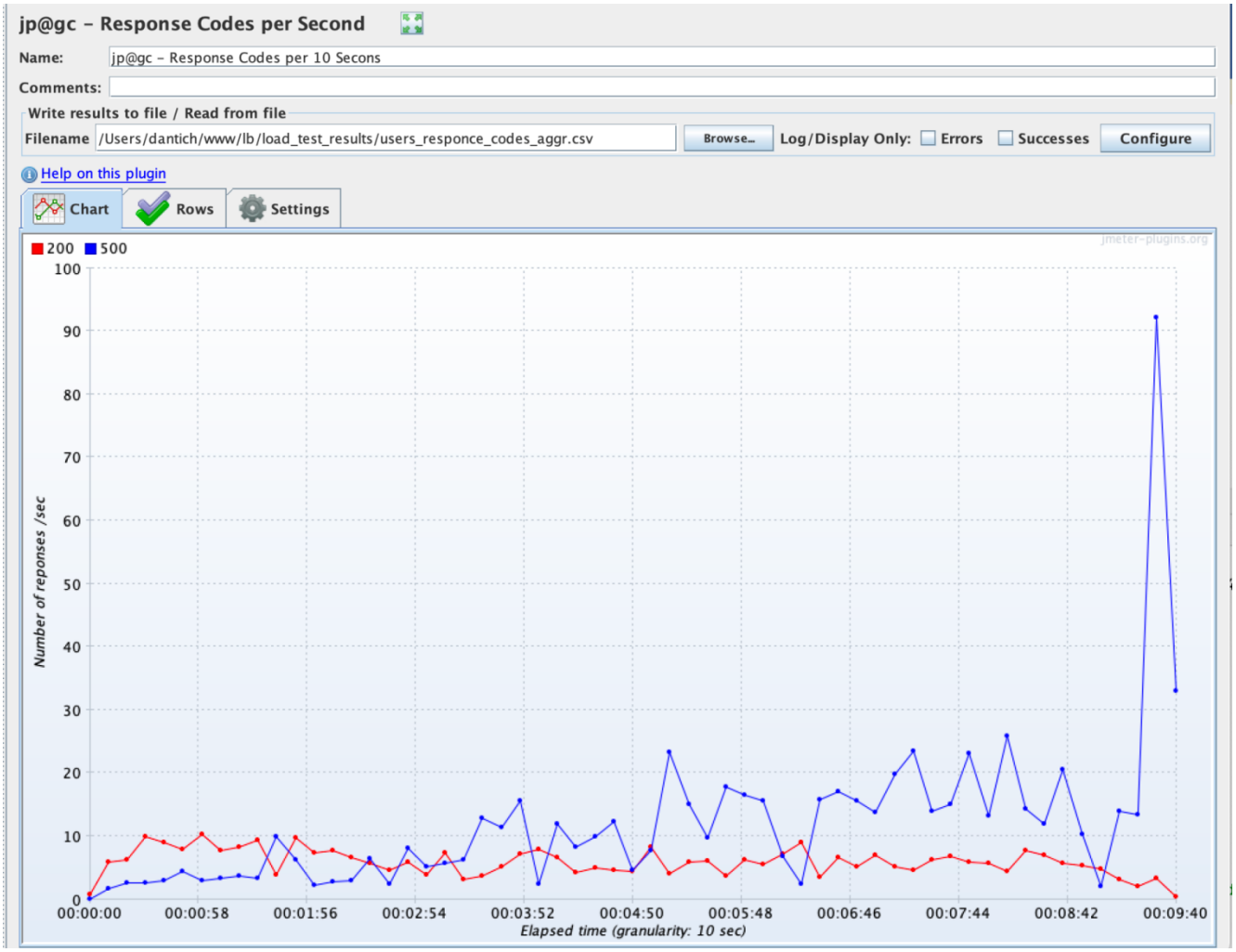
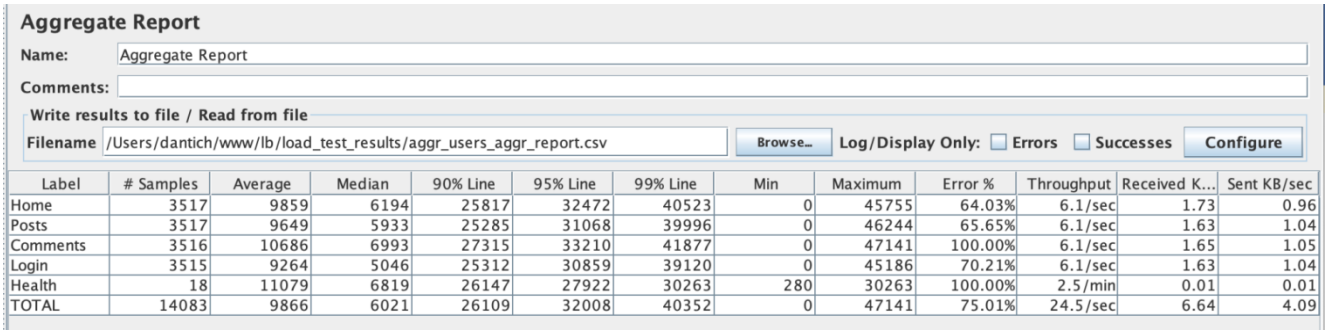


Рисунок 4.13. Графік результатів запитів користувачів під час навантажувального тестування без використання удосконаленого методу



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
Home	3517	9859	6194	25817	32472	40523	0	45755	64.03%	6.1/sec	1.73	0.96
Posts	3517	9649	5933	25285	31068	39996	0	46244	65.65%	6.1/sec	1.63	1.04
Comments	3516	10686	6993	27315	33210	41877	0	47141	100.00%	6.1/sec	1.65	1.05
Login	3515	9264	5046	25312	30859	39120	0	45186	70.21%	6.1/sec	1.63	1.04
Health	18	11079	6819	26147	27922	30263	280	30263	100.00%	2.5/min	0.01	0.01
TOTAL	14083	9866	6021	26109	32008	40352	0	47141	75.01%	24.5/sec	6.64	4.09

Рисунок 4.14 – Агрегований результат навантажувального тестування без використання удосконаленого методу

Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename /Users/dantich/www/lb/load_test_results/aggr_users_aggr_report.csv Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
Home	5657	166	121	256	316	1137	42	5644	26.71%	11.8/sec	3.43	1.93
Posts	5656	161	120	246	315	887	40	5997	25.94%	11.8/sec	3.31	2.04
Comments	5656	35	0	1	1	1278	0	7122	100.00%	11.8/sec	3.11	2.08
Login	5656	159	121	251	317	874	45	5536	26.87%	11.8/sec	3.31	2.05
Health	2157	30	1	1	1	1141	0	1694	100.00%	5.1/sec	1.35	1.02
TOTAL	24782	121	94	222	295	1079	0	7122	49.68%	51.5/sec	14.33	8.99

Рисунок 4.15 – Агрегований результат навантажувального тестування з використанням удосконаленого методу

Алгоритм вимикача дозволяє вимикати кінцеві точки, які повертають помилки, а це, у свою чергу, зменшує навантаження на додаток. Таким чином, додаток отримує менше навантаження, що дозволяє використовувати ресурси оптимальніше. Вказане покращення можна побачити на графіку часу відповіді кожної кінцевої точки (рисунок 4.16). В той же час аналогічний графік (представлений на рисунку 4.17) при відсутності удосконаленого методу демонструє нестабільність та перевантаження додатку.

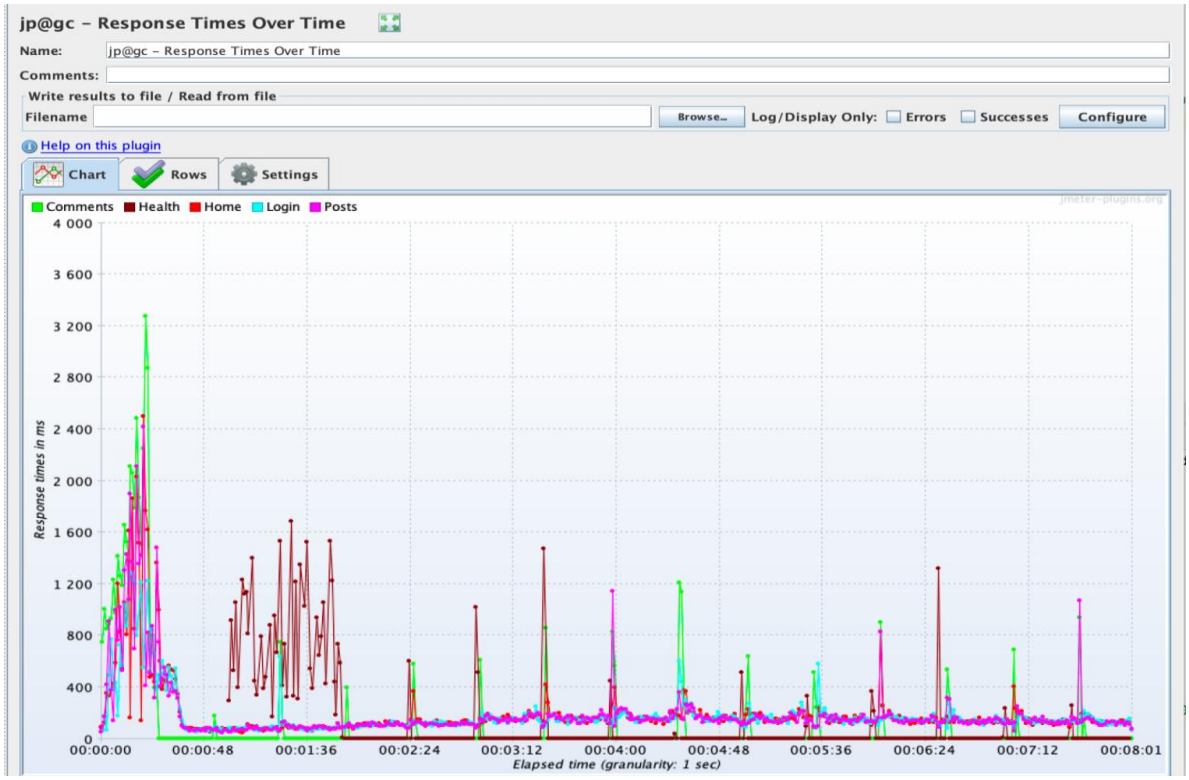


Рисунок 4.16 – Графік часу відповіді кінцевих точок при навантажувальному тестуванні з використанням удосконаленого методу

