Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота

Освітній ступінь: магістр

на тему: **«РОЗРОБКА БІБЛІОТЕКИ ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ»**

Виконав: студент 2-го року навчання,
Спеціальності
121 Інженерія програмного забезпечення

Папроцький Ігор Андрійович

Керівник Глибовець А.М.,
доктор технічних наук, доцент

Рецензент _____
(прізвище та ініціали)

Магістерська робота захищена з оцінкою _____

Секретар ЕК _____

«____» _____ 2023 р.

Київ – 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мережних технологій,

доктор. фіз.-мат. наук,

_____Г. І. Малашонок

(підпис)

“____”_____202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія програмного забезпечення
Папроцькому Ігорю
Розробити підхід до покращення відмовостійкості в мікросервісній архітектурі

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
Розділ 1. Аналіз предметної області
Розділ 2. Дослідження сучасних підходів забезпечення відмовостійкості у мікросервісній архітектурі
Розділ 3. Опис практичної частини дослідження
Висновки
Список джерел
Додатки

Дата видачі “___” _____ 202_ р.
Керівник _____ (підпис)

Завдання отримано _____ (підпис)

Тема: Розробка бібліотеки підвищення відмовостійкості в мікросервісній архітектурі

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проєкту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	01.11.2022	
2.	Огляд технічної літератури за темою роботи.	15.11.2022	
3.	Виконати аналіз сучасних методів забезпечення відмовостійкості в мікросервісній архітектурі.	29.11.2022	
4.	Розробка власного підходу до забезпечення відмовостійкості у мікросервісній архітектурі з використанням бібліотеки відмовостійкості	28.12.2022	
5.	Програмування розробленої моделі.	15.02.2023	
6.	Написання експериментів та застосування розробленої реалізації для знаходження коефіцієнтів та параметрів моделі. Написання експериментів із бібліотечною моделлю.	15.03.2023	
7.	Виконання порівняльного аналізу результатів експериментів, отриманих за допомогою розробленого алгоритму.	01.04.2023	
8.	Написання пояснювальної роботи.	29.04.2023	
9.	Створення слайдів для доповіді та написання доповіді.	14.05.2023	
10.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи.	20.05.2023	
11.	Корегування роботи за результатами попереднього захисту.	25.05.2023	
12.	Остаточне оформлення пояснювальної роботи та слайдів	30.05.2023	
13.	Захист магістерської роботи (проекту)	14.06.2023	

Студент _____Папроцький І.А._____

Керівник _____Глибовець А.М._____

“_____” _____ _____ р.

Зміст

Анотація

Дана робота присвячена дослідженню сучасних практик забезпечення відмовостійкості мікросервісів. Дослідження розглядає поширені патерни для реалізації цієї задачі. Основну увагу приділено відомому патерну під назвою Circuit Breaker, а саме спробі покращення його стандартної імплементації з точки зору надійності та продуктивності. Для досягнення цієї мети запропоновано підхід, який полягає у зменшенні кількості часових затримок, які використовуються при переході між його станами, а також зменшенні кількості станів самого інструменту Circuit Breaker за рахунок переходу до моделі прогнозування стабільності системи на основі метрик, що збираються в процесі роботи застосунку. Результатом роботи є модель Circuit Breaker яка працює у двох станах, а також порівняльні експерименти для перевірки правильності припущень покладених на ефективність цієї моделі.

Ключові слова: мікросервісна архітектура, забезпечення відмовостійкості, мікросервіс, шаблон програмування, Circuit Breaker, бібліотеки відмовостійкості, фреймворки відмовостійкості, метрики, порогове значення.

Вступ

У сучасній розробці програмного забезпечення все більше розробників переходять до мікросервісної архітектури для побудови все більш гнучких та масштабованих додатків. За даними опитування JetBrains, у 2022 році близько 86% респондентів використовували мікросервісний підхід у дизайні своїх програмних систем [1]. Причини для забезпечення гнучкості та масштабованості очевидні: при збільшенні розміру програмної системи за рахунок додавання її нових складових частин, а також при збільшенні навантаження на систему, виникає необхідність збереження її працездатності та уникнення втрат у її ефективності роботи.

За загальноприйнятими даними, мікросервісна архітектура полягає в тому, що складний додаток розбивається на набір дрібніших сервісів, які можуть бути розгорнуті, масштабовані та налаштовані незалежно один від одного, а кожен сервіс виконує свою частину функціоналу системи та має власний набір API для взаємодії з іншими сервісами. Проте, при збільшенні кількості взаємодій всередині системи внаслідок зростання її складності, виникають нові можливості для відмов її компонентів, або ж помилок та збоїв у комунікації між сервісами, а отже функціонал системи може стати тією чи іншою мірою непрацездатним. Навіть невелика відмова одного з мікросервісів може призвести до серйозного перебою в роботі всієї системи та відсутності доступу користувачів. Тому, важливо мати ефективні методи забезпечення надійності та стійкості в мікросервісних архітектурах, щоб уникнути цього.

Поняття Fault Tolerance, або ж відмовостійкості, є ключовою проблемою в цьому контексті, і воно вимагає певних підходів та методів для забезпечення надійності мікросервісів.

У даній дипломній роботі розглядаються різні шаблони та підходи до забезпечення fault folerance в мікросервісній архітектурі. Серед шаблонів,

або ж патернів відмовостійкості, виділено популярний патерн Circuit Breaker, робота якого схожа на автоматичний вимикач у реальних електричних системах. Загальновідома реалізація даного патерну включає в себе використання 3 станів, в яких помічено використання затримок при переході між станами, яких можна оминути за рахунок підходу прогнозування стану системи на основі метрик, які збирає патерн під час своєї роботи. Виходячи з важливості досліджень у сфері забезпечення відмовостійкості для сучасної галузі розробки програмного забезпечення, за мету даної роботи були поставлені дослідження та розробка власної моделі покращення згаданого патерну.

Робота складається з трьох основних розділів.

Перший розділ присвячено огляду мікросервісів та викликів які постають перед розробниками при використанні даного виду архітектури. Також розглядається концепція забезпечення відмовостійкості.

У другому розділі описано відомі підходи та патерни, які використовуються для забезпечення fault tolerance. Також перелічено сучасні бібліотеки та фреймворки, які надають реалізації описаних патернів. Детально розглянуто патерн Circuit Breaker та його загальновідомий підхід з використанням його трьох основних станів, порівняні імплементації та їх архітектура у згаданих бібліотеках.

Третій розділ містить детальний опис запропонованого методу покращення архітектури реалізації Circuit Breaker за допомогою прогнозування переходів між його станами. Представлено окрему реалізацію на основі однієї з популярних бібліотек, також описано експерименти з порівняння між стандартною бібліотечною імплементацією та запропонованою в цій роботі.

Застосування запропонованої реалізації Circuit Breaker може покращити ефективність використання цього патерну при розробці системи з мікросервісною архітектурою.

Постановка задачі:

1. Дослідити сучасні підходи із гарантування відмовостійкості під час проектування та розробки програмного забезпечення з використанням мікросервісної архітектури.
2. Провести аналіз сучасних підходів та фреймворків із забезпечення відмовостійкості мікросервісів.
3. Розробити власну реалізацію гарантування відмовостійкості на основі проведеного аналізу.
4. Провести аналіз ефективності розробленої імплементації.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мікросервіси

Мікросервіси – це архітектурний підхід, який полягає в розділенні одного монолітного застосунку на менші незалежні компоненти, кожен з яких може забезпечувати свої функції, а також бути масштабованим окремо від інших.

Оскільки кожен сервіс у цій архітектурі є незалежним від інших, він може бути розроблений на іншій мові програмування або стеку технологій в цілому.

Кожен мікросервіс будується таким чином, щоб його можна було якомога легше масштабувати, особливо у динамічному середовищі. Для розгортання та масштабування такого роду архітектури можуть використовуватись інфраструктура хмарних провайдерів, контейнеризація, системи оркестрації контейнерів, балансувальники навантаження, та багато інших інструментів.

Дані сервіси можуть взаємодіяти між собою шляхом синхронної та/або асинхронної комунікації. Це може бути досягнуто за допомогою інтерфейсів та протоколів REST, RPC, протоколів обміну повідомленнями, веб-сокетів, або інших спеціальних протоколів передачі даних. [1]

На Рисунку 1 схематично зображена стандартна мікросервісна архітектура: клієнти звертаються до певного API Gateway (компонента, який розподіляє запити), який, в свою чергу, звертається до самих мікросервісів, кожен з яких може мати своє сховище даних. Самі ж сервіси можуть взаємодіяти між собою за певними протоколами передачі даних, якщо при виконанні їх бізнес логіки необхідно звертатись до бізнес логіки іншого мікросервісу. Для простоти на рисунку не зображені технічні деталі, такі як балансувальники навантаження, регістри сервісів, черги повідомлень або види взаємодії сервісів між собою, але варто зазначити, що

такі елементи архітектури можуть також застосовуватись при розробці системи за потреби.

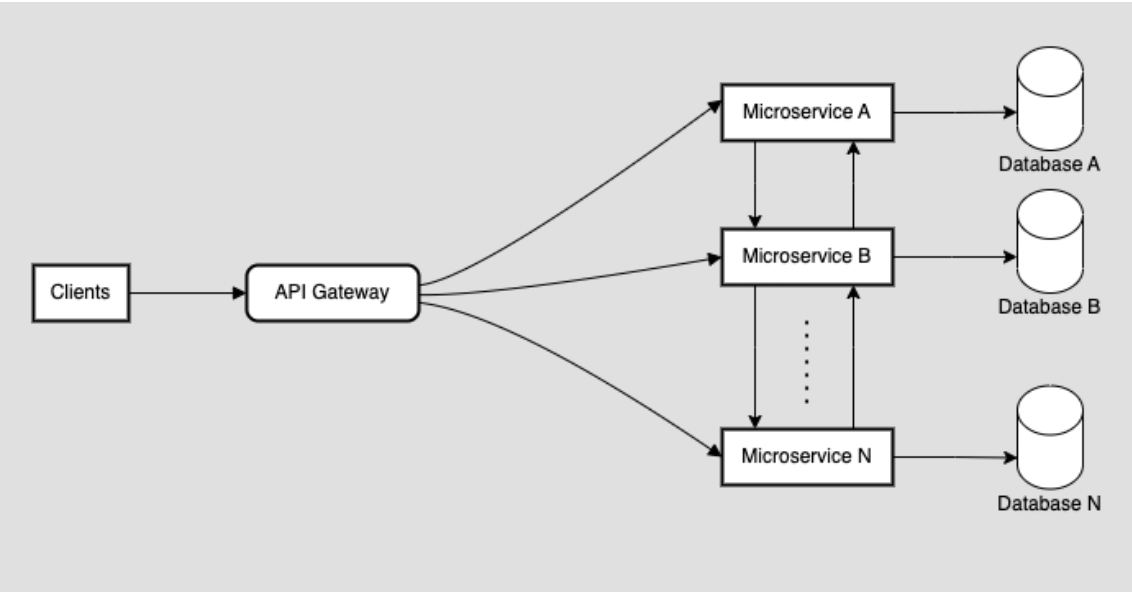

Рисунок 1. Мікросервісна архітектура

Серед основних переваг використання мікросервісів виділяють наступні [2]:

1. Масштабованість: мікросервіси дають змогу масштабувати окремі компоненти системи незалежно один від одного, що дозволяє розподіляти навантаження більш ефективно та реалізувати вимоги до продуктивності.
2. Гнучкість: мікросервіси дають змогу більш гнучко оновлювати та розвивати окремі компоненти системи, змінювати їх версії, без необхідності оновлювати весь монолітний додаток.
3. Незалежність: кожен мікросервіс може бути розгорнутий та підтримуваний незалежно від інших, що забезпечує більшу гнучкість в управлінні та розробці.
4. Легкість у впровадженні нових технологій: мікросервіси дозволяють впроваджувати нові технології та інструменти окремо для кожного компонента системи, що забезпечує більшу свободу та інноваційність у розробці.

