ПРОГРАМНО-ТЕХНІЧНІ КОМПЛЕКСИ

УДК 004.41

А.М. ГЛИБОВЕЦЬ
Національний університет «Києво-Могилянська академія», Київ, Україна,
e-mail: *a.glybovets@ukma.edu.ua*.

І.А. ПАПРОЦЬКИЙ
Національний університет «Києво-Могилянська академія», Київ, Україна,
e-mail: *i.paprotskyi@ukma.edu.ua*.

ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

Анотація. Мікросервісну архітектуру широко використовують під час побудови розподілених застосунків. Уже розв'язано багато проблем, що були наявні на початку використання цього підходу. Але досі залишається не розв'язаною одна з фундаментальних проблем, що серйозно впливає на відмовостійкість системи. Вона має назву «гарантована доставка повідомлень між сервісами». У статті проаналізовано стандартні підходи до розв'язання цієї проблеми. На основі проведеного аналізу виокремлено патерн Circuit Breaker та здійснено його модифікацію, що дало змогу зменшити затримку переходу між станами, а відповідно й відправлення повідомлень сервісом. Експериментально підтверджено ефективність запропонованої модифікації. Результат модифікації оформлено у вигляді швидкої конфігурації для Spring Boot.

Ключові слова: мікросервісна архітектура, відмовостійкість, мікросервіс, патерн, Circuit Breaker.

ВСТУП

Розробники сучасного програмного забезпечення дедалі частіше використовують мікросервісну архітектуру (МА) заради побудови більш гнучких та масштабованих застосунків. За даними опитування компанії JetBrains у 2022 р. близько 86 % респондентів застосовували мікросервісний підхід у дизайні своїх програмних систем [1]. Причини потреби у забезпеченні гнучкості та масштабованості очевидні: під час збільшення розміру програмної системи за рахунок додавання до неї нових складових частин, а також у разі зростання навантаження на систему необхідно зберегти її працездатність та ефективність роботи. Тому потрібно мати ефективні методи забезпечення надійності та стійкості в МА. Відмовостійкість (fault tolerance) є ключовою проблемою в забезпеченні надійності мікросервісів [2].

З-поміж патернів відмовостійкості фахівці виділяють патерн Circuit Breaker (CB), який працює подібно до автоматичного вимикача у реальних електричних системах. Загальновідома реалізація цього патерну включає в себе використання трьох станів. У них помічено застосування затримок під час переходу між станами, які можна оминути за рахунок прогнозування стану системи на основі метрик, які збирає патерн під час своєї роботи. У цій статті описано дослідження та розроблення власної моделі покращення відмовостійкості зазначеного патерну.

Кожен мікросервіс будують так, щоб його можна було якомога легше масштабувати, особливо у динамічному середовищі. Сервіси можуть взаємодіяти між собою шляхом синхронної та/або асинхронної комунікації. Цю взаємодію можна забезпечити за допомогою інтерфейсів та протоколів REST, RPC, протоколів обміну повідомленнями, вебсокетів або інших спеціальних протоколів передавання даних [1].

У стандартній МА клієнти звертаються до певного API Gateway (компонента, що розподіляє запити), який, у свою чергу, звертається до мікросервісів, кожен з яких може мати своє сховище даних. Зі свого боку сервіси можуть взаємодіяти між собою за певними протоколами передавання даних.

У розподілених системах та МА кожен сервіс може мати свій власний життєвий цикл. Під час взаємодії один сервіс може не «знати», що інший сервіс у цей момент не спроможний обробляти запити. Саме тому потрібно планувати та розробляти механізми відмовостійкості, як-от: оброблення помилок, резервне копіювання даних, моніторинг, відновлення після відмови тощо. Відмовостійкість можна забезпечити за допомогою різних підходів та технологій, а саме кластеризації, оркестрації, реплікації, автоматичного масштабування тощо. Серед найвідоміших патернів стабільності можна виділити такі: Timeouts, Retries, Circuit Breaker, Bulkheads, Rate Limiters [3]. Менш поширеними методами є Steady State, Fail Fast, Let It Crash, Handshaking, Test Harnesses, Decoupling Middleware, Shed Load, Governor тощо [3, 4].

Одна з проблем виникає тоді, коли є певний ланцюг викликів між сервісами. Цей процес називають каскадними відмовами (cascading failures) [5]. Одним зі способів подолання цієї проблеми є використання патерну Circuite Breaker (CB).