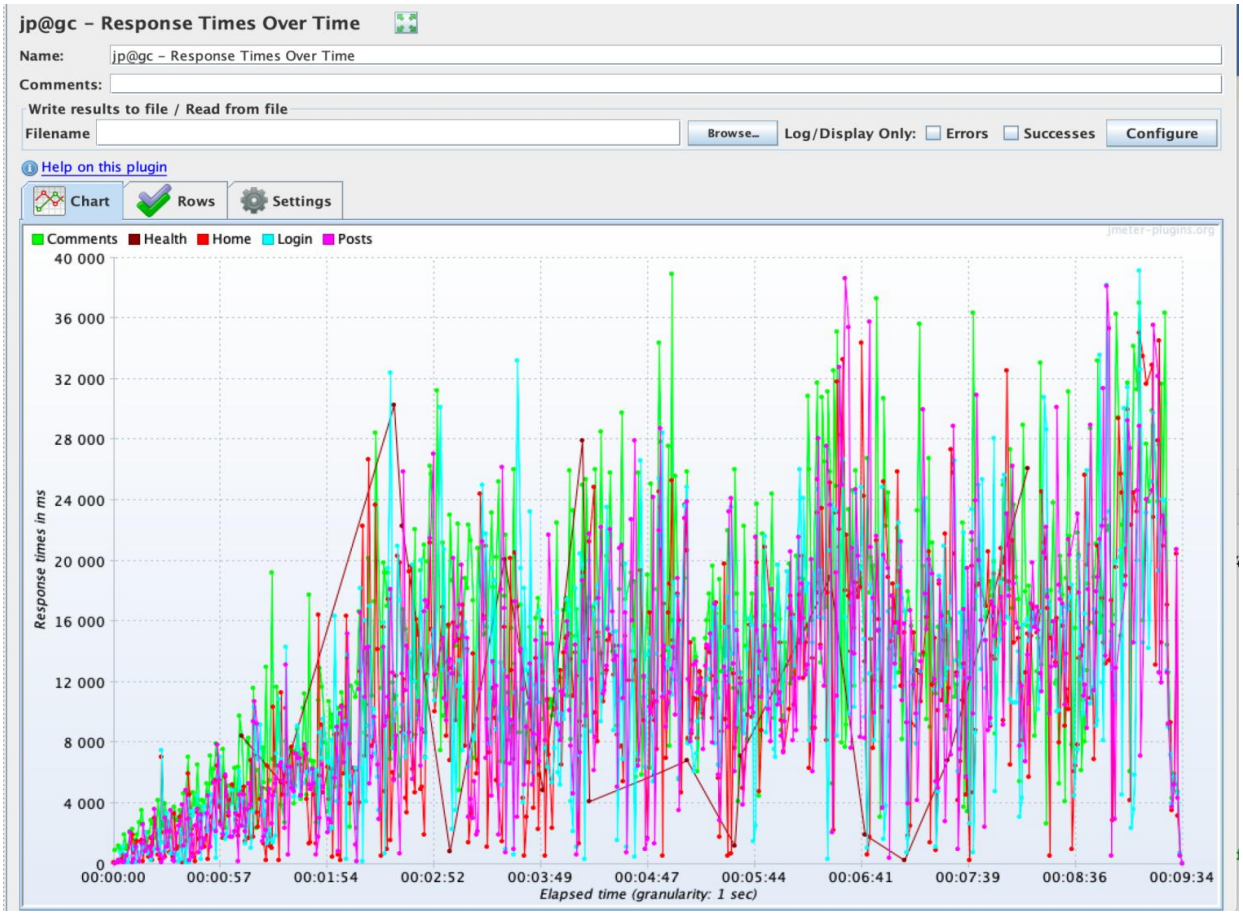


Рисунок 4.17 – Графік часу відповіді кінцевих точок при навантажувальному тестуванні без використання удосконаленого методу

З огляду на отримані результати, можна зробити висновок, що удосконалений метод відмовостійкості ПС на 50% разів зменшує кількість помилок у системі та на 50% збільшує пропускну здатність системи.

4.4 Інтеграція та налаштування програмного засобу

Програмний засіб рекомендовано установлювати як розширення до існуючої програмної системи, тому середовище функціонування залежить від того, як функціонує основна програма. Програмний засіб можна використовувати на персональному комп’ютері, у хмарному середовищі або на сервері. Кроки інсталяції при цьому не змінюються; потрібно лише ознайомитись з інструкціями по налаштуванню сторонніх систем (таких як Node, NPM, Docker) та слідувати

інструкціям, які стосуються конкретного середовища сторонніх систем.

Для впровадження програмного засобу необхідно спершу встановити NPM, Node, Docker. Для цього рекомендовано слідувати інструкціям власників цих систем.

Далі необхідно виконати команду npm install. Після запуску команди виконається установка необхідних сторонніх бібліотек для коректної роботи ПС.

Для запуску програмного засобу необхідно виконати команду npm start.

Додатково можна використовувати змінні середовища для налаштування параметрів програмного засобу:

– HOST – ім’я хоста додатку;

– PORT – порт, за яким доступний програмний засіб;

– DOCKER_IMAGE – ім’я- докер образу додатку, який буде запущено;

– DOCKER_CONTAINER_PORT – порт за яким доступний додаток усередині докер-контейнера;

– MIN_CONTAINERS – мінімально можлива кількість контейнерів;

– MAX_CONTAINERS – максимально можлива кількість контейнерів;

– SCALE_TIMOUT – затримка між етапами додавання/видалення контейнерів;

– SCALE_UP_STEP – кількість контейнерів, які будуть запускатись за такт при збільшенні навантаження

– SCALE_DOWN_STEP – кількість контейнерів, які будуть запускатись за такт при зменшенні навантаження.

4.5 Висновки

У розділі описані модулі і компоненти розроблюваного програмного засобу та їх взаємодія.

Здійснена покрокова реалізація програмного засобу, описано деталі реалізації окремих модулів роботи і управління контейнерами, профайлера, вимикача та балансувача.

Також розглянута реалізація методів поліпшення технічних характеристик системи шляхом імплементації шардінгу для розподілених додатків і переосмисленню його підходів та методів.

Проведено емпіричне дослідження розробленого ПЗ. Всі його модулі та функції були протестовані за допомогою модульного тестування, описані тестові сценарії та випадки. Окрім модульного тестування, проведено також навантажувальне тестування, щоб переконатись у працездатності шардінгу та балансувача.

На основі аналізу ефективності удосконаленого методу відмовостійкості ПС було з’ясовано, що метод на 50-60% зменшує кількість помилок у системі та на 50% збільшує її пропускну здатність.

ВИСНОВКИ

У процесі дипломного проєктування досліджено галузь відмовостійкості та сучасні методи і засоби забезпечення відмовостійкості ПС, визначено невирішені проблеми у галузі, на основі чого розроблено удосконалений метод забезпечення відмовостійкості ПС, виконано його програмну реалізацію та тестування.

У першому розділі дипломної роботи досліджено та проаналізовано сферу відмовостійкості ПС, останні джерела та публікації, існуючі методи та засоби забезпечення відмовостійкості ПС з метою виявлення невирішених питань та проблем. На основі проведеного аналізу визначені методологічні підходи до вирішення задачі удосконалення відмовостійкості ПС та виконана розгорнута постановка задач дослідження.

У другому розділі здійснено аналіз концепцій, моделей, методів та алгоритмів вирішення проблем відмовостійкості ПС. Невирішені проблеми запропоновано вирішити шляхом розвитку методу шардінгу (у напрямку його застосування до розподілених додатків) та доопрацювання і комбінації класичних патернів відмовостійкості ПС. Розроблений метод підвищує відмовостійкість ПС, оскільки ПС буде здатна сама визначити тип потенційної проблеми та реагувати відповідно.

У третьому розділі описана технологія реалізації удосконаленого методу відмовостійкості ПС, зокрема, здійснено аналіз вимог до ПЗ, розробку його структури та структури даних. Також обґрунтовано вибір засобів реалізації програми. Для програмної реалізації методу використано технології Docker, NodeJS, Express та інші, а для тестування – технології Jest та JMetter.

У четвертому розділі описано програмну реалізацію та тестування ПЗ, розробленого на основі удосконаленого методу відмовостійкості ПС. У процесі тестування було з’ясовано, що удосконалений метод відмовостійкості ПС дає змогу суттєво зменшити кількість помилок у системі, оскільки розроблений метод превентивно ізолює шкідливих користувачів, що, у свою чергу, збільшує пропускну здатність ПС.

Розроблений метод покращує відмовостійкість додатків, побудованих за мікросервісною або кліент-серверною архітектурою, шляхом реалізації нових підходів, переосмислення та комбінації існуючих алгоритмів відмовостійкості ПС, що дало змогу удосконалити працездатність та стабільність систем, збільшити їх пропускну здатність та зменшити кількість помилок.

Проведені емпіричні дослідження удосконаленого методу підтверджують його адекватність та ефективність у порівнянні з наявними рішеннями у галузі, а також працездатність та функціональну придатність розробленого на його основі програмного засобу. Апробація отриманих результатів показала зменшення кількості помилок у програмних системах приблизно на 50% і збільшення кількості оброблених запитів на 50% у порівнянні з існуючими алгоритмами відмовостійкості програмних систем.

Отже, розроблений програмний засіб можна рекомендувати для використання ІТ-підприємствам, які мають на меті підвищити відмовостійкість власних розроблюваних додатків.

Таким чином, результати дослідження за темою дипломної роботи мають наукову новизну та практичну цінність.

Результати дослідження опубліковані у фаховому науковому виданні [21] та у збірнику матеріалів Міжнародної науково-практичної конференції [22].

Копії наукових публікацій подані у додатку Б.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Digital Around the World. *Datareportal*. URL: https://datareportal.com/global-digital-overview

2. Cem Dilmegani. 85+ Digital Transformation Stats from Reputable Sources. *AIMultiple*. URL: https://research.aimultiple.com/digital-transformation-stats/

3. How Loading Time Affects Your Bottom Line. *KISSmetrics*. URL: https://blog.kissmetrics.com/wp-content/uploads/2011/04/loading-time.pdf

4. Firesmith D. System Resilience what Exactly is it. *Software Engineering Institute (SEI) at Carnegie Mellon University*: [site]. URL: https://insights.sei.cmu.edu/-blog/system-resilience-what-exactly-is-it/

5. What is Fault Tolerance? *Fortinet*. URL: https://www.fortinet.com/-resources/cyberglossary/fault-tolerance

6. High Availability Versus Fault Tolerance. *IBM*. URL: https://www.ibm.com/-docs/en/powerha-aix/7.2?topic=aix-high-availability-versus-fault-tolerance

7. Stratus Fault-Tolerant Systems: [site]. URL: https://www.stratus.com/fault-tolerant/

8. Warren Axelrod. Investing In Software Resiliency. *ResearchGate*. URL : https://www.researchgate.net/publication/293515438_Investing_in_software_resiliency

9. Jonathan Johnson. Resilience Engineering: An Introduction. *BMC*. URL: https://www.bmc.com/blogs/resilience-engineering/

10. Measurement Frameworks and Metrics for Resilient Networks and Services: Technical report. ENISA, 2011. 109 p.

11. Miruna Stoicescu. Architecting Resilient Computing Systems: a Component-Based Approach. Ubiquitous Computing. Institut National Polytechnique de Toulouse. INPT, 2013. 153 p.

12. Bulkhead pattern. *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead

13. Circuit breaker pattern *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

14. Michael T. Nygard. Release It! Design and Deploy Production-Ready Software ; 2nd Edition. Pragmatic Bookshelf, 2018. 378 p.

15. Compensating Transaction pattern *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

16. Health endpoint monitoring pattern *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring

17. Sid Choudhury. How Data Sharding Works in a Distributed SQL Database. *Yugabyte*. URL: https://blog.yugabyte.com/how-data-sharding-works-in-a-distributed-sql-database

18. Queue-based load leveling pattern *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling

19. Retry pattern. *Microsoft.com*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

20. Chaos Monkey. *Netflix*. URL: https://netflix.github.io/chaosmonkey/

21. Антіч Д. Ю., Радельчук Г. І. Удосконалення алгоритмів забезпечення відмовостійкості програмних систем // Вісник Хмельницького національного університету. – 2021. – №4 (299). – С. 54-58.

22. Радельчук Г. І., Антіч Д. Ю. Деякі методологічні підходи до удосконалення відмовостійкості програмних систем // Тенденції розвитку науки та освіти: виклики сучасного інформаційного суспільства: збірник тез доповідей міжнародної науково-практичної конференції (Полтава, 30 вересня 2021 р.). Полтава: ЦФЕНД, 2021. – С. 49-53.