5. Ефективність: мікросервіси дозволяють використовувати ресурси більш ефективно, оскільки окремі компоненти системи можуть бути розгорнуті на різних серверах та інфраструктурах.
6. Швидкість розробки: розробка окремих мікросервісів може відбуватись паралельно та незалежно, що забезпечує пришвидшення процесу розробки та зменшення залежностей між компонентами системи.

1.2 Виклики при використанні мікросервісної архітектури

Як вже було згадано, при виборі такого виду ускладнення в дизайні системи в цілому, як розбиття її на багато частин, викликає деяка кількість викликів до розробників, які ускладнюють процес розробки.

Серед таких складнощів можна виділити наступні [3]:

1. Деякі операції у системі, що розподілена мікросервісами, можуть бути складними для розуміння з першого погляду для усунення неполадок, що збільшує час на виявлення та виправлення помилок.
2. Деякі операції можуть потребувати виконання складного механізму узгодження транзакцій, оскільки розподілення доменів застосунку по мікросервісам вимагає використання окремих баз даних для кожного з них.
3. Оскільки кожен сервіс може бути розгорнутий на окремому сервері та мати власну базу даних, то для забезпечення повного функціонування бізнес логіки потрібно забезпечити ефективну комунікацію між ними та зовнішніми системами.
4. Складність розгортання, тестування, та налаштування архітектури в цілому: перед розробниками з’являються такі задачі як маршрутизація запитів всередині архітектури, автентифікація та безпека запитів та сесій користувачів, контроль за версійністю розгорнутих сервісів.

5. Архітектурний підхід вимагає збільшення кількості написаного коду, адже замість одного монолітного застосунку створюється багато менших.

1.3 Відмовостійкість

Поняття відмовостійкості вживається у різних сферах науки та техніки. Наприклад, серед відомих сфер використання цього поняття, можна виділити наступні [4]:

- Hardware Fault Tolerance, або ж відмовостійкість у апаратному забезпеченні. Вважається найбільш зрілою галуззю в якій досліджується та розвивається відмовостійкість.
- Information Redundancy. Досліджуються помилки які виникають при пересиланні інформації з одного програмного місця в інше, від однієї системи до іншої, або навіть коли дані зберігаються в тих чи інших сховищах даних. Наприклад, використовується різні методи кодування інформації або підходи RAID (Redundant Array of Independent Disks) для забезпечення цілісності даних у різних ситуаціях.
- Fault-Tolerant Networks. Досліджується відмовостійкість мережних технологій.
- Fault Detection in Cryptographic Systems. Досліджує відмови у пристроях та алгоритмах шифрування та дешифрування.
- Software Fault Tolerance – відмовостійкість у програмному забезпеченні. Саме цей вид відмовостійкості розглядається у деталях в даній роботі, а особливо відмовостійкість у мікросервісній комунікації.

У розподілених системах та мікросервісній архітектурі кожен сервіс може мати свій власний життєвий цикл. При взаємодії, один сервіс може не

знати, що інший сервіс у даний момент не може обробляти запити до нього. Ця ситуація може погіршуватись, якщо для виконання запиту користувача необхідно зробити ланцюжок запитів між різними сервісами всередині системи. Відповідно, відмова одного сервісу може мати каскадний ефект на інші сервіси та взаємодію з ними, що призведе до зниження доступності всієї системи.

Саме тому важливо планувати та розробляти механізми відмовостійкості, такі як обробка помилок, резервне копіювання даних, моніторинг, відновлення після відмови та інші. Відмовостійкість можна досягти за допомогою різних підходів та технологій, таких як кластеризація, оркестрація, реплікація, автоматичне масштабування та інші.

У залежності від задач та особливостей конкретної програмної системи та її вимог до доступності, можуть бути застосовані різні стратегії або ж патерни відмовостійкості. Важливо враховувати цей аспект при проектуванні у мікросервісній архітектурі, щоб забезпечити її стабільну та надійну роботу. У наступному розділі детальніше розглянуто деякі методи та патерни відмовостійкості.

2 ДОСЛІДЖЕННЯ СУЧАСНИХ ПІДХОДІВ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

2.1 Сучасні методи та патерни відмовостійкості

Як і у будь-якій сфері розробки програмного забезпечення, щодо відмовостійкості існує значна кількість практик, методів та шаблонів запобігання відмов у різних частинах програмного оточення.

Серед найвідоміших Stability Patterns, тобто патернів стабільності можна виділити наступні: Timeouts, Retries, Circuit Breaker, Bulkheads, Rate Limiters. [5]

Серед менш популярних методів можна також згадати Steady State, Fail Fast, Let if Crash, Handshaking, Test Harnesses, Decoupling Middleware, Shed Load, Governor, та інші. [5]

2.1.1 Retry Pattern

Механізм Retry полягає у повторному надсиланні запиту до іншого сервісу у разі повернення помилки або перевищення часу очікування відповіді. В контексті одиничного виконання логіки застосунку, наприклад одного зовнішнього виклику сервісу, повтори можуть виконуватись у заданій конфігурацією застосунку кількості, тобто при повторенні помилок запити будуть надсилатись повторно стільки разів, скільки необхідно.

Поняття «retry», або ж повторів, є загальновідомим у контексті комунікації між сервісами. Застосування такого шаблону поведінки застосунку дозволяє уникати наслідків випадкових короткочасних перебоїв в мережі, зайнятості ресурсів цільового сервісу, та інших короткострокових причин, які можуть заважати виконанню логіки застосунку в одиничних ситуаціях.

Класичний механізм повтору запитів може здаватись простою задачею при роботі в застосунку з невеликим навантаженням. Проте, при збільшенні

кількості потоків, які конкурують між собою, а також при збільшенні кількості клієнтів або операцій, які викликають таку логіку, виникає загроза відмови цільового сервісу, оскільки він потенційно не може виконати таку кількість запитів за певний проміжок часу.

Для вирішення таких проблем використовуються, наприклад, такі алгоритми як *exponential backoff* та *jitter* [6]. Exponential backoff збільшує час очікування експоненційно після кожної спроби запиту, що дозволяє розвантажити цільовий сервіс при великій кількості клієнтів, що одночасно звертаються до нього. Jitter, в свою чергу, додає деяку випадкову величину до часу між запитами, що в свою чергу, розріджує запити від багатьох користувачів, які потенційно можуть звернутись до цільового сервісу одночасно. [6]

Варто також пам’ятати про те, що у разі довгострокової нездатності цільового сервісу відповідати на запит, наприклад, внаслідок відмови, механізм Retry може призводити до певної надлишковості, оскільки буде робити повторні спроби запитів, які, очевидно, будуть неуспішними, що може затримувати виконання поточної логіки застосунку на деякий час. Для цього необхідно обмежувати кількість повторів певним виправданим значенням.

2.1.2 Timeouts

Timeouts, або таймаути, є критичною частиною синхронної взаємодії між сервісами. Їх суть полягає у використанні деякого значення часу за який сервіс, або клієнт, що викликає цільовий сервіс, буде чекати відповіді на його виклик.

Однією з важливих задач розподілених систем являється управління доступністю (availability) їх компонентів, та можливостями надавати відповідь (responsiveness) на запити. [7] Правильно налаштовані таймаути запитів у системі дозволяють сприяти покращенню цих параметрів, та

запобігати занадто повільним відповідям сервісів, як показано на Рисунку 2:

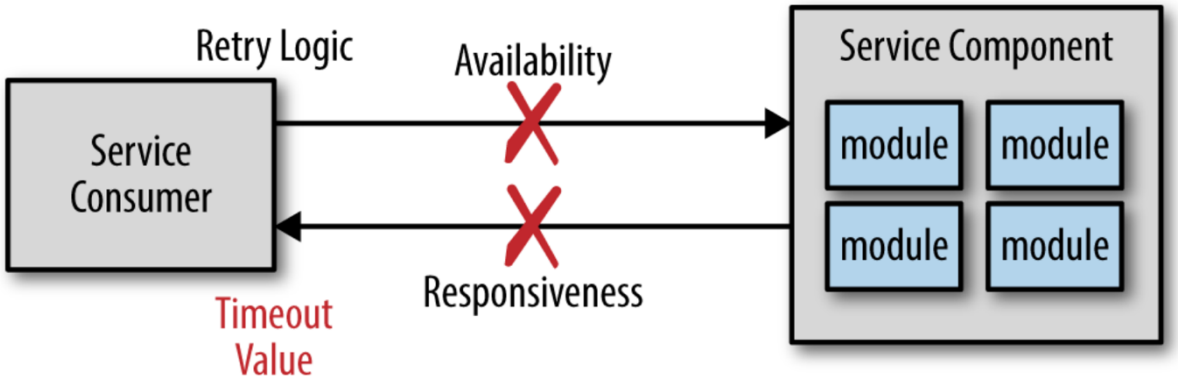

Рисунок 2. Використання Timeout при взаємодії з сервісом [7]

Клієнт (Service Consumer) робить запити на сервіс (Service Component), чим випробовує його доступність (availability) та чекає на відповідь впродовж деякого часу (Timeout Value), випробовуючи здатність на відповідь (responsiveness) сервісу, та не очікуючи на результат безкінечно. У цій схемі також згадується механізм Retry (Retry Logic) із попереднього пункту роботи, оскільки запити можуть повторюватись для спроби досягнення результату.

Проте, використання таймаутів може також створювати певну надлишковість, за рахунок того, що клієнт може очікувати занадто довго на відповідь, хоча потенційно цільовий сервіс може не мати можливість відповісти, наприклад, через відмову. Особливо ця ситуація погіршується за використання одночасно Retry та Timeout: оскільки при кожній повторній спробі запиту застосунок буде чекати певний максимально налаштований час у разі відмови іншого сервісу. Саме тому деякі джерела вважають це антипатерном [7].

2.1.3 Bulkhead

Патерн Bulkhead являється типом дизайну архітектури програмного забезпечення, в якій елементи застосунку ізолюються в набори (пули) таким

чином, що коли один з них перестає функціонувати, інші продовжують працювати. Назва даного патерну асоціюється з корабельними перегородками під ватерлінією – при пробої заповнюється водою лише один відсік, в той час як інша частина корабля продовжує функціонувати. [8]

У мікросервісній архітектурі основною ідеєю цього патерну являється розділення мікросервісів на окремі групи, які можуть працювати незалежно одна від одної. Ці групи являють собою роздільники, які відділяють один сервіс від іншого, надаючи їм власні ресурси, такі як пам’ять, процесорний час та мережеве з’єднання (наприклад, у вигляді connection pools – пулів з’єднань). Якщо один із сервісів в групі зазнає надмірного навантаження або відмовляє, то це не вплине на роботу інших сервісів розташованих у інших групах.

2.1.4 Rate Limiting Pattern

Бувають ситуації, коли один з сервісів, до якого звертаються клієнти або ж інші сервіси, при запиті до нього виконує певні складні операції які вимагають багато ресурсів (наприклад, інтенсивно використовує потоки вводу/виводу). Тому, такий запит може займати певний час до відповіді клієнту, і тому сервіс може обробити обмежену кількість таких запитів за певний часовий проміжок. Проте, якщо такий сервіс отримує велику кількість запитів від його споживачів, існує ризик відмови цього сервісу. [9]

Одним із способів подолання такої проблеми є використання патерну Rate Limiter (або ж Rate Limiting). Даний шаблон полягає в тому, що цільовий сервіс повинен відхиляти частину запитів, якщо їх кількість за певний налаштований часовий проміжок перевищує деякий поріг.

Існує деяка кількість варіантів того, як можна обробляти ті запити, які потрапляють до сервісу при перевищенні порогу встановленого ліміту, або при заданні певних типів запитів які будуть вважатися надлишковими. Такі

запити можна просто відхиляти, або побудувати певну чергу для того, щоб обробляти їх пізніше, або ж навіть комбінувати ці підходи. [10]

Отже, споживачі, запити яких були відхилені, повинні зачекати деякий час та спробувати повторити свій запит через деякий час. Жертвуючи обробкою лише частини запитів при великому навантаженні, сервіс робить спробу захистити себе від потенційно повної відмови, під час якої не будуть оброблятись повністю усі запити.

2.1.5 Circuit Breaker

Як вже було зазначено у першому розділі даної роботи, взаємодія мікросервісів між собою може приносити певні проблеми у разі відмови цільового сервісу до якого звертається сервіс-споживач. Одна з таких проблем виникає тоді, коли існує певний ланцюг викликів між сервісами: один сервіс звертається до другого, другий в процесі обробки виклику звертається до третього, і т.д.). У такому разі, відмова якогось з сервісів в кінці ланцюжка спричинить довге очікування та/або помилки в усіх взаємодіях між сервісами що знаходяться в цій послідовності запитів перед ним. Такий процес називається Cascading Failures, або ж каскадними відмовами. [11] Один зі способів боротьби з такою проблемою передбачає використання шаблону Circuit Breaker.

Головна ідея патерну Circuit Breaker є досить простою: деяка функція (наприклад, яка робить запит до цільового сервісу) обгортається об’єктом circuit breaker, який моніторить стан її виконання та слідить за помилками. У разі перевищення певного порогу помилок які відбуваються впродовж викликів цієї функції, об’єкт перемикача переходить у стан, в якому усі наступні виклики цієї функції повертатимуть помилку, при цьому сама функція виконуватись не буде. Зазвичай система певним чином автоматично сповіщається про перехід у цей стан. [5, 12]

Назва цього патерну асоціюється зі звичайним автоматичним вимикачем (або ж запобіжником) у електричних мережах будівлі, який

«відкривається» при перевищенні певних параметрів у мережі (наприклад, напруги) та розриває собою електричне коло. Зазвичай у реальній електричній мережі для відновлення роботи системи вимагається фізичне втручання людини, яка замінить запобіжник або проведе певний ремонт. Проте, в програмному забезпеченні однойменному патерну необхідно автоматично виявити умови, при яких він повинен знову «закритись» та відновити доступ до виконання функції взаємодії між сервісами.

Шаблон Circuit Breaker імітує таку поведінку за рахунок знаходження у певних станах, які відповідають за його реакцію на виклики функції взаємодії. Класична реалізація цього патерну передбачає три стани вимикача, які схематично зображені на Рисунку 3 [12]:

1. Closed (закритий) – виклики дозволяються, поки вони успішні (success, на схемі) до певного порогу (under threshold, на схемі) кількості перехоплених помилок при виконанні функції (fail).
2. Open (відкритий) – у разі перевищення порогу помилок (threshold reached), усі запити до функції будуть відхилятись із помилкою, при цьому сама функція виконуватись не буде. Через певний встановлений часовий проміжок (reset timeout) після переходу в цей стан, він змінюється на наступний.
3. Half-Open (напіввідкритий) – після таймауту вимикач ніби «призакривається» на деяку кількість дозволених запитів, щоб протестувати функцію взаємодії на предмет повторення помилок. У разі їх відсутності – стан змінюється на Closed, де всі запити знову будуть дозволятись, інакше – стан повертається до Open на оновлений таймаут.

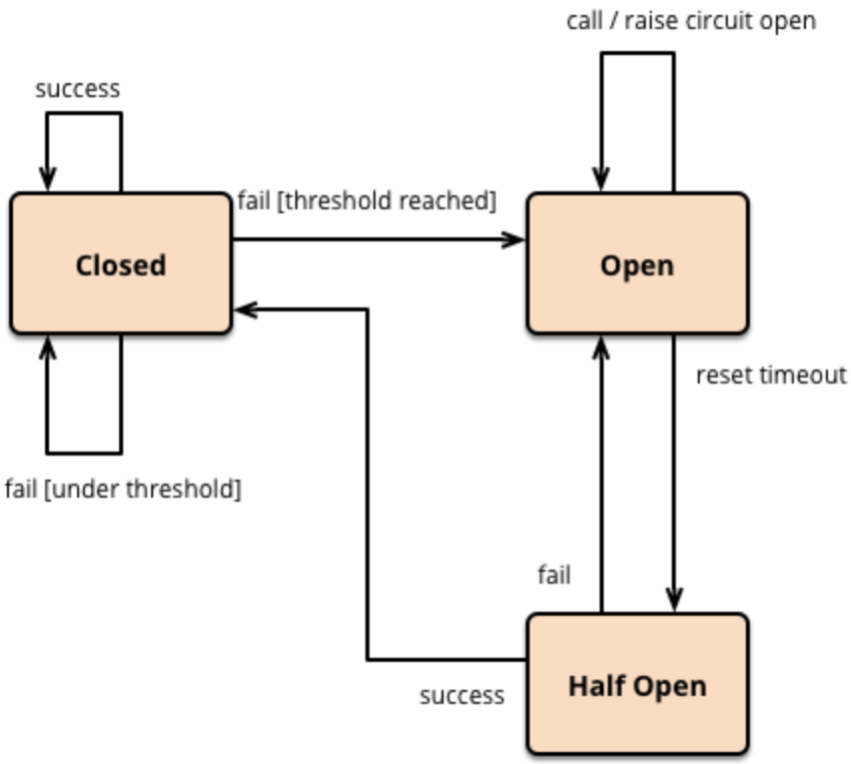

Рисунок 3. Шаблон Circuit Breaker з трьома станами [12]

Цю досить просту схему роботи патерну можна широко налаштовувати в залежності від потреб та реалізації, наприклад, задавати різні значення таймаутів при очікуванні у відкритому стані, задавати різні порогові значення помилок у закритому та напіввідкритому стані, фільтрувати частину помилок по їх типу при обрахунку перевищення порогових значень, використовувати функції зворотньої роботи (fallback) які спрацьовують у разі помилки всередині функції взаємодії, або при відхиленні запиту самим вимикачем відповідно до його поточного стану, та інші.

Цей шаблон можна також комбінувати одночасно з іншими патернами відмовостійкості, наприклад, Retry, Timeout (або ж Time Limiter), Bulkhead, Rate Limiter, тобто усіма згаданими у попередніх пунктах.

Простота та популярність Circuit Breaker (що можна побачити з наступного пункту, адже його реалізація існує у більшості популярних бібліотек відмовостійкості), стали причиною найбільшої зацікавленості

саме цим патерном у даній роботі для спроби його покращення у практичній частині дипломної роботи.

2.2 Бібліотеки та фреймворки із забезпечення відмовостійкості

В рамках цієї роботи знайдено наступні відомі фреймворки та бібліотеки, які тією чи іншою реалізовують вищенаведені підходи та патерни.

2.2.1 Netflix Hystrix

Netflix Hystrix – це потужна бібліотека, розроблена компанією Netflix, яка забезпечує надійність та стійкість у мікросервісних архітектурах. Вона надає засоби для керування відмовами та контролю взаємодії між мікросервісами.

Бібліотека була створена командою розробників Netflix API у 2011 році, і вона досі широко використовується компанією у своїх системах. Наприклад, офіційна документації вказує на щоденну статистику у розмірі від десятків до сотень мільярдів захищених викликів щодня у продуктах Netflix. [14]

Не є секретом, що компанія Netflix входить в топ світових IT-корпорацій, а її продуктами користуються користувачі зі всього світу. Це доводить статистика ваги таких компаній у відомому фондовому індексі S&P 500, одним із компонентів якого є саме Netflix. На момент написання цієї дипломної роботи, компанія посідала 39 місце за вагою у цьому індексі. [13]

Серед реалізованих Hystrix патернів та підходів можна перечислити: вже описаний патерн Circuit Breaker, функції fallback (запасні механізми при відмові мікросервісу), Bulkhead, Timeout, кешування та згортання запитів, адаптивна паралельність виконання запитів, ізоляція за допомогою семафорів та на рівні потоків, контроль навантаження. Головним способом

організації роботи з викликами у бібліотеці являється обгортання усіх викликів до сторонніх сервісів в об’єкт класів HystrixCommand або HystrixObservableCommand який зазвичай виконується у окремому потоці або пулі потоків, при цьому виконуючи задані користувачем обмеження для відмовостійкості. [14]

Тривалий час Netflix Hystrix була популярним інструментом у мікросервісному середовищі завдяки його гнучкості, легкості використання та великому набору функцій. Проте, у 2018 році підтримка бібліотеки була завершена. Компанія все ще використовує бібліотеку для своїх старіших сервісів та застосунків, проте для новіших – рекомендує використовувати такі проекти, як Resilience4j. [14]

2.2.2 Istio

Istio – це відкрите програмне забезпечення, розроблене компанією IBM, яке надає сервісну мережу (service mesh) для управління взаємодією мікросервісів у мікросервісній архітектурі. Дане рішення надає широкі можливості з спостереження та оптимізації мережею мікросервісів. [15]

Серед пропонованих можливостей можна перечислити: управління мережею, забезпечення безпеки, маршрутизація, балансування навантаження, керування версіями, моніторинг та трейсинг. Цікаво також, що Istio реалізує різні патерни відмовостійкості, такі як Circuit Breaker, механізми fallback, Timeout, Retry, Rate Limiter, Fault Injection, Canary Testing та інші. [15]

Дане програмне забезпечення розширює відому платформу для управління контейнерами Kubernetes [15], та може бути складним для впровадження рішенням. Воно вимагає глибокого розуміння мережевих принципів і налаштування, що може стати викликом для новачків або команд з обмеженими обчислювальними ресурсами. Це також значить що

Istio вимагає додаткових ресурсів, таких як проксі-сервери та контролери, що можуть збільшити вимоги до інфраструктури.

Звісно ж, ці недоліки не означають, що Istio не є цінним інструментом, але їх варто враховувати при обранні підходящого рішення у вигляді бібліотеки, фреймворку, або сервісної мережі під конкретні потреби проекту.

2.2.3 Sentinel

Sentinel - це фреймворк з відкритим кодом для управління політиками доступу і контролю в мікросервісних архітектурах. Він надає засоби для визначення, керування і застосування правил доступу, аутентифікації, авторизації і обмеження обсягу для різних компонентів системи. Фреймворк орієнтується на контроль трафіку та його обсягів, управління політиками доступу, моніторинг та логування, резервне керування системою. Почав розроблятись Sentinel у 2012 році групою компаній Alibaba Group, а в 2022 перетворився в “хмарний компонент «керування трафіком»”. Найбільшу популярність фреймворк отримав у проектах азійських країн, наприклад Alibaba, AliExpress, та інші. [16]

Серед відомих підходів та методів відмовостійкості Sentinel реалізує наступні: Circuit Breaker, Rate Limiting, механізм fallback, Adaptive Throttling, Retry, Timeout та інші. [16]

2.2.4 Модулі Spring

У відомому веб-фреймворку Spring існують модулі які підтримують зручну інтеграцію відомих бібліотек відмовостійкості. Серед таких модулів можна виділити наступні:

- Spring Cloud Netflix – модуль надає підтримку для використання бібліотеки Netflix OSS, в тому числі Netflix Hystrix для реалізації Circuit Breaker, розподілене балансування навантаження з

використанням Ribbon, а також Eureka (сервіс реєстрації мікросервісів).