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А.1 Програмний код модуля loadBalancer.js

```
const ContainersStore = require('./utils/ContainersStore')
const CircuitBreacker = require('./utils/CircuitBreacker')
const configs = require('./utils/configs')
const express = require("express");
const cookieParser = require("cookie-parser");
const xUuidMiddleware = require("./middlewares/xUuid");
const profilerMiddleware = require("./middlewares/profiler");
const stats = require("./middlewares/stats");
const urlChooser = require("./middlewares/urlChooser");
const handler = require("./handlers/proxy");
const Sharding = require("./utils/Sharding");

const {
    DOCKER_CONTAINER_PORT,
    DOCKER_IMAGE,
    MIN_CONTAINERS,
    MAX_CONTAINERS,
    PORT,
    SCALE_TIMOUT,
    SCALE_UP_STEP,
    SCALE_DOWN_STEP
} = configs

let store;

const startUp = (async () => {
    store = new ContainersStore({
        image: DOCKER_IMAGE,
        containerPort: DOCKER_CONTAINER_PORT,
        minContainers: MIN_CONTAINERS,
        maxContainers: MAX_CONTAINERS,
        scaleTimeOut: SCALE_TIMOUT,
        scaleUpStep: SCALE_UP_STEP,
        scaleDownStep: SCALE_DOWN_STEP
    })
    const cb = new CircuitBreacker();
    const sharding = new Sharding(store)
    store.monitor();

    const app = express();
    app.use(cookieParser(), xUuidMiddleware, urlChooser(store, sharding),
stats(cb, sharding));
    const server = app.all('*', handler(store, cb));
    server.listen(PORT);
})()
```

А.2 Програмний код модуля ContainersStore.js

```
const Container = require('./Container');
const configs = require('./configs')
const {CONTAINER_STATUS, CONTAINER_HEALTH} = configs

class ContainersStore {
    containerPort = 80;
    image = null;
    lastUsedPort = 55000;
    containers = {};
    desiredContainers = 1;
    minContainers = 1;
    maxContainers = 1;
    timer = null;
    lastScale = 0;
    scaleTimeOut = 500;
    scaleUpStep = 1;
    scaleDownStep = 1;

    constructor({
                    image,
                    containerPort,
                    minContainers,
                    maxContainers,
                    scaleTimeOut,
                    scaleUpStep,
                    scaleDownStep,
                }) {
        this.image = image;
        this.containerPort = containerPort;
        this.minContainers = minContainers;
        this.desiredContainers = minContainers;
        this.maxContainers = maxContainers;
        this.scaleTimeOut = scaleTimeOut;
        this.scaleUpStep = scaleUpStep;
        this.scaleDownStep = scaleDownStep;
    }

    addContainer = (id, newContainer) => this.containers = {...this.containers,
[id]: newContainer};
    getContainers = () => this.containers;
    getHealthyContainers = () => Object.fromEntries(
        Object.entries(this.containers).filter(
            ([id, container]) =>
                container.health === CONTAINER_HEALTH.healthy
                && container.status === CONTAINER_STATUS.running
        )
    );
    getHealthyContainersKeys = () => Object.keys(this.getHealthyContainers());
    getContainer = id => this.containers[id] ?? null;
    getContainersKeys = () => Object.keys(this.containers);
    getContainersCount = () => this.getContainersKeys().length;
    removeContainer = (id) => {
        delete this.containers[id];
    }

    createContainer = async () => {
        const container = new Container(this.image, this.lastUsedPort,
this.containerPort)
        await container.create()
        await container.start()
        container.monitor()
```

```
        this.lastUsedPort++;

        return container;
    }

    setContainersToDesired = async () => {
        const containersDiff = this.desiredContainers - this.getContainersCount();
        if (containersDiff > 0) {
            for (let i = 0; i < containersDiff; i++) {
                const container = await this.createContainer();
                this.addContainer(container.id, container);
            }
        } else if (containersDiff < 0) {
            const keys = this.getContainersKeys()
            for (let i = 0; i < containersDiff * -1; i++) {
                const id = keys[i];
                const cont = this.getContainer(id)
                cont.setStatus(CONTAINER_STATUS.removing)
                cont.stop();
                this.removeContainer(id)
            }
        }
    }

    isUnhealthy = ({health, status, startAttemptsFailed = 0}) => health ===
CONTAINER_HEALTH.unhealthy
        || status === CONTAINER_STATUS.dead
        || status === CONTAINER_STATUS.removing
        || startAttemptsFailed >= 2

    killUnhealthy = async () => {
        const containerEntries = Object.entries(this.containers);
        await Promise.all(containerEntries.map(async ([id, container]) => {
            if (this.isUnhealthy(container)) {
                console.log('removing', id, container.status, container.health)
                await container.stop();
                this.removeContainer(id)
            }
        }))
    }

    scale = () => {
        if (this.lastScale + (this.scaleTimeOut * 1000) < Date.now()) {
            const containerValues = Object.values(this.containers)
            const sumMemoryUsage = containerValues.reduce((sum, {memoryUsage}) =>
sum + (memoryUsage ?? 0), 0)
            const sumCPUUsage = containerValues.reduce((sum, {cpuUsage}) => sum +
(cpuUsage ?? 0), 0)
            const averageMemoryUsage = sumMemoryUsage / this.desiredContainers;
            const averageCPUUsage = sumCPUUsage / this.desiredContainers;
            // console.log({averageCPUUsage, averageMemoryUsage},
this.desiredContainers)
            if ((averageMemoryUsage > 30 || averageCPUUsage > 35) &&
this.desiredContainers < this.maxContainers) {
                this.lastScale = Date.now();
                this.desiredContainers =
                    this.desiredContainers + Math.min(this.scaleUpStep,
this.maxContainers - this.desiredContainers)
            } else if ((averageMemoryUsage <= 15 && averageCPUUsage <= 20) &&
this.desiredContainers > this.minContainers) {
                this.lastScale = Date.now();
                this.desiredContainers =
                    this.desiredContainers - Math.min(this.scaleDownStep,
this.desiredContainers - this.minContainers)
```

```
            }
        }
    }

    monitor = () => {
        this.timer = setInterval(async () => {
            await this.killUnhealthy();
            await this.scale();
            await this.setContainersToDesired();
        }, 1000)
    }

    stop = () => {
        /*istanbul ignore else*/
        if (this.timer) {
            clearInterval(this.timer)
        }
        Object.values(this.containers).map(container => container.stop())
    }
}

module.exports = ContainersStore;
```

А.3 Програмний код модуля Container.js

```
const Docker = require('dockerode');
const docker = new Docker();

const configs = require('./configs')
const {CONTAINER_STATUS, CONTAINER_HEALTH} = configs

class Container {
    id = null;
    instance = null;
    image = null;
    hostPort = null;
    containerPort = null;
    status = null;
    health = null;
    startAttemptsFailed = 0;
    memoryUsage = 0;
    cpuUsage = 0;
    timer = null;

    constructor(image, hostPort, containerPort) {
        this.image = image;
        this.hostPort = hostPort;
        this.containerPort = containerPort;
    }

    setStatus = (newStatus) => this.status = newStatus;
    create = async () => {
        try {
            const instance = await docker.createContainer({
              Image: this.image,
              AttachStdin: false,
              AttachStdout: true,
              AttachStderr: true,
              ExposedPorts: {
                  [`${this.containerPort}/tcp`]: {}
              },
              HostConfig: {
                  CpuShares: 1024,
```

```
                Memory: 128 * 1048576,
                PortBindings: {[`${this.containerPort}/tcp`]: [{HostPort:
`${this.hostPort}`}]}
            },
            Tty: true,
        });
            this.id = instance.id;
            this.instance = instance
        } catch (e) {
            console.error('Container creation failed:', e)
        }
    }

    start = async () => {
        try {
            await this.instance?.start();
        } catch (e) {
            this.startAttemptsFailed = this.startAttemptsFailed + 1
            console.error('Container start failed:', e)
        }
    }


    getStatus = async () => {
        try {
            const data = await this.instance?.inspect();
            this.status = data?.State?.Status
            this.health = data?.State?.Health?.Status
        } catch (e) {
            console.error('Container status detection failed:', e)
            this.health = CONTAINER_HEALTH.unhealthy;
        }
    }

    getConsumption = async () => {
        try {
            const containerStats = await this.instance.stats({stream: false});
            let memoryUsage = 0;
            let cpuUsage = 0;
            const {memory_stats, cpu_stats, precpu_stats} = containerStats;
            if (memory_stats && cpu_stats && precpu_stats) {
                const {usage, stats, limit} = memory_stats;

                /*istanbul ignore else*/
                if (typeof usage !== undefined
                    && typeof stats?.cache !== undefined
                    && typeof limit !== undefined) {
                    memoryUsage = ((usage - stats?.cache) / limit) * 100;
                }

                const {cpu_usage, system_cpu_usage, online_cpus} = cpu_stats;
                const {cpu_usage: pre_cpu_usage, system_cpu_usage:
pre_system_cpu_usage} = precpu_stats;

                /*istanbul ignore else*/
                if (typeof cpu_usage?.total_usage !== undefined
                    && typeof system_cpu_usage !== undefined
                    && typeof online_cpus !== undefined
                    && typeof pre_cpu_usage?.total_usage !== undefined
                    && typeof pre_system_cpu_usage !== undefined) {
                    const cpu_delta = cpu_usage.total_usage -
pre_cpu_usage?.total_usage;
                    const system_cpu_delta = system_cpu_usage -
pre_system_cpu_usage;
```

```
                    cpuUsage = (cpu_delta / system_cpu_delta) * online_cpus * 100;
                }
            }

            this.memoryUsage = memoryUsage || this.memoryUsage
            this.cpuUsage = cpuUsage || this.cpuUsage

        } catch (e) {
            console.error('Container consumption detection failed:', e)
            this.health = CONTAINER_HEALTH.unhealthy;
        }
    }

    stop = async () => {
        /*istanbul ignore else*/
        if (this.timer) {
            clearInterval(this.timer);
        }
        try {
            await this.instance?.remove({force: true})
        } catch (e) {
            console.error('Container stop failed:', e)
        }
    }

    monitor = () => {
        this.timer = setInterval(async () => {
            if (this.status === CONTAINER_STATUS.created) {
                await this.start()
            }
            await this.getStatus();
            await this.getConsumption()
        }, 1000);
    }
}

module.exports = Container;
```

А.4 Програмний код модуля CircuitBreacker.js

```
const CB_STATE = {OPEN: 'OPEN', HALF_OPEN: 'HALF_OPEN', CLOSED: 'CLOSED'}

class CircuitBreacker {
    URLs = {};

    constructor() {
        setInterval(() => {
            this.clearURLCalls()
        }, 1000)
    }

    sanitizeUrl = url => url.split('?')?.[0]
    getURLResults = url => this.URLs?.[this.sanitizeUrl(url)] ?? null;
    addURLResult = (path, serverId, result) => {
        const url = this.sanitizeUrl(path);
        if (!this.getURLResults(url)) {
            this.URLs[url] = {calls: []};
        }
        this.URLs[url].calls.push({result, timeStamp: Date.now(), serverId})

        if (this.URLs[url]?.state === CB_STATE.HALF_OPEN) {
            if (result) {
                console.log('CB is closing', url);
```

```
                this.URLs[url].state = CB_STATE.CLOSED;
                this.URLs[url].closedTimeStamp = Date.now();
            } else {
                console.log('CB is re-opening', url);
                this.URLs[url].state = CB_STATE.OPEN;
                this.URLs[url].openTimeStamp = Date.now();
            }
        }
    }

    clearURLCalls = () =>
        Object.keys(this.URLs).forEach(key => {
            this.URLs[key].calls =
                this.URLs[key].calls.filter(({timeStamp}) => (timeStamp ?? 1) + 1
* 60 * 1000 > Date.now())
            if (!this.URLs[key].calls.length && this.URLs[key].state !==
CB_STATE.OPEN) {
                delete this.URLs[key];
            }
        })

    shouldCBOpen = (path) => {
        const results = this.getURLResults(path);
        if (results?.calls?.length > 50) {
            const successCalls = results?.calls.filter(({result}) => result);
            return (successCalls.length / results.calls.length * 100) < 25
        }
        return false;
    }

    validatePath = (path) => {
        const url = this.sanitizeUrl(path);
        const results = this.getURLResults(url);
        if (results?.state === CB_STATE.CLOSED
            && results?.closedTimeStamp
            && results?.closedTimeStamp + 30 * 1000 > Date.now()) {
            console.log('paused', url)
            return true;
        }

        if (results?.state === CB_STATE.OPEN
            && results?.openTimeStamp
            && results?.openTimeStamp + 30 * 1000 < Date.now()) {
            this.URLs[url].state = CB_STATE.HALF_OPEN;

            console.log('CB is setting half open', url);
            return true;
        }

        if (results?.state !== CB_STATE.OPEN
            && this.shouldCBOpen(url)) {
            this.URLs[url].state = CB_STATE.OPEN;
            this.URLs[url].openTimeStamp = Date.now();
            console.log('CB is opening', url);
        }

        return this.URLs?.[url]?.state !== CB_STATE.OPEN
    }
}

const CBModule = module.exports = CircuitBreacker;
CBModule.CB_STATE = CB_STATE
```