- Spring Cloud Circuit Breaker – цей модуль є інструментом абстракції для інтеграції з різними бібліотеками які реалізують Circuit Breaker, такими як Resilience4j, Netflix Hystrix або Sentinel. Він надає механізм єдиної конфігурації та використання Circuit Breaker незалежно від конкретної реалізації.
- Spring Retry – модуль надає підтримку повторів виконання методів у разі виникнення помилок під час їх роботи.
- Spring Cloud Load Balancer – модуль надає підтримку балансування навантаження між екземплярами одного і того ж сервісу.

Всі наведені модулі спрямовані на надання зручних механізмів інтеграції реалізацій підходів з покращення відмовостійкості у конкретний сервіс, який пише користувач використовуючи поширену екосистему Spring. Це дозволяє розробникам легше перемикатись між реалізаціями в своїх застосунках з тих, чи інших причин, при цьому витрачати менше часу на вивчення конфігурацій окремих бібліотек, які є зазвичай зовсім різними.

2.2.5 Resilience4j

Resilience4j – потужна та гнучка бібліотека з відкритим кодом для покращення відмовостійкості застосунків, що і є її основною метою.

Вона пропонує реалізацію таких патернів як Circuit Breaker, Bulkhead, Rate Limiter, Retry, Time Limiter (Timeout), Cache. Resilience4j дозволяє легко налаштовувати ці патерни з використанням анотацій або програмно, інтегруючись із різними фреймворками та бібліотеками. Початково бібліотека була створена для мови Java, але тепер також існує підтримка мови Kotlin. Вона підтримує інтеграцію з такими інструментами та бібліотеками як Spring Boot 2&3, Micronaut, Spring Reactor, Spring Cloud, RxJava 2&3, Feign, Micrometer та Graphana. [17]

Цікаво також, що бібліотека надає легкі способи налаштовувати себе різними способами, наприклад, у поєднанні з Spring Boot, бібліотека надає широкі можливості налаштування використаних патернів за допомогою файлів розширення .property та .yml, програмним методом з допомогою патерну Builder використовуючи лише можливості мови Java, або ж за допомогою анотацій. [17]

Як уже було згадано у пункті 2.2.1, документація бібліотеки Netflix Hystrix посилається на те, що тепер замість Hystrix варто розглянути використання бібліотеки Resilience4j, а розробники будуть використовувати її у своїх нових проектах.

У порівнянні з бібліотекою Netflix Hystrix, можна відмітити наступні відмінності [17]:

- дизайн Resilience4j розрахований на функціональну парадигму програмування;
- у Hystrix запити до зовнішніх систем повинні бути обгорнуті в об’єкт класу HystrixCommand. У цій же бібліотеці можна використовувати декоратори для будь-якого функціонального інтерфейсу, лямбда-виразу або відсилки до методу. Існують також декоратори для реалізації Retry та fallback-методів. Декоратори можна накладати один на одного, виконувати їх синхронно або асинхронно, що додає зручності та функціональності бібліотеці;
- реалізація CircuitBreaker в бібліотеці Resilience4j краще налаштовується та має ширші можливості. Наприклад, можна вказати певний поріг повільних або помилкових запитів для його відкриття. Або, наприклад, можна вказувати кількість пробних запитів у стані Half-Open, а не лише один, як в Hystrix, з відповідними пороговими значеннями для повернення у стан Closed, або в стан Open. Деталі роботи з цим патерном в бібліотеці Resilience4j наведені у 3 розділі;

- ця бібліотека надає спеціальні оператори для роботи з бібіотеками реактивного програмування, наприклад RxJava та Reactor.

Широкі можливості Resilience4j, її гнучкість та прогресивність, а також посилання розробників Netflix Hystrix на дану бібліотеку стали причиною використання саме цієї бібліотеки як основу для практичної частини даної дипломної роботи, яка описана у третьому розділі.

2.3 Пов’язані роботи

Задача посилення відмовостійкості застосунків є досить популярною у сфері інформаційних технологій. На цю тему існують численні наукові дослідження, наприклад наступні.

Circuit breaker, Discovery, та API gateways були оглянуті Монтесі та Вебер у [18]. Патерн Circuit Breaker був розглянутий у якості рішення проблем з комунікацією між мікросервісами через обмін повідомленнями, а також таймаути між взаємодіями сервісі.

Мендонса, Адералдо, Камара та Гарлан у [19] розглянули поведінку патернів Circuit Breaker та Retry використовуючи ймовірнісну модель PRISM. У результаті своєї роботи вони виокремили різні конфігурації налаштувань для цих патернів, які доводять, що при “правильному налаштуванні” патерни відмовостійкості значно знижують затримки у змаганні сервісів за ресурси в системі (resource contention).

Цікава ідея представлена в роботі Хаджара Хаміда Адділа [20], де детально розглядається патерн Circuit Breaker та, на думку автора, недолік у ньому, який проявляється у часі його очікування в стані Open перед тим, як перейти в стан Half-Open та спробувати надіслати повторний запит до зовнішнього сервісу. Автор пропонує власну модель реалізації патерну Circuit Breaker, яка написана на фреймворку Laravel та мові PHP. Схематично запропоновану в згаданій роботі модель (яка називається

DFTM – Dynamic Fault Tolerance Model) автор описує наступною схемою, представленою на Рисунку 4:

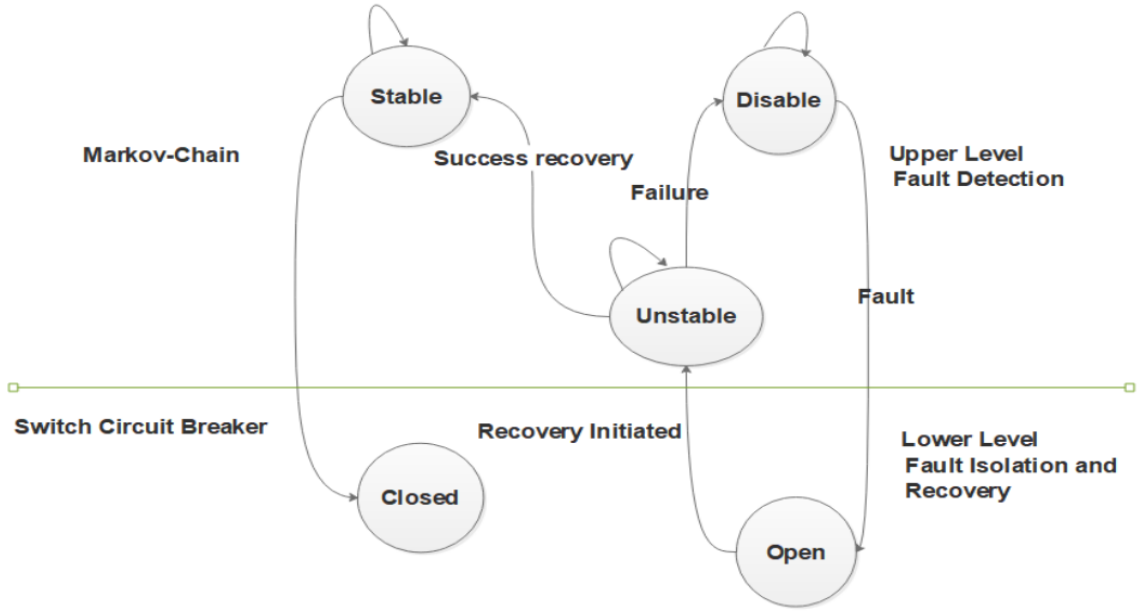

Рисунок 4. Граф переходів між станами моделі DFTM, представленої в роботі Х. Хаміда Адділа [20]

Ця реалізація Circuit Breaker працює як перемикач між лише двома станами (Switch Circuit Breaker), а замість Half-Open стану використовується окрема модель ланцюгів Маркова, яка має свої три стани – Stable, Unstable та Disable. Після появи ознак відмови у взаємодії між мікросервісами, Switch Circuit Breaker звертається до ланцюга Маркова для визначення того, чи може він повертатись до стану Closed, чи система зараз є нестабільною, або ж знаходиться у стані повної відмови. Це відбувається без додаткових затримок та таймаутів, що прискорює процес роботи патерну.

Проблематика, вказана в роботі Хаміда Адділа, може бути вирішена різними способами, що й дало поштовх для цього дослідження. Дана дипломна робота розвиває свою версію реалізації покращення патерну Circuit Breaker, та, на відміну від згаданої роботи, базується на використанні моделі прогнозування стану Circuit Breaker на основі метричних показників,

які збираються під час взаємодії двох мікросервісів. Деталі цієї реалізації описані у 3 розділі дипломної роботи.

3 ОПИС ПРАКТИЧНОЇ ЧАСТИНИ ДОСЛІДЖЕННЯ

3.1 Затримки у роботі патерну Circuit Breaker

Тема забезпечення відмовостійкості мікросервісів є досить широкою, особливо зважаючи на кількість практик та підходів які існують в сучасному світі інформаційних технологій. Саме тому для покращення ефективності та осмисленості даного дослідження, було вирішено звузити тематику практичної частини до покращення патерну Circuit Breaker, який пропонується багатьма бібліотеками, в тому числі тими, що були розглянуті у другому розділі.

Як вже було згадано, традиційна модель Circuit Breaker має три стани: Closed, Open та Half-Open. У стані Open цей інструмент відмовостійкості очікує деякий визначений час, який необхідний для того, щоб дати шанс мережі або цільовому сервісу відновити свою роботу. Після очікування, Circuit Breaker переходить в стан Half-Open, у якому дозволяється один або декілька (в залежності від реалізації та конфігурації) пробних запитів. Якщо всі вони (або деякий більше ніж деяке порогове значення) були помилковими – інструмент повертається в стан Open, де продовжить переривати наступні запити впродовж ще деякого часу, деколи навіть більшого ніж початковий. У разі ж успішності усіх (або деякої частини пробних запитів у стані Half-Open, інструмент переходить у стан Closed, де запити дозволяються, а система відновлює свою звичайну роботу.

Такий принцип є досить логічним бо рішення про дозвіл або заборону виконання запитів робиться на основі інформації отриманої (або не отриманої) від цільового сервісу. Проте, у ньому є недолік: час очікування у стані Open встановлюється заздалегідь за допомогою конфігурації застосунку або бібліотеки. Звісно, деякі бібліотеки мають реалізації алгоритмів подовження тривалості затримки з кожним наступним циклом переходу від Half-Open стану до Open стану (це означає що цільовий сервіс

або мережа є дуже навантаженими або знаходяться у стані відмови, і тому варто чекати все довше з кожною наступною невдалою спробою запиту). Проте, навіть такі алгоритми не дають впевненості у тому, що за кожну наступну одиницю часу яку «простоює» Circuit Breaker у стані очікування, цільовий сервіс ще не відновився.

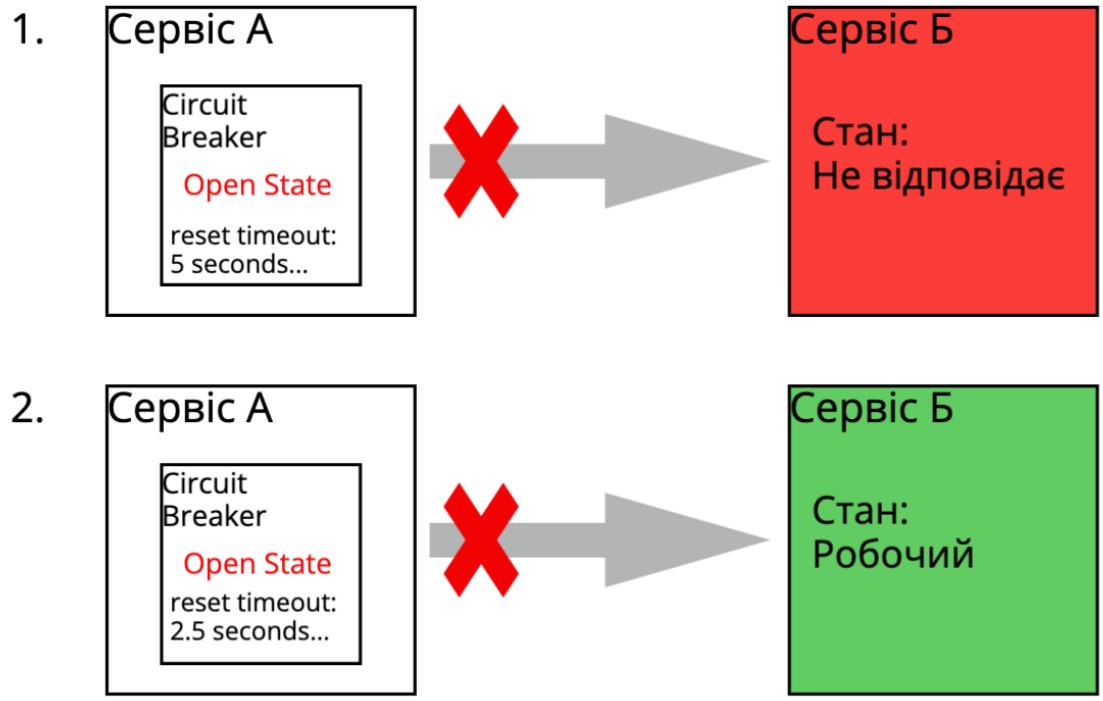

Рисунок 5. Процес очікування у стані Open State

Наприклад, нехай Circuit Breaker налаштований на очікування у стані Open впродовж 5 секунд. Схематично це зображено на Рисунку 5 вище. Тоді, у разі короткострокової відмови цільового сервісу або мережі, після декількох невдалих запитів Circuit Breaker перейде у стан Open та чекатиме 5 секунд, проте цільовий сервіс або мережа може в цей час у випадковий момент, наприклад через 2.5 секунди, відновити свою працездатність. У такому разі, Circuit Breaker буде блокувати всі наступні запити користувачів впродовж ще 2.5 секунд, хоча міг відновитись до станів Half-Open та Closed швидше, що надало б набагато більше користі користувачам системи. Звісно, ж, це лише приклад, і реальна конфігурація могла б очікувати іншу

кількість часу, а не 5 секунд. Проте, навіть при конфігурації патерну з меншими значеннями очікування у стані Open, традиційний Circuit Breaker все ще потенційно марнує деяку частку цього часу перед тим, як матиме інформацію про те, що система вийшла з відмови. В масштабах високонавантажених розподілених систем кожна доля секунди може мати значення для працездатності системи в цілому, особливо в умовах великої кількості одночасних запитів від користувачів.

Очевидно те, як важливо знаходити способи покращувати кожну деталь в таких відомих шаблонах відмовостійкості як Circuit Breaker. Якщо брати до прикладу відому компанію Netflix, яка використовує Circuit Breaker у своїх системах, як було згадано в попередньому розділі, то кожна лишня доля секунди у стані відмови може коштувати величезних грошових втрат (реальних чи потенційних), враховуючи масштаби бізнесу цієї корпорації.

3.2 Розробка моделі прогнозування переходу між станами

Для вирішення проблеми, описаної у попередньому пункті, необхідно знайти спосіб того, як можна більше покладатися на реальний стан системи на момент спроби отримання дозволу застосунком у Circuit Breaker на зовнішній запит.

У згаданій раніше роботі Хаміда Адділа [20] пропонується використання зміненої моделі Circuit Breaker, яка звертається до ланцюга Маркова для передбачення стану системи, а сам Circuit Breaker має лише два стани – Closed та Open. Згаданий ланцюг має свої 3 стани – Stable, Unstable, Disable. Для визначення ймовірності переходу між станами ланцюга використовується дві матриці – present matrix та transition matrix, які вираховуються за певними правилами та відповідно до показників швидкості запитів в системі. У разі знаходження ланцюга в стані Stable – стан Circuit Breaker визначається як Closed, в стані Unstable – система використовує механізм Retry для відновлення зв’язку із цільовим сервісом,

а в стані Disable Circuit Breaker переходить в стан Open та використовує дані з кешу для відповіді користувачу. У роботі автор доводить працездатність даного методу, але не вказує головне – скільки часу система знаходиться у стані Disable і чи повторює це той же принцип таймауту, який присутній в традиційному Circuit Breaker. Саме тому було вирішено розробити власну реалізацію з іншим підходом, але який намагається оптимізувати ту ж проблему.

Сконцентруємось на ідеї того, що мета Circuit Breaker – передбачити стан сервісу, до якого надається доступ на запит. У традиційній моделі, яка використовується у всіх згаданих у другому розділі бібліотеках [14, 15, 16, 17], існує 3 вже раніше згаданих стани, а рішення з надання доступу на запит після виявлення відмови робиться за рахунок виконання тестового запиту через конкретний встановлений заздалегідь час.

У рамках даного дослідження пропонується власна модель прогнозування стану системи. Основна ідея полягає у тому, що час затримки між переходом у стан Open та готовністю надавати доступ до запитів після виявлення відмови, буде розраховуватись динамічно, в залежності від попередніх результатів запитів а також метричних даних, які збираються під час роботи застосунку.

Пропонована модель містить лише два стани – Closed та Open. Замість очікування та виконання пробних запитів, після переходу у стан Open кожен наступний запит користувача викликатиме логіку обчислення деякого значення прогнозу, яке порівнюється з пороговим значенням, заданим статично. Це значення прогнозу означає собою рейтинг працездатності системи, або ж рейтинг готовності Circuit Breaker дозволити користувачу запит на цільовий сервіс. Якщо рейтинг більший ніж налаштоване заздалегідь значення – Circuit Breaker переходить у стан Closed та дозволяє запит. У разі помилкового припущення про працездатність цільового запиту, спрацьовує механізм fallback-методу, який віддає підготовлену відповідь що

може містити дані з кешу, деяке повідомлення про помилку, або результат будь-якої іншої бізнес-логіки для обробки даної ситуації. Також, для обмеження можливості перебування Circuit Breaker у стані Open занадто довго, поточний час перебування інструменту в цьому стані порівнюється із деяким заздалегідь встановленим максимальним значенням.

Даний алгоритм та набір станів можна зобразити наступним чином на рисунку 6:

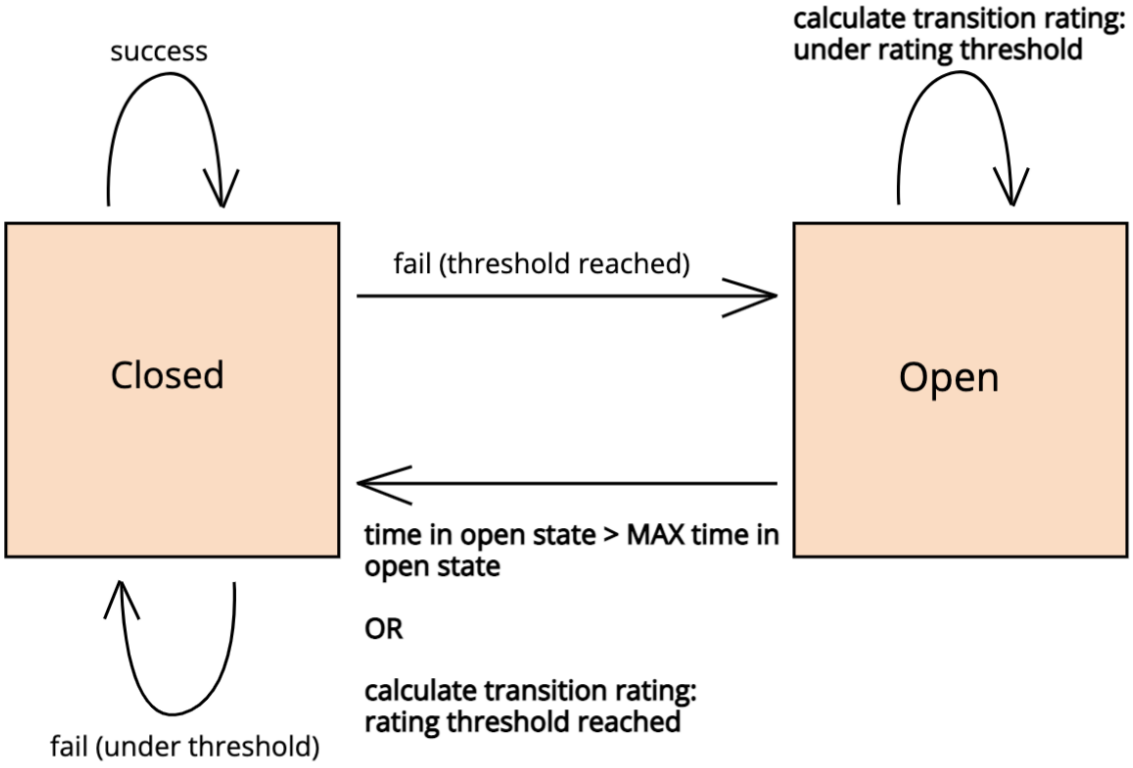

Рисунок 6. Стани та умови переходу між ними запропонованого методу реалізації Circuit Breaker

Згадане значення прогнозу розраховується за допомогою формули, яка враховує наступні метричні показники, відібрані із міркувань їх можливого впливу на рішення щодо працездатності системи:

- **failure rate** – співвідношення помилкових запитів до загальної кількості останніх запитів. До помилок не відносяться ті, які містять виключення бізнес-логіки, адже такі результати запитів

можуть мати вплив на поведінку системи зсередини та викликати свою бізнес-логіку обробки таких виключень. Також вони не означають відмову системи чи мережі, тому не мають враховуватись у даному показнику;

- **slow call rate** – співвідношення повільних запитів (тих, відповідь від яких тривала довше, ніж зазначений часовий поріг) до загальної кількості останніх запитів. Такі запити можуть свідчити про перенавантаження на сервіс або мережу;
- **success call rate** – співвідношення успішних запитів до загальної кількості останніх запитів. Успішні запити також можуть бути повільними, і враховуватись у попередньому показнику;
- **поточний час знаходження Circuit Breaker у стані Open (time in open state)**. Це значення повинне також порівнюватись із окремим встановленим максимальним пороговим значенням, у разі перевищення цього ліміту, Circuit Breaker повинен примусово дозволити запит. Це робиться для того, щоб обмежити час перебування в цьому стані.

Звісно ж, кількість та склад метрик, які впливають на рішення про працездатність сервісу може бути змінена, але у рамках даної роботи вирішено обмежитись представленим набором для того, щоб сконцентруватись на створенні саме нового підходу.