А.5 Програмний код модуля Sharding.js

```
class Sharding {
    userRequests = [];
    sharding = {};
    containersStore

    constructor(containersStore) {
        this.containersStore = containersStore;
        setInterval(() => {
            this.clearOldResults()
        }, 1000)
    }

    addUserRequest = (userUuid, result, serverId) => {
        if (!userUuid) {
            return;
        }
        if (!this.userRequests[userUuid]) {
            this.userRequests[userUuid] = [];
        }
        this.userRequests[userUuid].push({result, timeStamp: Date.now(),
serverCalled: serverId})
    }
    getUserRequests = (userUuid) => this.userRequests?.[userUuid] ?? null;

    clearOldUserResults = () => {
        const userKeys = Object.keys(this.userRequests)
        userKeys.forEach(key => {
            this.userRequests[key] =
                this.userRequests[key].filter(({timeStamp}) => (timeStamp ?? 1) +
2 * 60 * 1000 > Date.now())
            if (!this.userRequests[key].length) {
                delete this.userRequests[key];
            }
        })
    }

    defineShard = (userUuid, isMalicious) => {
        const keys = this.containersStore.getHealthyContainersKeys()
        const userShards = keys.map((value) => ({value, sort: Math.random()}))
                                .sort((a, b) => a.sort - b.sort)
                                .map(({value}) => value).slice(0, isMalicious ? 1 :
Math.max(2, Math.ceil(keys.length / 3))
            );
        this.sharding[userUuid] = {
            userShards,
            current: 0,
            containersCount: keys.length,
            isMalicious
        }
    }

    isUserMalicious = (userUuid) => {
        const userRequests = this.getUserRequests(userUuid);
        if ((userRequests?.length ?? 0) > 25) {
            const errors = userRequests.filter(req => !req.result);
            const serversCalled = errors.map(req => req.serverCalled);
            const serversAffected = [...new Set(serversCalled)].filter(res =>
res);
            const errorsNumber = errors.length;
            const successNumber = userRequests.length - errorsNumber;
            const successRate = successNumber / userRequests.length * 100;
```

```
            return successRate < 80 && serversAffected.length > 3;
        }
        return false;
    }

    useNextShard = (userUuid) => {
        if (this.sharding?.[userUuid]?.userShards?.length > 1) {
            this.sharding[userUuid].current =
                (this.sharding[userUuid].current + 1 <
this.sharding[userUuid].userShards.length)
                    ? this.sharding[userUuid].current + 1
                    : 0;
        }
    }
    updateLastAccessedShard = (userUuid) => {
        /*istanbul ignore else*/
        if (this.sharding?.[userUuid]) {
            this.sharding[userUuid].lastAccessed = Date.now();
        }
    }

    checkAndReplaceUnhealthyUserShards = (userUuid) => {
        /*istanbul ignore else*/
        if (this.sharding?.[userUuid]?.userShards?.length) {
            const containersKeys =
this.containersStore.getHealthyContainersKeys();
            const allAlive = this.sharding[userUuid].userShards.every(shardId =>
containersKeys.includes(shardId));
            /*istanbul ignore else*/
            if (!allAlive) {
                this.sharding[userUuid].userShards =

this.getNRandContainerKeys(this.sharding[userUuid].userShards.length);
            }
        }
    }

    getNRandContainerKeys = (n = 1) =>
        this.containersStore.getHealthyContainersKeys()
            .map((value) => ({value, sort: Math.random()}))
            .sort((a, b) => a.sort - b.sort)
            .map(({value}) => value).slice(0, n);


    getUserShards = (userUuid) => {
        const isMalicious = this.isUserMalicious(userUuid);
        if (isMalicious) {
            console.warn('Isolating malicious user', userUuid);
        }
        /*istanbul ignore else*/
        if (!this.sharding?.[userUuid]?.userShards.length || isMalicious !==
this.sharding?.[userUuid]?.isMalicious) {
            this.defineShard(userUuid, isMalicious);
        }

        this.checkAndReplaceUnhealthyUserShards(userUuid);
        this.useNextShard(userUuid)
        this.updateLastAccessedShard()
        return this.sharding[userUuid];
    }

    clearUserShards = () => {
        const shardingKeys = Object.keys(this.sharding)
        shardingKeys.forEach(key => {
```

```
            /*istanbul ignore else*/
            if (this.sharding?.[key]?.lastAccessed &&
(this.sharding[key]?.lastAccessed + 2 * 60 * 1000) < Date.now()) {
                delete this.sharding[key]
            }
        })
    }

    clearOldResults = () => {
        this.clearOldUserResults()
        this.clearUserShards()
    }
}

module.exports = Sharding;
```

А.6 Програмний код компонента proxy.js

```
const request = require('request');
const getNextServer = require("../utils/getNextServer");
const getContainerUrl = require("../utils/getContainerUrl");

const handler = (store, cb) => async (req, res) => {
    const url = req?.locals?.proxyUrl ?? getContainerUrl(getNextServer(store))
    if (!url) {
        console.log('503: Service temporary unavailable')
        return res.status(503).send('Service temporary unavailable');
    }
    const isValid = cb?.validatePath(req.url);
    if (!isValid) {
        console.log('504: Service temporary unavailable')
        return res.status(504).send('Service temporary unavailable');
    }

    const proxy = request({url: url + req.url}).on('error', error => {
        res.status(500).send(error.message);
    });
    req.pipe(proxy).pipe(res);
};
module.exports = handler;
```

А.7 Програмний код компонента xUuid.js

```
const createUUID = require("../utils/createUUID");

const xUuidMiddleware = (req, res, next) => {
    let xUuid = req?.cookies?.xUuid
    if (!xUuid){
        const ip = req.connection.remoteAddress;
        const browser = req.headers["user-agent"];
        const language = req.headers["accept-language"];
        xUuid = createUUID(`${ip}-${browser}-${language}`)
        res.cookie('xUuid', xUuid, {maxAge: 900000})
    }

    if (!req.locals) {
        req.locals = {};
    }
    req.locals.xUuid = xUuid
    next();
};

module.exports = xUuidMiddleware;
```

А.8 Програмний код компоненту profiler.js

```
const profilerMiddleware = (req, res, next) => {
    const start = Date.now();
    res.on('finish', () => {
        console.log('Completed', req.method, req.url, 'status:', res.statusCode,
'time:', Date.now() - start,);
    });
    next();
};

module.exports = profilerMiddleware;
```

А.9 Програмний код службового модуля stats.js

```
const stats = (cb, sharding) => (req, res, next) => {
    res.on('finish', () => {
        /* istanbul ignore else*/
        if (res.statusCode !== 504) {
            const result = res.statusCode < 300;
            sharding.addUserRequest(req.locals.xUuid, result, req.locals.serverId)
            cb.addURLResult(req.url, req.locals.serverId, result)
        }

    });
    next();
};

module.exports = stats;
```

А.10 Програмний код createUUID.js

```
const stats = (cb, sharding) => (req, res, next) => {
const uuidV4 = require('uuid').v4;
const uuidV5 = require('uuid').v5;

const crypto = require('crypto');
const MY_NAMESPACE = 'f4dfa959-b907-4eb0-9f75-c82692f93784'

const createUUID = (string) => {
    // const hash = crypto.createHash('md5').update(string).digest('hex');
    // return uuidV5(hash, MY_NAMESPACE);
    return uuidV4();
}

module.exports = createUUID;
```

А.11 Програмний код getContainerUrl.js

```
const configs = require('./configs')
const {HOST} = configs
const getContainerUrl = (server) => server?.hostPort ?
`${HOST}:${server.hostPort}` : null
module.exports = getContainerUrl;
```

А.12 Програмний код getNextServer.js

```
let cur = 0;
const getNextServer = (store, limit = 3) => {
    const healthyServers = store?.getHealthyContainers()
    const servers = Object.values(healthyServers)
    cur = cur + 1 < servers.length ? cur + 1 : 0;
```

```
    const server = servers?.[cur];
    if (!server) {
        if (limit - 1 > 0) {
            return getNextServer(store, limit - 1);
        }
        return null;
    }
    return server;
}
module.exports = getNextServer;
```

А.13 Програмний код urlChooser.js

```
const getContainerUrl = require("../utils/getContainerUrl");
const urlChooser = (store, sharding) => (req, res, next) => {
    /* istanbul ignore else*/
    if (req.locals.xUuid) {
        const shard = sharding?.getUserShards(req.locals.xUuid);
        const server = store?.getContainer(shard.userShards[shard.current]);
        /* istanbul ignore else*/
        if (server) {
            req.locals.serverId = server.id;
            req.locals.proxyUrl = getContainerUrl(server);
        }
    }
    next();
};

module.exports = urlChooser;
```

ДОДАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ISSN 2307-5732
DOI 10.31891/2307-5732

НАУКОВИЙ ЖУРНАЛ **4.2021**

ВІСНИК

Хмельницького національного університету

Технічні науки

Technical sciences

SCIENTIFIC JOURNAL
HERALD OF KHMELNYTSKYI NATIONAL UNIVERSITY

2021, Issue 4, Volume 299

Хмельницький

ВІСНИК
ХМЕЛЬНИЦЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
серія: Технічні науки

Затверджений як фахове видання категорії «Б»,
РІШЕННЯ АТЕСТАЦІЙНОЇ КОЛЕГІЇ № 1643 ВІД 28.12.2019 та №409 від 17.03.2020

Засновано в липні 1997 р. ***Виходить 6 разів на рік***

Хмельницький, 2021, № 4(299)

Засновник і видавець: Хмельницький національний університет
(до 2005 р. – Технологічний університет Поділля, м. Хмельницький)

Наукова бібліотека України ім. В.І. Вернадського http://nbuv.gov.ua/j-tit/Vchnu_tekh

Включено до науково-метричних баз:

Google Scholar	http://scholar.google.com.ua/citations?hl=uk&user=aIUP9OYAAAAJ
Index Copernicus	http://jml2012.indexcopernicus.com/passport.php?id=4538&id_lang=3
Polish Scholarly Bibliography	https://pbn.nauka.gov.pl/journals/46221
CrossRef	http://doi.org/10.31891/2307-5732

Технічний редактор Горященко К. Л., к.т.н.
Редактор-коректор Броженко В. О.

Рекомендовано до друку рішенням вченої ради Хмельницького національного університету, протокол № 1 від 26.08.2021 р.

Адреса редакції: редакція журналу "Вісник Хмельницького національного університету"
Хмельницький національний університет
вул. Інститутська, 11, м. Хмельницький, Україна, 29016

☎ (038-2) 67-51-08 **web:** http://journals.khnu.km.ua/vestnik
http://lib.khnu.km.ua/visnyk_tup.htm

e-mail: visnyk.khnu@khmnu.edu.ua
visnyk.khnu@gmail.com

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 24922-14862ПР від 12 липня 2021 року

ЗМІСТ

DOI 10.31891/2307-5732-2021-297-3-54-58
УДК 004.052.3

Д.Ю. АНТІЧ, Г.І. РАДЕЛЬЧУК
Хмельницький національний університет

УДОСКОНАЛЕННЯ АЛГОРИТМІВ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ ПРОГРАМНИХ СИСТЕМ

У роботі представлено концепції відмовостійкості програмних систем та методи реагування системи на відмову. У дослідженні вдосконалено метод реагування системи на відмову шляхом проєктування комплексного рішення, що включає в себе доопрацювання та розширення класичних патернів відмовостійкості. Обґрунтовано доцільність та актуальність проектування нового методу відмовостійкості. Удосконалено алгоритми автоматичного реагування та попередження відмов, що дозволяє зменшити кількість помилок, які виникають у системі, у порівнянні з існуючими рішеннями. Результатом дослідження є покращений метод забезпечення відмовостійкості програмних систем.

Ключові слова: програмна система, алгоритм, відмовостійкість, патерн відмовостійкості

DMYTRO YURIIOVYCH ANTICH, GALINA IVANIVNA RADELCHUK
Khmelnytsky National University

IMPROVEMENT OF SOFTWARE SYSTEMS FAULT TOLERANCE ENSURING ALGORITHMS

The study investigates the concepts of fault tolerance and methods of system responses to failures. The study is based on the research of modern resiliency patterns and common approaches of reaction to failures. During the research, the common unresolved issues with modern resiliency and fault tolerance approaches were defined. The study improved the method of the system response to failures by designing a comprehensive solution that includes refinement and expansion of classical patterns of fault tolerance as a proposal to resolve common problems. The new solution of fault tolerance is based on the combination of basic monitoring approaches, load balancing approaches, circuit breaker pattern, and re-designing of the sharding pattern to be applicable not only for databases but also for modern applications. The new solution is based on an automatic decision-making expert system, which based on anonymous data saved by the monitoring layer decides the root cause of the issue and validates which scenario is applicable for the current situation. Based on the decision system can either enable a user and load balancing approaches by isolating harmful users using improved sharding and load-balancing solutions or enable a circuit breaker to temporarily disable the faulty features. The new method of resiliency is supposed to prevent and reduce more errors compared to the existing solutions in the domain of fault tolerance and resiliency, thus the efficiency of the new approach is higher. The expediency and urgency of designing a new method of fault tolerance are substantiated by expressing the importance of resolving existing problems. Improved methods of automatic response and failure prevention, which allowed to reduce the number of errors that occur in the system compared to existing solutions in resiliency and fault tolerance.