Оскільки в результаті обчислень Circuit Breaker повинен отримати значення прогнозу, або ж очікуваного рейтингу працездатності системи, яке повинне бути порівняне з деяким початковим значенням, вирішено обмежити їх у діапазоні $rating \in [0; 1]$, де значення рейтингу повинне бути дійсним числом.

Відповідно до таких умов, значення метрик у загальній формулі розрахунку рейтингу необхідно множити на унікальні для кожної метрики коефіцієнти та додавати між собою, а самі коефіцієнти повинні в сумі

дорівнювати одиниці. Також, коефіцієнти необхідні задля того, щоб давати можливість налаштовувати ступінь впливу кожної з метрик на результат обрахунку рейтингу. Наприклад, failure rate, очевидно, може бути більш важливий для визначення стану цільового сервісу, ніж slow call rate, тому коефіцієнт, на який множиться ця метрика буде більшим.

Отже, формулу обрахунку значення прогнозування можна представити наступним чином:

$$rating = c1 * (1 - FR) + c2 * (1 - SCR) + c3 * SR + \quad (3.1)$$
$$+ c4 * TOS/MOT$$
$$\text{при } c1 + c2 + c3 + c4 = 1,$$
$$FR, SCR, SR, TOS/MOT \in [0; 1],$$

де FR – Failure Rate, SCR – Slow Call Rate, SR – Success Rate, TOS – Time in Open State, MOT – Maximum Open Time – максимальний час перебування в стані Open, заданий конфігурацією застосунку, c1, c2, c3, c4 – відповідні коефіцієнти, задані конфігурацією застосунку.

Метрики, які мають негативний вплив на значення рейтингу відповідно віднімаються від 1.

3.3 Реалізація моделі прогнозування Circuit Breaker на основі бібліотеки Resilience4j

Як вже було згадано у другому розділі, практичну частину даної роботи вирішено реалізувати з використанням фреймворку Resilience4j, веб-фреймворку Spring Boot 3, та мови програмування Java версії 17. Вибір фреймворку, окрім його популярності, зумовлений тим, що у ньому вже є всі необхідні конфігурації, аспекти, реалізації патернів та методів для того, щоб інтегрувати той, чи інший пропонований патерн в застосунок. Найголовнішим також є те, що фреймворк має відкритий код та активну підтримку, тому розширення деяких його інтерфейсів було б потенційно практичнішим для використання у майбутньому.

Отже, реалізація описаного методу базується на імплементації інтерфейсів, які пов’язані з патерном Circuit Breaker із бібліотеки Resilience4j.

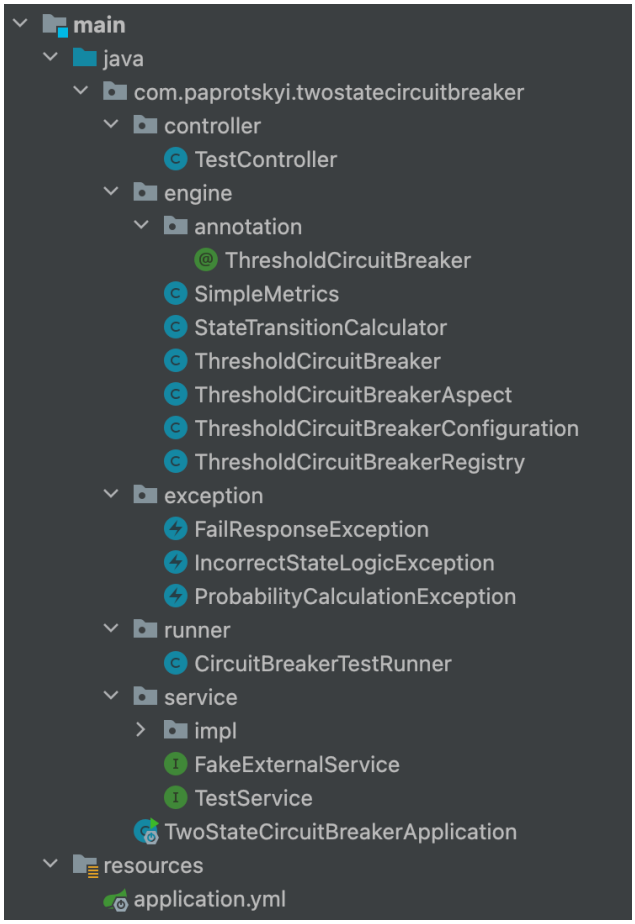

Рисунок 7. Структура проекту

На Рисунку 7 представлена структура проекту, де реалізація покращеного Circuit Breaker знаходиться в модулі engine. Головним класом імплементації методу являється **ThresholdCircuitBreaker**. Він названий в честь порогового значення (threshold), з яким порівнюється рейтинг отриманий з формули (3.1), представленої у попередньому пункті. Даний проект також містить класи необхідні для тестування ефективності даного методу, та для порівняння його з традиційною моделлю.

У бібліотеці Resilience4j існує також набір анотацій, які дозволяють позначати методи, які будуть обгортатись у необхідний реалізований нею патерн. Серед таких анотацій: @CircuitBreaker, @RateLimiter, @Bulkhead,

та інші [17]. Для підтримання аналогічного підходу до реалізації, у ці практичній роботі була створена схожа анотація - @ThresholdCircuitBreaker, яка обгортає цільовий метод (наприклад, призначений для відправки запиту до іншого сервісу) у ThresholdCircuitBreaker. Для того, щоб контекст фреймворку Spring мав усі необхідні створені об’єкти та щоб це працювало, необхідно також реалізувати спеціальний аспект ThresholdCircuitBreakerAspect, який сканує методи проекту на наявність цієї анотації, та створює або перевикористовує необхідний ThresholdCircuitBreaker всередині ThresholdCircuitBreakerRegistry – контейнера з наявними у застосунку екземплярами цього класу. Клас даного контейнеру прийшлось теж реалізувати спеціально для даного Circuit Breaker. Описану частину створених файлів схематично зображено на Рисунку 8:

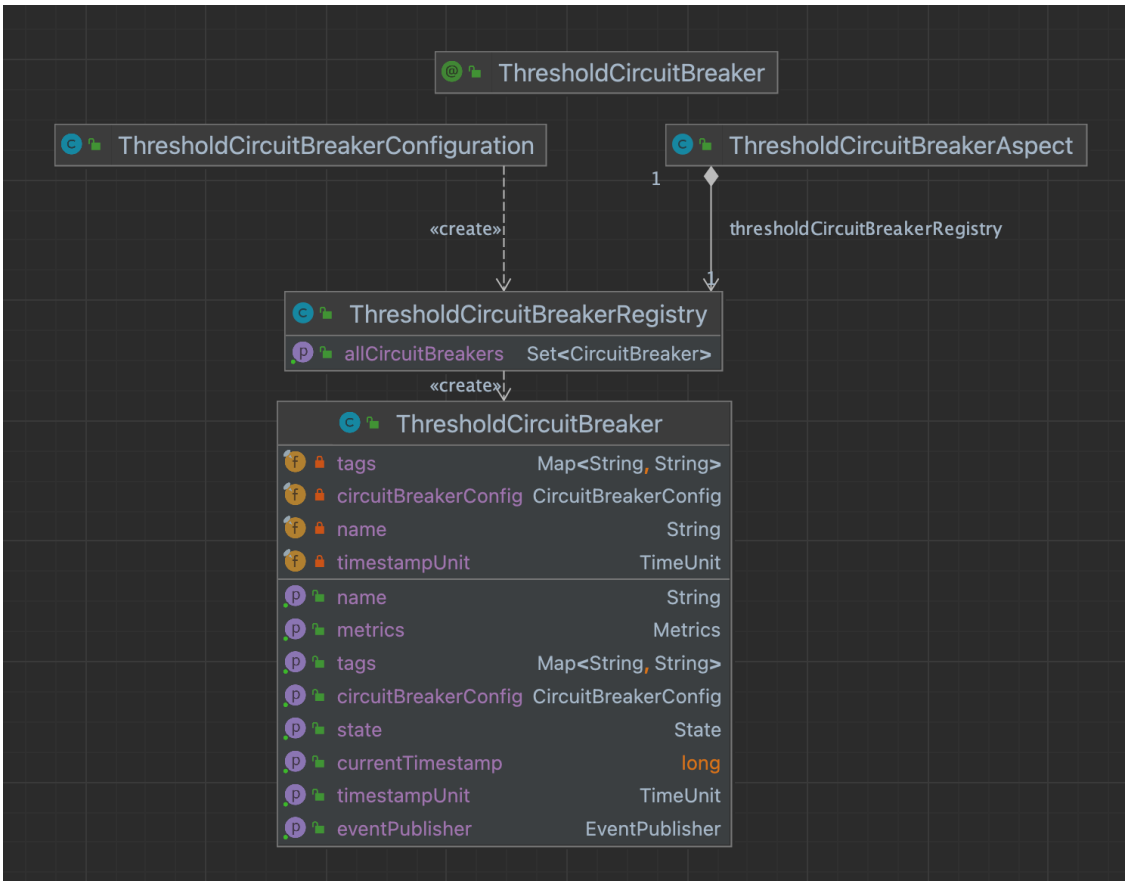

Рисунок 8. Класи для створення та конфігурації ThresholdCircuitBreaker

Описуючи реалізацію ThresholdCircuitBreaker, варто зазначити, що вона базується навколо роботи з метриками, які збираються під час виконання та отримання результатів запитів. В бібліотеці Resilience4j існує два варіанти збору метрик – **time-based sliding window** та **count-based sliding window** [17]. Сам sliding window являє собою деякий масив з об’єктами у які записуються дані з кожного запиту. Ці дані являють собою метрики, такі як: час виконання запиту, кількість повільних запитів, кількість помилкових запитів, кількість повільних помилкових запитів, кількість запитів. Кількість об’єктів у згаданому масиві налаштовується конфігурацією бібліотеки, та, як і більшість інших параметрів, може бути перевизначена користувачем. Time-based sliding window записує перелічені вище показники по всім запитам що відбулись за певний часовий проміжок у згаданий об’єкт в масиві. Count-based sliding window записує перелічені показники для кожного запиту, таким чином беручи до увагу певну кількість останніх з них. Коли об’єктів з цими даними стає більше, ніж заданий sliding window size, найстаріший об’єкт видаляється, а сам процес є циклічним.

Для даного дослідження вирішено реалізувати роботу з метриками, які використовують Count-based sliding window через прозорість та легшу вимірюваність. Безперечно, пропонований метод реалізації Circuit Breaker може працювати і з Time-based sliding window.

Отже, діаграма класів з реалізацією основної частини логіки даного методу, представлена на Рисунку 9:

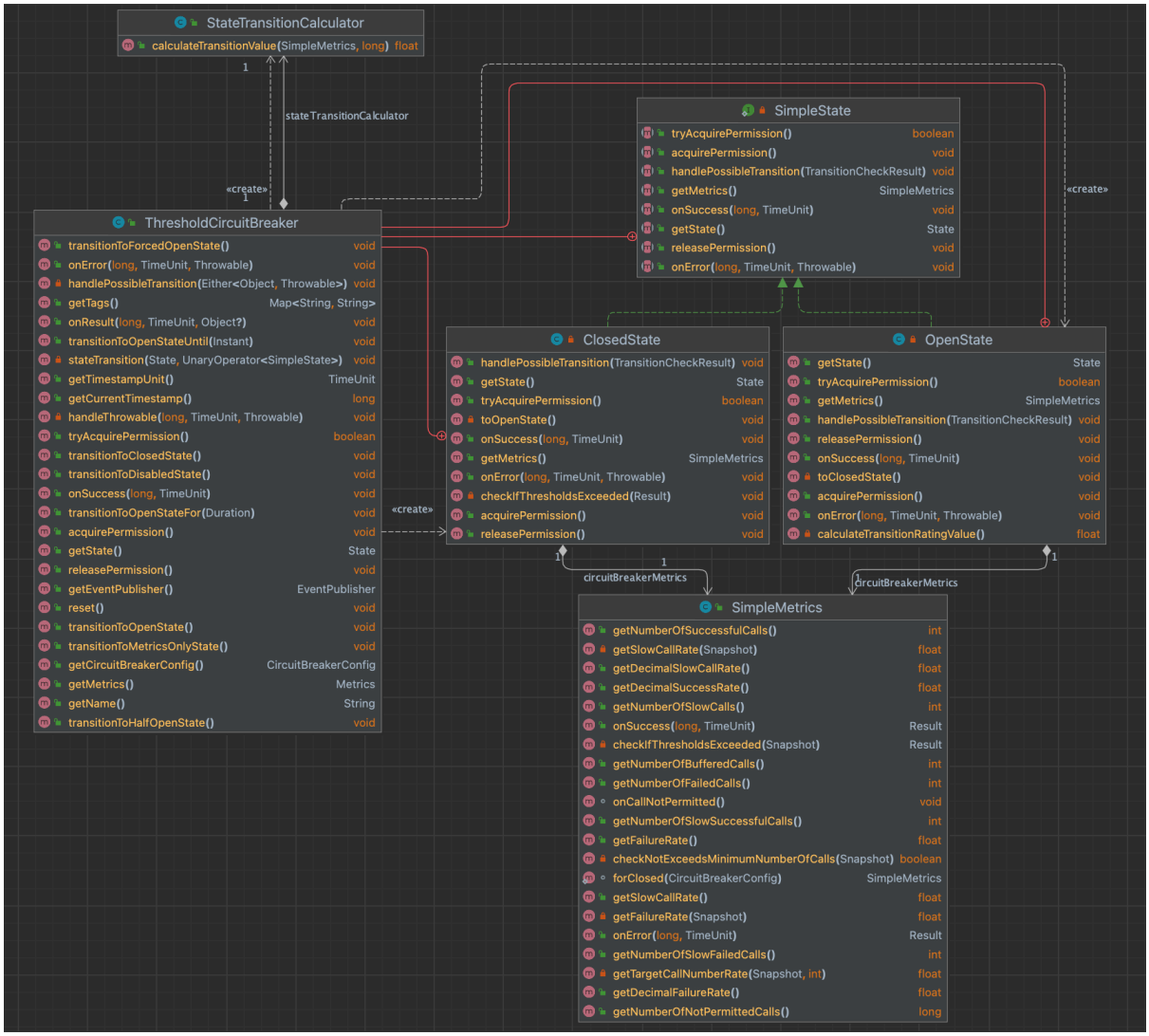

Рисунок 9. Діаграма основних класів та методів механізму ThresholdCircuitBreaker

Через невелику обмеженість у рівнях доступу класів та модулів поточної реалізації в бібліотеці Resilience4j, для цього дослідження довелось переписати реалізацію класу, який відповідає за збір метрик, аналогічну стандартній (SimpleMetrics).

На рисунку вище видно також інтерфейс SimpleState, який декларує спільні методи для двох класів ClosedState та OpenState (внутрішніх у ThresholdCircuitBreaker). ClosedState у багатопоточному режимі перевіряє, чи дійсно можна надавати користувачу доступ на запит, а також переходить в стан OpenState у разі перевищення порогових значень, налаштованих у застосунку. При виклику аналогічного методу (під назвою tryAcquirePermission()) на спробу отримання доступу на запит у стані

OpenState, викликається метод класу StateTransitionCalculator, який і пропонує головну концепцію підходу, описаного в даному дослідженні – обрахунок рейтингу, описаного формулою (3.1). **Java-код цього методу представлений у Додатку А**.

Основну ж логіку порівняння значення обрахованого рейтингу із встановленим пороговим значенням, а також рішення про перехід до стану ClosedState та наданням доступу на запит представлено **Java-кодом у Додатку Б**. Із коду видно, що на відміну від традиційної реалізації, у даній не використовується явно зазначеного постійного таймауту у стані OpenState, а ж лише максимальне значення часу, більше якого сервіс не може очікувати у цьому стані. У такому разі час очікування залежить від результатів останніх виконаних запитів, що дає можливість динамічного реагування на зміни у стані системи.

Варто також зазначити, що усі необхідні операції в класі ThresholdCircuitBreaker виконуються з використанням класу AtomicReference (наприклад, для роботи з об’єктами станів), а там, де необхідно – з методом synchronized. Це робиться для надання можливості працювати багатопоточно, чого й вимагає робота більшості веб-застосунків.

Як результат, описані вище класи дозволяють використовувати ThresholdCircuitBreaker на такому ж рівні, що і стандартна реалізація CircuitBreaker, над бібліотекою Resilience4j. Наприклад, на Рисунку 10 зображене використання відповідних анотацій для тестування відправки запитів до штучного зовнішнього сервісу, яке проводиться в експериментальному порівнянні, описаному в наступному пункті даного розділу:

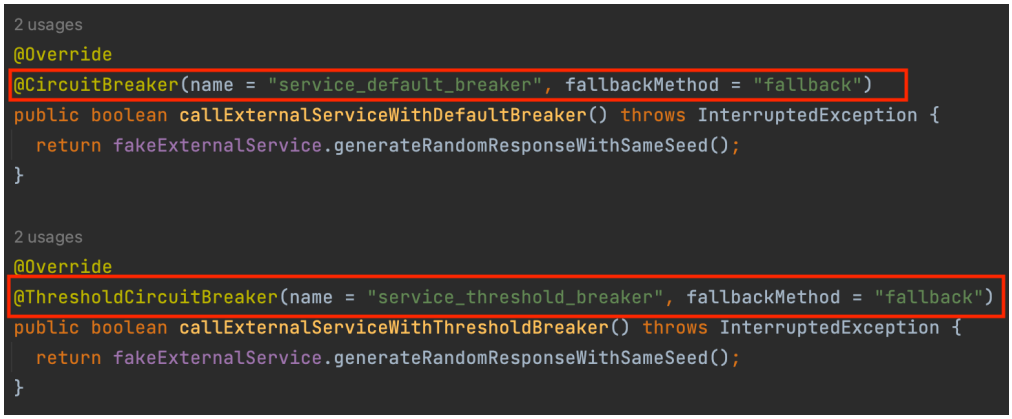

Рисунок 10. Фрагмент коду порівняльного використання анотацій @CircuitBreaker та @ThresholdCircuitBreaker

3.4 Експериментальні дослідження та їх результати

Для порівняння ефективності традиційної та запропонованої в цій роботі реалізацій, необхідно проаналізувати – що саме оптимізується покращеним підходом. Ціль нового підходу полягає в зменшенні очікування, за яке потенційно цільовий сервіс може бути відновленим. У такому разі, теоретично, при використанні Threshold Circuit Breaker кількість дозволених запитів із непомилковим результатом (тобто таких запитів, які були надіслані в момент, коли цільовий сервіс не був у стані відмови) має бути більшою, ніж при використанні традиційного методу.

Саме тому суть експериментів полягає у відтворенні такої ситуації, коли у рівних умовах обидва варіанти Circuit Breaker працюють із однаковою кількістю запитів до цільового сервісу, який входить у стан «відмови» з однаковою випадковою частотою та на однакові проміжки часу.

Для організації такої поведінки у експериментах було реалізовано штучний віддалений сервіс, який імітується Java-класом, та який має всередині метод, до якого звертаються виклики, обгорнуті в реалізації Circuit Breaker. Даний клас було розроблено таким чином, щоб він міг по таймеру з псевдовипадковою ймовірністю переходити в стан, коли він не може надавати відповідь. Через деякий псевдовипадковий час у визначених

межах він переходить у початковий стан, в якому віддає відповідь через псевдовипадковий інтервал у ще одних зазначених межах, імітуючи час на обробку своєї логіки. Кожне використання псевдовипадкових значень виконується з використанням однакового seed-числа для кожного експерименту, що гарантує однакову послідовність штучних «відмов» та однакові часові проміжки на штучне відновлення та відповідь для обох варіантів використання Circuit Breaker.

Варто зазначити, що експерименти також полягають у підборі найбільш ефективних величин для коефіцієнтів у формулі (3.1) обрахунку рейтингу, і порогового значення, з яким порівнюється результат формули. У даній роботі цей процес відбувається шляхом ручного підбору коефіцієнтів та значень між спробами та їх порівняння між собою.

Оскільки описаний процес підбору значень та умов для Circuit Breaker обох реалізацій зайняв багато часу та велику кількість спроб, різниця між якими полягала лише в зміні одного чи двох значень коефіцієнтів, порогу рейтингу, умов конфігурації для самого Circuit Breaker, або ж поведінки тестового сервісу, варто зобразити лише деякі фінальні дані, які продемонстрували найцікавіші показники. Умови та результати таких експериментів представлені на таблиці 3.1 нижче.

Таблиця 3.1 – Порівняння результатів деяких з проведених експериментів (рядки – окремі експерименти, жирним виділено значущі параметри для кожного експерименту, підкресленим – кращий результат з двох моделей у поточному експерименті)

Поведінка віддаленого сервісу	**Конфігурація обох варіантів Circuit Breaker**	**Коефіцієнти**	**Результати (Success call rate)**
Мін. час успішної відповіді: 50 мс; Макс. час відповіді: 6 с; Мін. час перевірки стану за таймером: 1 с; Макс. час перевірки стану за таймером: 10 с; Ймовірність переходу у стан відмови: 0.2; Мін. час відновлення зі стану відмови: 1 с; Макс. час відновлення зі стану відмови: 10с.	Кількість запитів: 100 Розмір sliding window: 20 Час очікування у Open State (для традиційного Circuit Breaker): 3 с. Failure Rate поріг: 50% Slow Call поріг: 3 с. **Поріг рейтингу для переходу у стан Closed: 0.45**	Failure rate: 0.4 Slow call rate: 0.2 Success call rate: 0.3 Time in open state: 0.1	Традиційний CB: 63% Threshold CB: 74%
Мін. час успішної відповіді: 50 мс; Макс. час відповіді: 3 с; Мін. час перевірки стану за таймером: 1 с; Макс. час перевірки стану за таймером: 10 с; Ймовірність переходу у стан відмови: 0.3; Мін. час відновлення зі стану відмови: 1 с;	Кількість запитів: 100 Розмір sliding window: 20 Час очікування у Open State (для традиційного Circuit Breaker): 5 с. Failure Rate Threshold: 40% Slow Call Threshold: 1.5 с.	Failure rate: 0.4 Slow call rate: 0.15 Success call rate: 0.35 Time in open state: 0.1	Традиційний CB: 51% Threshold CB: 61%

Макс. час відновлення зі стану відмови: 10с.	**Поріг рейтингу для переходу у стан Closed: 0.4**		
Мін. час успішної відповіді: 50 мс; Макс. час відповіді: 3 с; Мін. час перевірки стану за таймером: 1 с; Макс. час перевірки стану за таймером: 10 с; Ймовірність переходу у стан відмови: 0.3; Мін. час відновлення зі стану відмови: 1 с; Макс. час відновлення зі стану відмови: 10с.	Кількість запитів: 100 **Розмір sliding window: 10** Час очікування у Open State (для традиційного Circuit Breaker): 5 с. Failure Rate Threshold: 40% Slow Call Threshold: 1.5 с. **Поріг рейтингу для переходу у стан Closed: 0.4**	Failure rate: 0.4 Slow call rate: 0.15 Success call rate: 0.35 Time in open state: 0.1	Традиційний CB: 41% Threshold CB: 36%

Як можна побачити, запропонований Threshold Circuit Breaker показує себе краще ніж бібліотечна реалізація за показником кількості запитів, які успішно виконуються в умовах періодичних відмов цільового сервісу. Дійсно, перші два рядки таблиці відрізняються між собою деякими умовами відмови цільового сервісу, та коефіцієнтами, але в обох випадках спостерігається більш вдала пропускна здатність патерну Circuit Breaker.