Keywords: software system, algorithm, fault tolerance, fault tolerance pattern

Вступ. Постановка проблеми

У сучасному світі мережа Інтернет займає значну частину життя людини. Кожен з нас щодня перевіряє інформацію про погоду, купує товари, спілкується з друзями чи знайомими, читає новини, книги, переглядає відео онлайн тощо. За час пандемії більшість звичних нам речей змушена була адаптуватись до нової реальності, зокрема, значна кількість видів діяльності почала здійснюватись онлайн. Наприклад, багато компаній змушені були перевести співробітників на віддалену роботу, університети та школи перейшли у режим дистанційного навчання тощо.

Таким чином, у зв'язку зі стрімкою діджиталізацією виникає потреба забезпечити надійність та відмовостійкість програмних систем (ПС). А це, у свою чергу, породжує потребу в алгоритмах відмовостійкості, які нададуть можливість підвищити працездатність ПС та забезпечити їх швидке відновлення після настання відмов.

Аналіз останніх досліджень та публікацій

У результаті досліджень університету Карнегі-Меллона (США) визначено поняття відмовостійкості та доведено, що жодна програмна система не є на 100% відмовостійкою або не є стійкою до відмов взагалі [1]. Група дослідників [2] визначила основні фактори, що шкодять відмовостійкості, та який вплив ці фактори мають на ПС. Такими факторами є наступні: розмір та складність програмної системи; взаємозалежність та взаємозв'язок; мережева орієнтованість; стрімка глобалізація компанії; програмне забезпечення з відкритим вихідним кодом; гібридизація; швидкі зміни; повторне використання у різних контекстах.

Комплексний підхід до відмовостійкості програмних систем включає в себе чотири базові методи реалізації відмовостійкості, а саме:

- реакція на помилки (як тільки виникає помилка, програмна система автоматично або розробники програми у ручному режимі повинні відреагувати на помилки);
- автоматичне логування (програмна система повинна автоматично логувати всі проблеми та помилки для можливості автоматичного реагування на проблеми);
- метрики MTBF, MTTF, MTTR (ці метрики визначають середній час між відмовами, середній час до відмови, середній час на усунення відмови);
- автоматизація плану відновлення (необхідно автоматизувати кроки, які система має здійснити для відновлення нормального функціонування).

Виділення невирішених частин загальної проблеми

Проаналізувавши описані у роботі [3] стратегії управління та запобігання потенційним помилкам, можна виділити наступний алгоритм реакції системи на відмову:

- автоматичне визначення помилки – система повинна самостійно виявити помилку, зберегти додаткову інформацію (ланцюжок виклику, подію, що спричинила помилку, тощо) та, за можливості, вивести користувачу наперед визначене повідомлення про помилку;
- аналіз та пояснення помилки (на цьому етапі необхідно висунути гіпотезу про причину помилки та знайти спосіб її вирішення);
- обробка помилок програмною системою за допомогою використання патернів відмовостійкості, що підходять для конкретної проблеми.

Найчастіше використовуються наступні методи реалізації відмовостійкості ПС:

- шаблон вимикача;
- шаблон повторень;
- шаблон компенсації транзакцій;
- шардінг.

Шаблон вимикача обробляє несправності, для вирішення яких може знадобитися різний час при підключенні до віддаленого сервісу чи ресурсу. Вимикач діє як проксі-сервер для операцій, які можуть вийти з ладу. Проксі відстежує кількість останніх помилок, які сталися, і використовує цю інформацію для прийняття рішення – дозволити продовження операції чи перервати її та повернути помилку [4].

Шаблон повторень реалізує програму для обробки збоїв, коли система намагається здійснити запит до сервісу чи ресурсу, шляхом прозорого повторення невдалої операції. Якщо програма виявляє помилку під час спроби надіслати запит віддаленому сервісу, вона може впоратися з помилкою, використовуючи такі стратегії:

- скасувати запит (якщо помилка вказує на те, що несправність не є тимчасовою або запит навряд чи буде успішним при його повторенні, то програма повинна скасувати операцію та повідомити про помилку);
- повторити спробу (якщо конкретне повідомлення про помилку є незвичним чи рідкісним, то це може бути спричинено незвичними обставинами, такими, наприклад, як пошкодження мережевого пакета під час його передачі; у цьому випадку програма може негайно повторити невдалий запит, оскільки та сама помилка навряд чи буде повторена, і запит, ймовірно, буде успішним);
- повторити спробу після затримки (якщо несправність спричинена проблемами з функціонуванням мережі або надмірним навантаженням на додаток, то може знадобитися деякий період часу, поки проблеми з підключенням будуть виправлені або всі поточні запити оброблені); програма повинна почекати деякий час перед повторною спробою запиту [5].

Шаблон компенсації транзакцій скасовує роботу, виконану за допомогою декількох кроків, які в сукупності визначають узгоджену в кінцевому рахунку операцію, якщо один або кілька кроків завершилися з помилкою. Таким чином, цей шаблон дає змогу повернутись до попереднього стабільного стану програмної системи та уникнути розбіжностей в даних. Цей патерн є особливо актуальним у випадках, коли деяка дія потребує запису в декілька таблиць бази даних чи виклику декількох сервісів [6].

Шардінг реалізує розбиття таблиць у базі даних на менші «шматки» (шарди). Розбиття може бути як горизонтальним, так і вертикальним. При вертикальному розбитті стовпці таблиці зберігаються в окремій базі даних, а при горизонтальному – рядки однієї таблиці зберігаються в декількох вузлах бази даних [7].

Таким чином, стратегія відмовостійкості не передбачає проактивну реакцію на потенційні відмови ПС, а працює з уже існуючими. Тому вказана предметна область має наступні невирішені проблеми:

- відсутність алгоритму для передбачення відмови, що призводить до відсутності автоматичного блокування першопричини потенційної відмови;
- відсутність реакцій на часткові відмови та підходів для автоматичного вирішення часткової непрацездатності системи.

Загалом, методи та алгоритми відмовостійкості на сьогоднішній день фокусуються на тому, щоб зменшити негативний вплив відмов на ПС після того, як система почала працювати некоректно. Однак, методи та підходи, що використовуються у сучасних програмних продуктах, не реагують на часткові відмови, не здатні автоматично визначати причину часткової відмови та здійснювати автоматичні спроби уснути таку відмову або основну причину відмови.

Формулювання цілей

Для проведення дослідження сформовано наступні цілі: провести теоретичний аналіз способів та методів забезпечення відмовостійкості програмних систем; охарактеризувати основні поняття та проблеми відмовостійкості; описати існуючі методи відмовостійкості; виділити наявні проблеми у галузі відмовостійкості ПС та описати шляхи їх вирішення; удосконалити існуючі алгоритми забезпечення відмовостійкості ПС.

Виклад основного матеріалу

На основі аналізу невирішених проблем відмовостійкості програмних систем було запропоновано розробити власний алгоритм шляхом комбінації та модифікації існуючих методів, що дозволить оптимізувати наявні вузькі місця в існуючих підходах. Визначено вимоги до удосконаленого алгоритму

відмовостійкості ПС:

- алгоритм повинен реагувати на часткові відмови, а саме: автоматично аналізувати та сканувати усі помилки, що з'являються у програмній системі;
- алгоритм повинен мати можливість автоматично висувати гіпотези про причину помилок та намагатись автоматично усувати помилку.

Для початку було вирішено використати підходи до моніторингу та збереження даних задля того, щоб алгоритм мав змогу аналізувати вхідні дані користувачів та робити припущення про можливі потенційні відмови системи.

Для вирішення поставленої задачі необхідно зберігати дані про користувачів та їх запити, тому удосконалений метод відмовостійкості збиратиме та аналізуватиме дані, які представлені у таблиці 1.

Таблиця 1

Вхідні дані алгоритму

Вхідний параметр	**Приклад даних**
IP-адреса користувача	188.123.19.208
Операційна система користувача	macOS 10.15.7
Браузер користувача	Chrome 91.0.4472.106
Запит, що здійснюється	somesite.com/login
Вхідні параметри запиту	{ "username": "dmytro", "password": "sometext" }
Відповідь системи на запит користувача	{ "code": 300, "message": "Login failed" "meta info": "Uncaught PHP Exception Ramsey\Uuid\Exception\InvalidUuidStringException: "Invalid UUID string: " at /var/www/project/vendor/ramsey/uuid/src/Codec/StringCodec.php line 146" }

На основі даних користувача формується унікальний анонімний хеш-ключ, який слугує ідентифікатором користувача. Аналізуючи дані, алгоритм може автоматично припустити, де існує проблема, та спробувати її автоматично вирішити.

Після завершення роботи моніторингу запускається логіка припущень. Для цього реалізовано алгоритм припущень, який на основі даних, збережених моніторингом, чи використовуючи балансувач та шардінг, ізолює потенційно небезпечних користувачів або виконує запуск патерну вимикача.

У разі, якщо система буде надсилати помилки всім користувачам, які виконують певний запит, то тоді алгоритм здійснить припущення, що проблема є локальною та стосується лише одного сервісу (чи однієї кінцевої точки). В результаті алгоритм автоматично вимкне даний сервіс за допомогою патерна вимикача. Наприклад, якщо сторінка авторизації не працює, користувачі здійснюють запити на авторизацію, проте отримують помилки, тоді алгоритм аналізує помилки і дані користувачів та робить припущення, що проблема стосується конкретного модуля системи і блокує всі подальші запити до вирішення проблеми.

Схематичне відображення роботи алгоритму у такому випадку представлено на рисунку 1.

На рисунку 2 представлена робота алгоритму у випадку повної працездатності ПС.

Описаний підхід дасть можливість ізолювати проблемні частини програмного продукту, а це, у свою чергу, дасть можливість користувачам продовжувати користуватись частиною програми, яка для них залишається працездатною.

Рис. 1. Робота алгоритму вимикача – відкритий стан

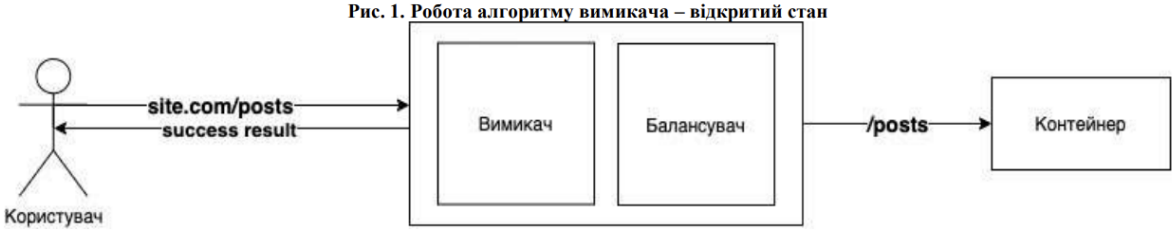


Рис. 2. Робота алгоритму вимикача – закритий стан

У той же час алгоритм відмовостійкості та моніторинг даних про користувачів можуть здійснити припущення про потребу в ізоляції користувачів та ізолювати їх від інших у випадку, якщо система почне повертати помилки лише певним користувачам або групі користувачів.

Для реалізації нового підходу до балансування на основі користувачів та навантаження було вирішено переосмислити підходи шардінгу та скомбінувати їх з патерном балансувача. Зазвичай, такий алгоритм використовується у базах даних для збереження даних в різних таблицях, проте, завдяки доопрацюванню, його можна використати у сучасних додатках.

У результаті розроблено алгоритм, який ізолює потенційно небезпечного користувача або групу користувачів, що сприяє зниженню кількості потенційних помилок і повних відмов системи та надає можливість іншим користувачам ПС продовжувати використовувати додаток без перешкод.

У випадку виявлення потенційно небезпечного користувача алгоритм відмовостійкості на основі даних моніторингу здійснить розподіл навантаження користувачів за допомогою алгоритму шардінгу таким чином, щоб група підозрілих користувачів була максимально ізольована та не могла завадити використовувати програмний продукт іншим користувачам.

Схематична робота описаного алгоритму представлена на рисунку 3. В даному випадку користувача 3 ізольовано у контейнері С для того, щоб користувачі 1 та 2 могли продовжувати використовувати сервіс без перешкод.

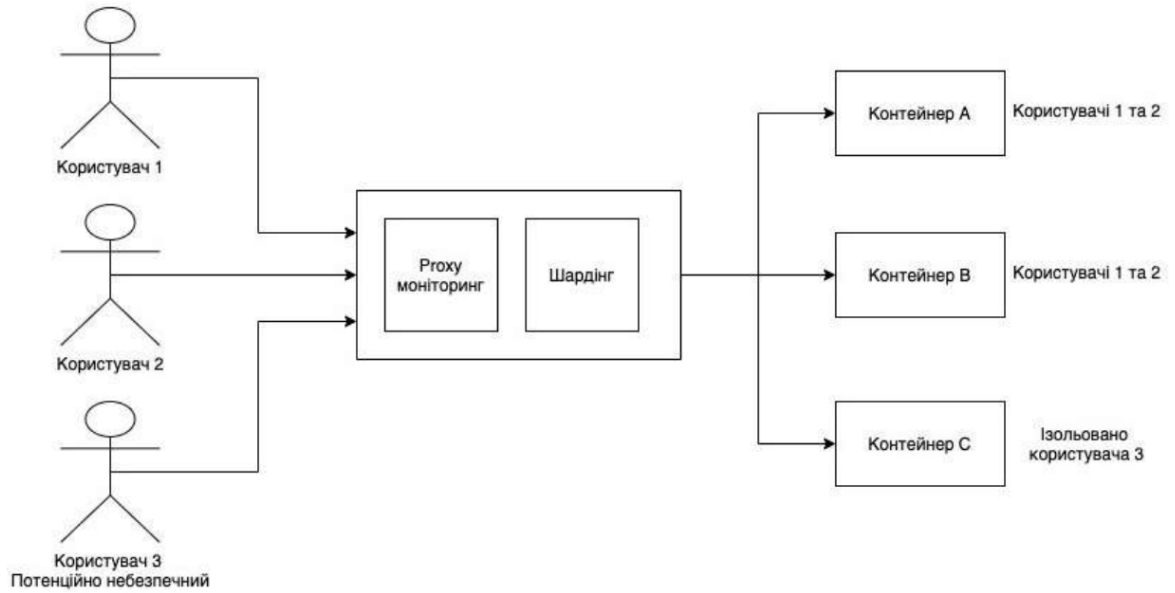


Рис. 3. Розподіл користувачів та сервісів

Висновки

Таким чином, у статті охарактеризовані основні поняття та проблеми відмовостійкості програмних систем, досліджено методи та алгоритми забезпечення відмовостійкості ПС. Зокрема, описані існуючі методи відмовостійкості ПС, на основі аналізу яких виявлені наявні проблеми у галузі відмовостійкості ПС; описані шляхи вирішення наявних проблем за допомогою реалізації нового алгоритму відмовостійкості, який дозволяє зменшити потенційні відмови ПС та проактивно реагувати на проблеми.

Література

1. Firesmith D. System resilience what exactly is it. Software Engineering Institute (SEI) at Carnegie Mellon University. URL: https://insights.sei.cmu.edu/blog/system-resilience-what-exactly-is-it/

2. Warren Axelrod C. Investing in Software resiliency. ResearchGate. URL: https://www.researchgate.net/publication/293515438_Investing_in_software_resiliency

3. Frese M. Error management or error prevention. Two strategies to deal with errors in software design. ResearchGate. URL: https://www.researchgate.net/publication/-30811276_Error_management_or_error_prevention_Two_strategies_to_deal_with_errors_in_software_design

4. Circuit breaker pattern. MICROSOFT.COM. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

5. Retry pattern. MICROSOFT.COM. URL: https://docs.microsoft.com/en-us/azure/–architecture/patterns/retry

6. Compensation transaction pattern. MICROSOFT.COM. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

7. Choudhury S. How data sharding works in a distributed sql database. YugabyteDB. URL: https://blog.yugabyte.com/how-data-sharding-works-in-a-distributed-sql-database/

ЦФЕНД ЦЕНТР ФІНАНСОВО-ЕКОНОМІЧНИХ НАУКОВИХ ДОСЛІДЖЕНЬ

МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ
INTERNATIONAL SCIENTIFIC-PRACTICAL CONFERENCE

ТЕНДЕНЦІЇ РОЗВИТКУ НАУКИ ТА ОСВІТИ: ВИКЛИКИ СУЧАСНОГО ІНФОРМАЦІЙНОГО СУСПІЛЬСТВА

TRENDS IN THE DEVELOPMENT OF SCIENCE AND EDUCATION: CHALLENGES OF THE MODERN INFORMATION SOCIETY

Збірник тез доповідей
Book of abstracts

30 вересня 2021 р.
September 30, 2021

м. Полтава, Україна
Poltava, Ukraine

ЗМІСТ
CONTENTS

Тобто, при визначенні на скільки одна скалярна величина більша за іншу в одному і тому ж одновимірному просторі, ми використовуємо єдину міру одновимірного простору, в якому існують дві порівнювані скалярні величини. Розрахунковим параметром є відстань між двома точками відповідної числової вісі, або число, визначене як різниця X і Y. Спроба введення іншого запобіжного заходу для характеристики ступеня нерівності двох одновимірних об'єктів – скалярних величин X і Y – призводить до неадекватного способу порівняння об'єктів в двовимірному просторі. При обчисленні параметрів двох скалярних величин ми визначаємо тригонометричну функцію деякого кута α, який характеризує ступінь нерівності двох точок числової вісі на площині. Отже, не коректно оцінювати ступінь нерівності двох величин, що відображають довжину, за допомогою інтегральної величини - площі. З практики відомо,що використання ефекту G-гіперболізму усуває ряд проблем при розрахунку, як базисних, так і ланцюгових економічних показників, які широко використовують в економіці.

Слід зазначити, що сьогодні відкривається новий шлях вдосконалення оціночного процесу з розглядом показників не в двовимірному, а тривимірному просторі, якому підпорядковуються майже всі системи планети. Тут визначальну роль зіграє третя вісь, фізичний зміст, якої становить певний інтерес і дає можливість для оцінки показників використати відповідний математичний метод тривимірного простору.

УДК 004.052.3

Радельчук Г. І.
к. т. н., доцент,
доцент кафедри інженерії програмного забезпечення,
Хмельницький національний університет,
Антіч Д. Ю.
магістрант,
Хмельницький національний університет

ДЕЯКІ МЕТОДОЛОГІЧНІ ПІДХОДИ ДО УДОСКОНАЛЕННЯ ВІДМОВОСТІЙКОСТІ ПРОГРАМНИХ СИСТЕМ

Забезпечення відмовостійкості програмних систем (ПС) – це сучасний тренд у розробці програмного забезпечення (ПЗ), що має на меті реалізацію надійних та стабільних систем. Відмовостійкість ПС показує, наскільки добре ця система може підтримувати безперервну роботу своїх критичних сервісів за наявності руйнівних подій (таких як відмова обладнання, кібератаки тощо).

Загалом система вважається відмовостійкою, якщо вона може здійснювати свої основні функції, незважаючи на несприятливі події. «Бути відмовостійкою» – це важлива характеристика, оскільки не має значення, як добре ПС розроблена та які технології використані для її реалізації; рано чи пізно ПС може зіштовхнутись з подіями, які можуть призвести навіть до колапсу.

До прикладу, залишкові проблеми у програмному або апаратному забезпеченні, що не були виявлені під час тестування, можуть викликати часткову або повну деградацію системи або ж спричинити невідповідність системи вимогам якості (тобто до відсутності доступності, ємності, сумісності, продуктивності, надійності, безпеки та зручності використання). Невідома або невиправлена вразливість може призвести до хакерської атаки. Сторонні проблеми (втрата електроенергії, перегрів серверів тощо) можуть призвести до повної недоступності сервісу [1, 2].

Згідно із групою досліджень, жодна система не є відмовостійкою на 100%. Тому можна сформувати визначення, що система є відмовостійкою до того моменту, доки вона здатна швидко та ефективно захищати її критичні функції від порушень, спричинених несприятливими подіями та умовами [3, 4].

Згідно з дослідженням [5] факторами, що шкодять відмовостійкості, є: розмір та складність програмної системи; взаємозалежність та взаємозв'язок; мережева орієнтованість; стрімка глобалізація компанії; ПЗ з відкритим кодом; гібридизація; швидкі зміни; повторне використання в різних контекстах.

Комплексний підхід до забезпечення відмовостійкості ПС включає в себе чотири базові методи реалізації відмовостійкості, а саме [6]:

– реакція на помилки (як тільки виникає помилка, ПС автоматично або розробники програми у ручному режимі мають відреагувати на помилки);

– автоматичне логування (ПС має автоматично логувати всі проблеми та помилки для можливості автоматичного реагування на проблеми);

– аналіз та покращення значень метрик MTBF, MTTF, MTTR (ці метрики визначають середній час між відмовами, середній час до відмови, середній час на виправлення відмови);

– автоматизація плану відновлення (необхідно автоматизувати кроки, які система здійснить для відновлення нормального функціонування).

Існують різні метрики для визначення відмовостійкості ПС [7]. Але, як правило, науковці використовують ці метрики для дослідження повних відмов системи. Загалом, існуючі методи та алгоритми відмовостійкості на сьогоднішній день фокусуються на тому, щоб зменшити негативний вплив відмов на ПС після того, як ПС почала

працювати некоректно. Однак, методи та підходи, що використовуються у сучасному ПЗ, не реагують на часткові відмови, не здатні автоматично визначати причину часткової відмови та здійснювати автоматичні спроби уснути таку відмову або основну причину відмови.

Враховуючи зазначене вище, методологічні підходи до удосконалення відмовостійкості ПС варто сконцентрувати саме на алгоритмах усунення часткових відмов ПС, зокрема на їх прогнозуванні, автоматичному визначенні їх причин та відповідних реакціях системи.

На практиці трапляються випадки, коли критичні частини ПС все ще залишаються працездатними у той час, коли низькопріоритетні частини продукту можуть частково не працювати або не працювати взагалі. Для прикладу, в Інтернет-магазині може не працювати сервіс, що відповідає за автоматичну відправку рекламних СМС-повідомлень, в той час, як авторизація, кошик, пошук та інші важливі сервіси працюють безперебійно.

Також часто трапляється, коли частина системи не працює через збій у зовнішній бібліотеці. Наприклад, сервіс авторизації має два способи авторизації: за допомогою Gmail та за допомогою Facebook. У випадку проблем із бібліотекою Facebook сервіс авторизації буде частково непрацездатним.

Саме тому є потреба проаналізувати і розробити методи та підходи визначення часткової деградації сервісів та, за потреби, переадресовувати користувачів на робочий сервіс. Наприклад, вивести повідомлення «Наразі є проблеми з авторизацією за допомогою Facebook, будь ласка, авторизуйтесь за допомогою Gmail або спробуйте пізніше» чи тимчасово відключити можливість навіть побачити кнопку авторизації за допомогою Facebook.

Для вирішення описаної проблеми можна застосувати підхід ізоляції та відключення непрацездатних частин системи; таким чином негативний вплив наявної відмови буде нижчим, оскільки користувачі не зможуть продовжувати використовувати непрацюючий функціонал, що зменшить використання критично важливих ресурсів системи (пам’ять та процесор).

Окремо слід зазначити, що важливим етапом у забезпеченні відмово-стійкості ПС у першу чергу є правильне проектування архітектури системи. Сучасні дослідження стверджують, що мікросервісна архітектура дає можливість проектувати системи, які здатні реагувати на часткові відмови [8], оскільки при використанні монолітного архітектурного підходу відмови спричиняють повну відмову системи. Завдяки використанню мікросервісного підходу забезпечується ізольованість кожного сервісу, і відмова одного із сервісів не обмежує використання інших сервісів ПС. Однак, мікросервісна архітектура, як і будь-яка інша, не гарантує 100%-ву надійність та відмовостійкість, тому

також необхідно забезпечити відмовостійкість шляхом програмної реалізації реагування на відмови, тобто використати так звані патерни чи алгоритми відмовостійкості. Оскільки в будь-який момент часу може трапитись відмова у будь-якому сервісі, необхідно, щоб інші сервіси автоматично реагували на проблему. Тому наступним етапом є розробка стратегії реагування на відмови та використання патернів відмовостійкості.