Проте, третій рядок спеціально наведений, щоб показати, що якщо зменшувати такі параметри як sliding window size (яким він і відрізняється від другого), то Threshold Circuit Breaker має менше інформації про останні запити (10 замість 20 записів) на момент прийняття рішення про дозвіл на наступний запит, і, відповідно, показує гірший результат. Схожа ситуація виникає і тоді, коли коефіцієнти, порогове значення, або параметри конфігурації Circuit Breaker налаштовані невірно: під час численних

повторів експерименту зі зміною їх значень можна було також побачити погіршення результатів Threshold Circuit Breaker у порівнянні із стандартною реалізацією.

Як результат, вручну було підібрано наступні значення коефіцієнтів та порогового значення, опираючись на їх успішність при порівнянні кількості виконаних запитів між моделями:

- порогове значення рейтингу: 0.4 – 0.45;
- коефіцієнт впливу Failure Rate: 0.4;
- коефіцієнт впливу Slow call rate: 0.15;
- коефіцієнт впливу Success call rate: 0.35;
- коефіцієнт впливу Time in Open state: 0.1;
- максимальний час у стані Open: 10 секунд.

Важливою є умова переходу у стан Closed у разі перевищення максимального часу перебування у стані Open. Якщо її не дотримуватись в алгоритмі – цей компонент у загальній формулі (3.1) перевищить одиницю, що негативно вплине на правильність обрахунків.

3.5 Можливі напрямки розвитку методу

Зі сказаного у попередньому пункті стає зрозумілим те, що основною складністю впровадження даної моделі є правильний підбір значень коефіцієнтів та порогового значення. Також, все ще залишається викликом підбір параметрів самого патерну: очевидно те, що вибором неправильних параметрів конфігурації цього механізму можна тільки погіршити продуктивність системи, а ця складність все ще залишається викликом.

Тому, одним із можливих майбутніх напрямків покращення, або ж розвитку даного підходу до реалізації Circuit Breaker могла б бути спроба знаходження оптимальних значень усіх згаданих параметрів за допомогою алгоритмів машинного навчання. Для пошуку та підбору значень, які

свідчили б про працездатність запропонованого методу було проведено десятки ручних змін та перезапусків експерименту. Важко переоцінити те, наскільки швидше б відбувався процес, та наскільки точнішими могли б бути результати, враховуючи можливі помилки внаслідок присутності людського фактору.

Також, до формули (3.1) можна було б додавати більше показників метрик та їх коефіцієнтів, проте це може ускладнити пошуки оптимального алгоритму.

Висновки

В рамках даної роботи було досліджено сучасні підходи гарантування відмовостійкості з використанням мікросервісної архітектури. Було проаналізовано деякі популярні фреймворки та бібліотеки із забезпечення відмовостійкості мікросервісів. Під час їх аналізу та дослідження джерел інформації було виявлено можливість покращення одного із загальновідомих патернів відмовостійкості – Circuit Breaker. Запропоновано підхід до реалізації даного патерну, який враховує метричні показники що збираються застосунком під час його роботи, та контролює взаємодію мікросервісів шляхом обчислення деякого числового значення, формула для обрахунку якого була також запропонована цією роботою.

У практичній частині дипломної роботи реалізовано описаний вище підхід шляхом використання механізмів бібліотеки відмовостійкості Resilience4j, таким чином розширюючи її функціонал, а також мови програмування Java.

Для підтвердження правильності судження про можливість покращення патерну Circuit Breaker таким способом, була проведена організація експериментів що виконувались в однакових умовах для обох варіантів реалізації – традиційної (бібліотечної), та тої, що запропонована в цій роботі. Внаслідок порівняння пропускної здатності зовнішніх запитів до сервісу, який схильний до періодичної відмови, з використанням обох реалізацій було доведено, що за умови знаходження правильних коефіцієнтів та порогових значень, нова модель показує кращі результати ніж бібліотечна реалізація.

Отже, як висновок, запропонована модель поведінки Circuit Breaker може бути використана як розширення бібліотеки Resilience4j для застосунків, що працюють на JVM, або ж написана іншими мовами з використанням інших бібліотек чи фреймворків. Її використання, особливо

з правильним підбором параметрів, може покращити відмовостійкість систем з мікросервісною архітектурою, та пришвидшити їх роботу, навіть у порівнянні з традиційною реалізацією.

Список використаної літератури

1. Jet Brains. Developer Ecosystem 2022 survey [Electronic resource] / JetBrains. – 2022 – Mode of access: https://www.jetbrains.com/lp/devecosystem-2022/microservices/

2. Nadareishvili I. Microservice Architecture: Aligning Principles, Practices, and Culture [Electronic resource] / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – 2016 – Mode of access: https://books.google.com.ua/books/about/Microservice_Architecture.html?id=BvUEvgAACAAJ&redir_esc=y

3. Richardson C. Microservices Patterns [Electronic resource] / C. Richardson – 2018 – Mode of access: https://www.manning.com/books/microservices-patterns?a_aid=microservices-patterns-chris&a_bid=2d6d8a4d

4. Koren I. Fault-Tolerant Systems, 2nd edition [Electronic resource] / I. Koren, C. Mani Krishna – 2021 – Mode of access: https://www.elsevier.com/books/fault-tolerant-systems/koren/978-0-12-818105-8

5. Nygard M. T., Release It!, 2nd edition [Electronic resource] / M. T. Nygard - 2014 – Mode of access: http://repo.darmajaya.ac.id/4586/1/Release%20It%21_%20Design%20and%20Deploy%20Production-Ready%20Software%20%28%20PDFDrive%20%29.pdf

6. Brooker M. Exponential Backoff and Jitter. AWS Architecture Blog [Electronic resource] / M. Brooker – 2015 – Mode of access: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

7. Richards M. Microservices AntiPatterns and Pitfalls. O'Reilly Media [Electronic resource] / M. Richards – 2016 – Mode of access: https://www.oreilly.com/content/microservices-antipatterns-and-pitfalls/

8. Microsoft. Cloud Design Patterns. Bulkhead Pattern [Electronic resource] / Microsoft – 2023 – Mode of access: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead

9. Microsoft. Cloud Design Patterns. Rate Limiting Pattern [Electronic resource] / Microsoft – 2023 – Mode of access: https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting-pattern

10. Resilience4j. Documentation: RateLimiter [Electronic resource] / Resilience4j – 2023 – Mode of access: https://resilience4j.readme.io/docs/ratelimiter

11.May F. Medium: Microservices and Cascading Failures [Electronic resource] / F. May – 2018 – Mode of access: https://medium.com/@floyd.may/microservices-and-cascading-failures-16ec91c6ec9b

12. Fowler M. CircuitBreaker [Electronic resource] / M. Fowler – 2014 – Mode of access: https://martinfowler.com/bliki/CircuitBreaker.html

13. Slickcharts. S&P 500 Companies by Weight. [Electronic resource] / – 2023 – Mode of access: https://www.slickcharts.com/sp500

14. Netflix. Neflix Hystrix Wiki [Electronic resource] / Netflix – 2017 – Mode of access: https://github.com/Netflix/Hystrix/wiki/

15. Istio. Documentation. Version 1.17.2 [Electronic resource] / Istio Authors – 2023 – Mode of access: https://istio.io/latest/docs/

16. Sentinel. Documentation. [Electronic resource] / The Sentinel Authors – 2023 – Mode of access: https://sentinelguard.io/en-us/docs/introduction.html

17. Resilience4j. Documentation. [Electronic resource] / Resilience4j Authors – 2023 – Mode of access: https://resilience4j.readme.io/docs/getting-started

18. Montesi F. & Weber J. Circuit Breakers, Discovery, and API Gateways in Microservices [Electronic resource] / F. Montesi, J.Weber – 2016 – Mode of

access:
https://www.researchgate.net/publication/308320883_Circuit_Breakers_Discovery_and_API_Gateways_in_Microservices

19. Mendonça C. Model-Based Analysis of Microservice Resiliency Patterns [Electronic resource] / C. Mendonça, C. M. Aderaldo, J. Cámara, D. Garlan – 2020 – Mode of access:
https://www.researchgate.net/publication/339488033_Model-Based_Analysis_of_Microservice_Resiliency_Patterns

20. Hajar Hameed A. A Dynamic Fault Tolerance Model for Microservices Architecture [Electronic resource] / Addeen Hajar Hameed – 2019 – Mode of access:
https://openprairie.sdstate.edu/cgi/viewcontent.cgi?article=4417&context=etd

Додаток А
(обов'язковий)

Фрагмент коду методу обрахунку рейтингу (прогнозу), клас StateTransitionCalculator

```
public float calculateTransitionValue(@NonNull SimpleMetrics metrics,
                                      long currentOpenStateDurationInNanos)
{
  //always close the circuit breaker when the time in open state is longer
than the given threshold
  if (currentOpenStateDurationInNanos > OPEN_STATE_DURATION_THRESHOLD) {
    return Float.POSITIVE_INFINITY;
  }

  float decimalFailureRating = (1 - metrics.getDecimalFailureRate()) *
FAILURE_RATE_COEFFICIENT;
  float decimalSlowCallRating = (1 - metrics.getDecimalSlowCallRate()) *
SLOW_CALL_RATE_COEFFICIENT;
  float decimalSuccessCallRating = metrics.getDecimalSuccessRate() *
SUCCESS_CALL_RATE_COEFFICIENT;

  //time in open state
  float timeInOpenStateRating = (float)
      currentOpenStateDurationInNanos / OPEN_STATE_DURATION_THRESHOLD *
TIME_IN_OPEN_STATE_COEFFICIENT;

  return decimalFailureRating + decimalSlowCallRating +
decimalSuccessCallRating + timeInOpenStateRating;
}
```

Додаток Б

Фрагмент коду порівняння значення прогнозу та надання дозволу на запит, клас ThresholdCircuitBreaker.OpenState

```
@Override
public boolean tryAcquirePermission() {
  if (isOpen.get()) {
    float toClosedTransitionRating = calculateTransitionRatingValue();
    log.info("Calculated transition rating {}", toClosedTransitionRating);
    if (toClosedTransitionRating >= TRANSITION_RATING_THRESHOLD) {
      toClosedState();
      return true;
    }
    log.debug("Declining the request, because the state is still OPEN");
    circuitBreakerMetrics.onCallNotPermitted();
    return false;
  }
  return true;
}

private float calculateTransitionRatingValue() {
  long currentOpenStateDuration = getCurrentTimestamp() -
openStateTransitionTimestamp;
  log.debug("Current open state duration in nanos: {}",
currentOpenStateDuration);
  return
stateTransitionCalculator.calculateTransitionValue(circuitBreakerMetrics,
currentOpenStateDuration);
}
```