Окрім того, помилки слід розділяти на очікувані та неочікувані. Наприклад, у випадку введення невірного паролю виведення помилки є правильною реакцією системи, тому відключення функціоналу системи є невиправданим; в той же час отримання помилки на введення правильного логіну та паролю є неочікуваною поведінкою системи. До того ж, варто враховувати, як часто трапляються помилки та скільки користувачів постраждали. Наприклад, користувач А завантажує на сервер файл розміром 100 МБ, і ресурси системи успішно виконують запит; в цей же час користувачі Б та С завантажують файли розміром 100 ГБ та 115 ГБ відповідно та отримують помилки, оскільки сервіси системи не розраховані на таке навантаження. Відключення функціоналу у випадку, коли помилки отримують лише деякі користувачі, призведе до того, що решта користувачів не зможуть використовувати систему. Оптимальним вирішенням описаної проблеми є ізоляція тих користувачів, що отримують помилки, в так званий «карантин», щоб вони не завадили використовувати продукт іншим користувачам.

Розглянемо, як буде працювати даний підхід на прикладі модульного середовища Moodle для студентів та викладачів. Припустимо, що кілька груп студентів мають складати іспит о 8:00. Кожен із них авторизується на сайті, починаючи із 7:50 до 8:05 включно. У такому випадку система дозволить пройти авторизацію першій сотні студентам, решта отримають помилку і змушені будуть чекати, доки система не оновить період. В той же час, якщо відбуватиметься хакерська атака у деякий момент часу, система ніяким чином не буде блокувати запити, які надходять від злочинців, та ніяк не буде ізолювати їх від реальних користувачів системи.

Таким чином, запропонована модель забезпечення максимальної відмовостійкості ПС із покриттям часткових відмов має складатися, як мінімум, з двох частин:

– визначення та ізоляція сервісів ПС, які дали «збій»;

– визначення та ізоляція користувачів, які отримують помилки, та потенційно небезпечних користувачів.

Подальші дослідження у цьому напрямку полягають у розробці та програмній реалізації відповідних алгоритмів забезпечення відмовостійкості програмних систем.

Список літератури

1. Firesmith D. System resilience what exactly is it. *Software Engineering Institute (SEI) at Carnegie Mellon University*: [site]. URL: https://insights.sei.cmu.edu/blog/system-resilience-what-exactly-is-it/
2. What is Fault Tolerance? *Fortinet*: [site]. URL: https://www.fortinet.com/resources/cyberglossary/fault-tolerance
3. IBM High availability versus fault tolerance. *IBM*: [site]. URL: https://www.ibm.com/docs/en/powerha-aix/7.2?topic=aix-high-availability-versus-fault-tolerance
4. Stratus Fault-Tolerant Systems. *Stratus*: [site]. URL: https://www.stratus.com/fault-tolerant/
5. Warren Axelrod C. Investing in Software resiliency. *ResearchGate*: [site]. URL : https://www.researchgate.net/publication/293515438_Investing_in_software_resiliency
6. Johnson J. Resilience Engineering: An Introduction. *BMC*: [site]. URL: https://www.bmc.com/blogs/resilience-engineering/
7. Measurement Frameworks and Metrics for Resilient Networks and Services: Technical report. ENISA, 2011. 109 p.
8. Stoicescu M. Architecting Resilient Computing Systems: a Component-Based Approach for Adaptive Fault Tolerance. *ResearchGate*: [site]. URL: https://www.researchgate.net/publication/312144535_Architecting_Resilient_Computing_Systems_A_Component-Based_Approach_for_Adaptive_Fault_Tolerance

УДК 621.311

Павлюс С. Г.
к.т.н., доцент кафедри енергетики,
ДВНЗ «Український держаний хіміко-технологічний університет»,
Шкрабець Ф. П.
д.т.н., проф. кафедри енергетики,
ДВНЗ «Український держаний хіміко-технологічний університет»,
Красовський П. Ю.
к.т.н., доцент кафедри енергетики,
ДВНЗ «Український держаний хіміко-технологічний університет»

ЯКІСТЬ ЕЛЕКТРИЧНОЇ ЕНЕРГІЇ НА НЕЛІНІЙНИХ СИСТЕМАХ СПОЖИВАННЯ

Методи розрахунку реактивної потужності при низькій якості електроенергії привертають увагу багатьох вчених. Постійно змінюється

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Кафедра інженерії програмного забезпечення

Удосконалення методу та засобів забезпечення відмовостійкості програмних систем

Виконав:
студент II курсу, група ІПЗм-20-1,
Антіч Дмитро

Керівник:
доцент, кандидат технічних наук,
Медзатий Дмитро Миколайович

Об'єкт, предмет і мета дослідження

Об'єкт дослідження – відмовостійкість програмних систем.

Предмет дослідження – методи, алгоритми та засоби забезпечення відмовостійкості ПС.

Мета дослідження – удосконалення методу відмовостійкості ПС та розробка програмного засобу на його основі, що дасть змогу обробляти та зменшувати негативний вплив відмов ПС (у тому числі часткових).

Актуальність теми

Актуальність роботи полягає у тому, що у будь-якій програмній системі (ПС) хоча би раз траплялись відмови, тому необхідно використовувати методи та алгоритми забезпечення відмовостійкості ПС.

Необхідність удосконалення методу та алгоритмів відмовостійкості ПС і розробки на їх основі відповідного програмного засобу ґрунтується на наявності невирішених питань у галузі відмовостійкості ПС.

Завдання дослідження

Відповідно до мети необхідно вирішити наступні задачі дослідження:

- проаналізувати (у загальному) сферу відмовостійкості ПС;
- дослідити сучасні підходи, концепції та методи забезпечення відмовостійкості ПС з метою виявлення наявних проблем та невирішених питань;
- удосконалити наявні методи забезпечення відмовостійкості ПС для вирішення (або часткового вирішення) виявлених проблем;
- на основі удосконаленого методу відмовостійкості ПС виконати проектування відповідного програмного засобу;
- виконати програмну реалізацію удосконаленого методу відмовостійкості ПС;
- провести тестування та практичну апробацію отриманих результатів;
- дослідити ефективність запропонованих рішень;
- проаналізувати отримані результати та сформувати рекомендації щодо доцільності впровадження результатів дослідження.

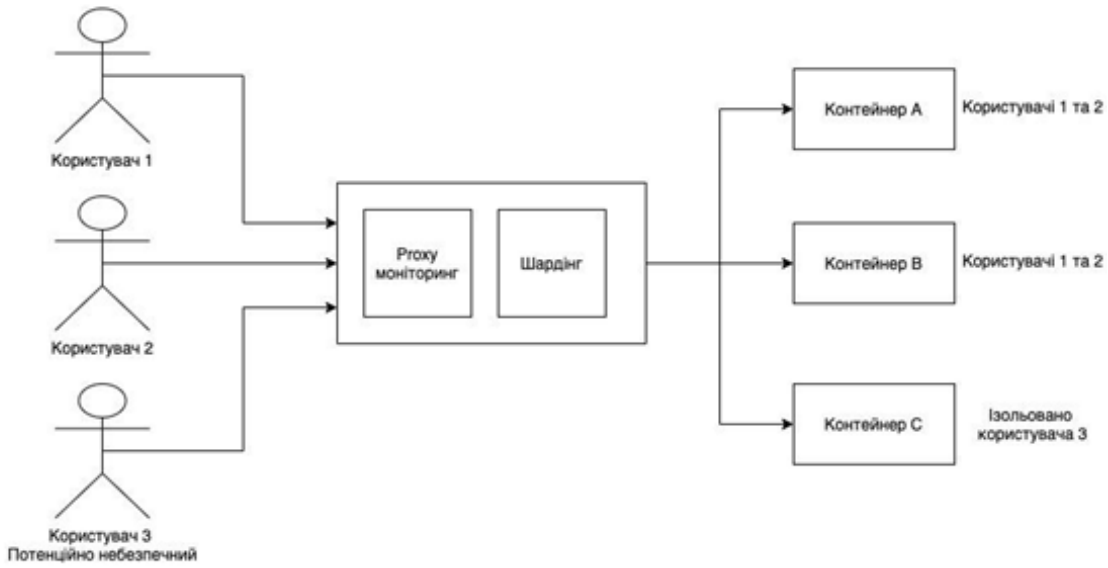
Аналіз галузі та виділення проблем

Основні проблеми у галузі відмовостійкості ПС:

- відсутність врахування та обробки часткових відмов ПС;
- неспроможність зменшити кількість помилок системи у реальному часі;
- відсутність спроможності визначати першопричину помилки.

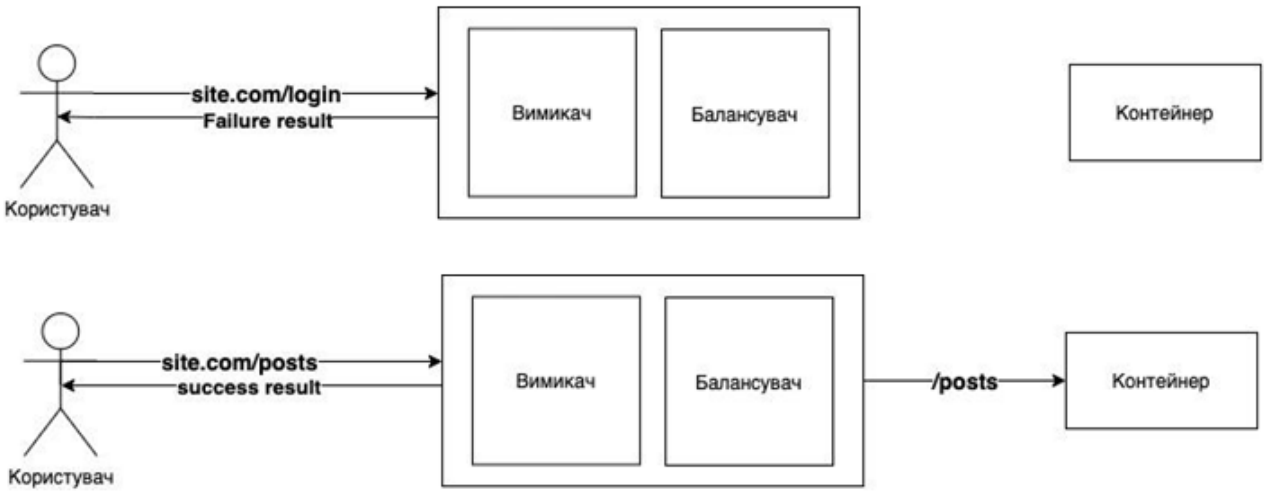
Розроблені методи

Запропоновано використання методу Шардінг з можливістю ізоляції користувачів ПС, які виконують певні дії, що призводять до помилок.

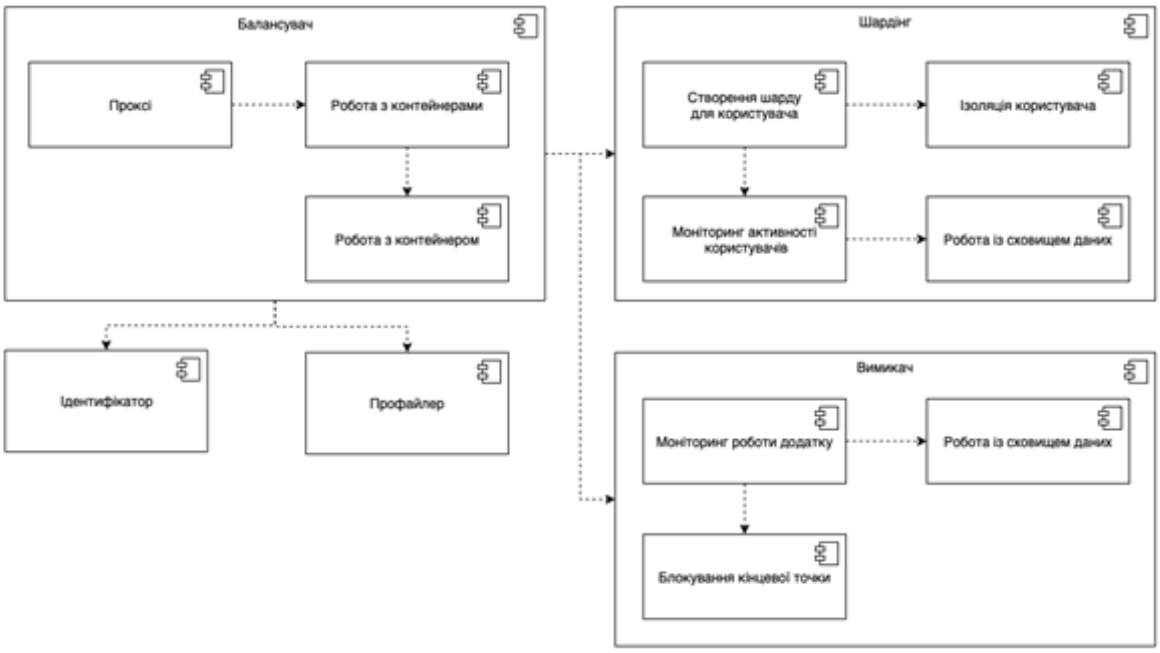


Розроблені методи

Удосконалено метод відмовостійкості ПС за рахунок комбінації патернів Вимикач та Балансувач.

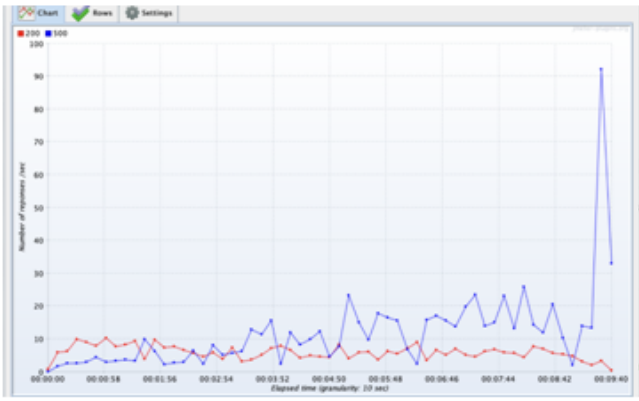


Загальна структура програмного засобу

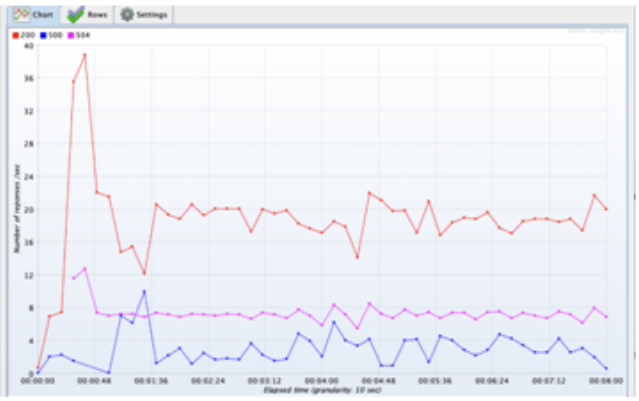


Апробація результатів

Суттєвою є різниця у графіках повернення кодів результату запиту. Так, при застосуванні удосконаленого методу відмовостійкості ПС зберігає працездатність навіть при часткових відмовах.



Графік результатів запитів користувачів під час навантажувального тестування без використання удосконаленого методу

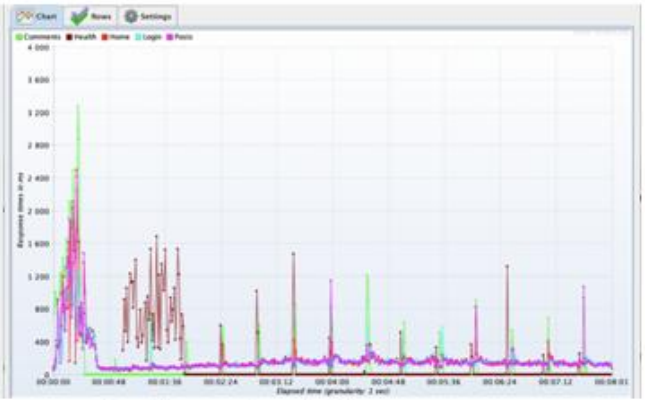


Графік результатів запитів користувачів під час навантажувального тестування з використанням удосконаленого методу.

Апробація результатів

Використання удосконаленого методу відмовостійкості ПС нормалізує час відповіді на запит, що значно підвищує пропускну здатність додатку.

Графік часу відповіді кінцевих точок при навантажувальному тестуванні без використання удосконаленого методу



Графік часу відповіді кінцевих точок при навантажувальному тестуванні з використанням удосконаленого методу

Отримані результати

Розроблений програмний засіб:

- сприяє зменшенню помилок у системі на 50% у порівнянні з класичними алгоритмами відмовостійкості ПС;
- збільшує на 50% пропускну здатність програмної системи у порівнянні з класичними алгоритмами відмовостійкості ПС.

Наукова новизна

– отримав подальший розвиток метод Шардінг у напрямку його застосування для розподілених мережних застосунків, що дозволило підвищити відмовостійкість ПС завдяки лімітації негативного впливу потенційно небезпечних користувачів лише до контейнерів їх шарду;

– розроблено метод визначення та ізоляції підозрілих користувачів, що дало змогу мінімізувати їх негативний вплив на ПС та, відповідно, зменшити кількість помилок у системі;

– удосконалено метод забезпечення відмовостійкості ПС за рахунок комбінації алгоритмів Балансувач, Вимикач та Шардінг, що дало змогу отримати максимум переваг кожного з патернів та забезпечити максимальне покриття як повних, так і часткових відмов ПС.

Практичне значення

Практична цінність отриманих результатів полягає в успішній розробці методу та програмного засобу забезпечення відмовостійкості ПС. Завдяки поліпшеним характеристикам, у порівнянні з класичними рішеннями, розроблений програмний засіб має високі конкурентні шанси на ринку.

Наукові публікації

1. Опублікована одна стаття у збірнику матеріалів Міжнародної науково-практичної Інтернет-конференції:

Радельчук Г. І., Антіч Д. Ю. Деякі методологічні підходи до удосконалення відмовостійкості програмних систем // Тенденції розвитку науки та освіти: виклики сучасного інформаційного суспільства: збірник тез доповідей міжнародної науково-практичної конференції (Полтава, 30 вересня 2021 р.). Полтава: ЦФЕНД, 2021. – С. 49-53.

2. Опублікована одна стаття у фаховому науковому виданні «Вісник Хмельницького національного університету. Серія «Технічні науки»:

Д. Ю. Антіч, Г. І. Радельчук Удосконалення алгоритмів забезпечення відмовостійкості програмних систем // Вісник Хмельницького національного університету. – 2021. – №4(299). – С. 54-58.

Висновки та рекомендації

У результаті виконання дипломної роботи здійснено системний аналіз предметної області у сфері забезпечення відмовостійкості ПС та визначені проблемні частини і невирішені питання. На основі проведеного аналізу удосконалено алгоритми та метод відмовостійкості ПС, спроектовано і реалізовано відповідний програмний засіб.

Результати практичної апробації удосконаленого методу підтверджують його працездатність та ефективність у порівнянні з наявними рішеннями у галузі, тому його можна рекомендувати використовувати в IT-компаніях, які зацікавлені у покращенні відмовостійкості власних програмних продуктів.

Дякую за увагу!

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Антіча Д. Ю.
факультет ІТ, 2 курс, група ІПЗм-20-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

23.11.2021 р.
дата

Thu Nov 25 13:32:59 EET 2021, Хіврич Володимир Русланович, Хмельницький національний університет,

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 6.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 11%**

ID: 97230 Назва: Удосконалення методу та засобів забезпечення відмовостійкості програмних систем Додано в БД: 2021-11-25 Автора: Д. Ю. Антіч Керівники: Д. М. Медзатий Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	117850	941	9588 (8%)	99 (11%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1009361839

Дата перевірки:
26.11.2021 09:41:10 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
26.11.2021 09:58:28 EET

ID користувача:
100005589

Назва документа: **ДР Антіч без додатків**

Кількість сторінок: 90 Кількість слів: 16957 Кількість символів: 137778 Розмір файлу: 5.02 MB ID файлу: 1009383373

3.85%
Схожість

Найбільша схожість: 2.5% з джерелом з Бібліотеки (ID файлу: 1005677564)

1.24% Джерела з Інтернету | 55 ..... Сторінка 92

3.23% Джерела з Бібліотеки | 93 ..... Сторінка 93

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Удосконалення методу та засобів забезпечення відмовостійкості програмних систем»
Автор: Антіч Дмитро Юрійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Медзатий Дмитро Миколайович, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титульний аркуш, бланк завдання), у структурі змісту, переліку скорочень, назвах деяких розділів/підрозділів та у назвах деяких публікацій у переліку джерел посилання;

2) усі запозичення не описують безпосередньо авторське дослідження і не стосуються результатів роботи;

3) усі запозичення є фрагментарними або мають належним чином оформлені посилання.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає **3,85%** і адресується до 55 джерел з Інтернет та 93 джерел з бібліотеки, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ Д. М. Медзатий

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ
освітнього ступеня «магістр»

Магістр Антіч Дмитро Юрійович

Тема Удосконалення методу та засобів забезпечення відмовостійкості програмних систем

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг дипломної роботи:

Кількість сторінок дипломної роботи 128 стор.

1. Короткий зміст роботи та прийнятих рішень У дипломній роботі здійснено системний аналіз предметної області у сфері забезпечення відмовостійкості програмних систем (ПС) та визначені проблемні частини і невирішені питання. На основі проведеного аналізу удосконалено та програмно реалізовано метод забезпечення відмовостійкості ПС. Розроблений метод покращує відмовостійкість додатків, побудованих за мікросервісною або кліент-серверною архітектурою, шляхом реалізації нових підходів, переосмислення та комбінації існуючих алгоритмів відмовостійкості ПС, що дало змогу удосконалити працездатність та стабільність систем, збільшити їх пропускну здатність та зменшити кількість помилок.

2. Висновок про відповідність роботи дипломному завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обгрунтовується актуальність теми роботи, формулюються мета та завдання дослідження, описується наукова новизна та практична цінність отриманих результатів. У першому розділі охарактеризовано структуру предметної області та існуючих методів і засобів забезпечення відмовостійкості ПС, визначені метологічні підходи до вирішення задачі, виконана розгорнута постановка задачі. У другому розділі досліджено методи і способи вирішення поставлених задач. Традиційні підходи до відмовостійкості ПС, досліджені на етапі аналізу, були вдосконалені шляхом комбінації традиційних підходів, їх переосмислення та впровадження нових. У третьому розділі обгрунтовано проектні рішення, що дають змогу реалізувати технічні вимоги, забезпечити сумісність та взаємодію різних компонентів системи. У четвертому розділі розглянуто питання, що стосуються реалізації програмного засобу на основі прийнятих проектних рішень, а також її технічні та технологічні характеристики. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленого програмного засобу та його функціональної придатності. Обгрунтована ефективність удосконаленого методу забезпечення відмовостійкості ПС та розроблено рекомендації з його застосування.

4. Позитивні сторони роботи Дипломна робота містить низку інноваційних рішень, зокрема, було доведено доцільність удосконалення методу та засобів забезпечення відмовостійкості ПС. Запропоновано використовувати підходи шардінгу в розподілених додатках, вперше здійснено методику визначення підозрілих користувачів та їх ізоляції. Удосконалений алгоритм показав свою ефективність під час апробації, а саме: використання алгоритму зменшує помилки на 50% та збільшує пропускну здатність системи на 50% у порівнянні із класичними алгоритмами відмовостійкості ПС.

5. Негативні сторони роботи У роботі використано відразу декілька алгоритмів та засобів забезпечення відмовостійкості ПС для вдосконалення розроблюваного методу, через що виникає питання: можливо краще було б вибрати менше алгоритмів, проте зробити це детальніше та з різних аспектів, ніж розфокусовувати дослідження до розв'язання більш широкого спектру проблем.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

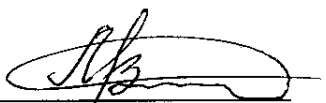
7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8.Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Мартинюк Валерій Володимирович, д-р техн. наук, професор, завідувач кафедри автоматизації та комп'ютерно-інтегрованих технологій ХНУ

30 листопада 2021 р.



(підпис)