Головна ідея патерну CB є досить простою: деяка функція (наприклад та, що здійснює запит до цільового сервісу) обгортається об'єктом CB, який моніторить стан її виконання та відслідковує помилки. У разі перевищення певного порогу помилок, які стаються впродовж викликів цієї функції, об'єкт перемикача переходить у стан, в якому всі наступні виклики цієї функції повертатимуть помилку, при цьому сама функція виконуватися не буде. Зазвичай система у певний спосіб отримує автоматичні сповіщення про перехід у цей стан [3, 6].

Патерн CB імітує таку поведінку за рахунок перебування у певних станах, які відповідають за його реакцію на виклики функції взаємодії. Класична реалізація цього патерну передбачає три стани перемикача [6].

1. Стан Closed (закритий). Виклики дозволяються, поки вони успішні (success) до певного порогу (under threshold) кількості перехоплених помилок під час виконання функції (fail).

2. Стан Open (відкритий). У разі перевищення порогу кількості перехоплених помилок (threshold reached) усі запити до функції будуть відхилятися з помилкою, при цьому сама функція виконуватися не буде. Через певний встановлений часовий проміжок (reset timeout) після переходу в цей стан він змінюється на наступний.

3. Стан Half-Open (напіввідкритий). Після завершення таймауту вимикач ніби «закривається» на деяку кількість дозволених запитів, щоб протестувати функцію взаємодії на предмет повторення помилок. У разі їхньої відсутності стан змінюється на Closed, де всі запити знову будуть дозволятись, в іншому разі стан повертається до стану Open і перебуватиме у ньому протягом оновленого таймауту.

Цю досить просту схему роботи патерну можна налаштовувати залежно від потреб та реалізації. Наприклад, можна задавати різні значення таймаутів у разі очікування у стані Open, встановлювати різні порогові значення помилок у стані Closed та стані Half-Open, фільтрувати частину помилок за їхнім типом під час обчислення перевищення порогових значень, використовувати функції зво-

ротної роботи (fallback) які спрацьовують у разі помилки всередині функції взаємодії, або під час відхилення запиту самим вимикачем відповідно до його поточного стану тощо.

Патерн CB можна також застосовувати у комбінації з іншими патернами забезпечення відмовостійкості, наприклад, Retry, Timeout (або Time Limiter), Bulkhead, Rate Limiter. Простий у застосуванні шаблон CB вибрано у цьому дослідженні для покращення саме через те, що його широко застосовують на практиці.

Є багато бібліотек та фреймворків підтримки патернів із забезпечення відмовостійкості: Netflix Hystrix [7], Istio [8], Sentinel [9] тощо.

1. ОГЛЯД НАЯВНИХ РІШЕНЬ

Задача підвищення відмовостійкості застосунків є досить поширеною в галузі інформаційних технологій. Її досліджено у численних наукових працях. Огляд патернів CB, Discovery, та API gateways представлено у [10]. Зокрема, патерн CB застосовано для розв'язання проблем з комунікацією між мікросервісами під час обміну повідомленнями та аналізу таймаутів між взаємодіями сервісів.

В [11] розглянуто поведінку патернів CB та Retry з використанням ймовірнісної моделі PRISM. Автори виокремили різні конфігурації налаштувань для цих патернів, які доводять, що у разі «правильного налаштування» патерни відмовостійкості значно знижують затримки у змаганні сервісів за ресурси в системі (resource contention).

Цікаву ідею представлено в роботі [12]. Автор зазначає, що основними недоліками застосування патерну CB є його тривале очікування у стані Open перед тим, як він перейде до стану Half-Open, та його спроба надіслати повторний запит до зовнішнього сервісу. Також автор пропонує власну модель DFTM (Dynamic Fault Tolerance Model) реалізації патерну CB з використанням фреймворку Laravel та мови PHP. Роботу моделі DFTM автор описує схемою, наведеною на рис. 1.

Тут CB відіграє роль перемикача між двома станами. Замість стану Half-Open використано окрему модель ланцюгів Маркова, яка має три власні стани: стабільний (Stable), нестабільний (Unstable) та вимкнутий (Disable). Після появи ознак відмови у взаємодії між мікросервісами Switch Circuit Breaker

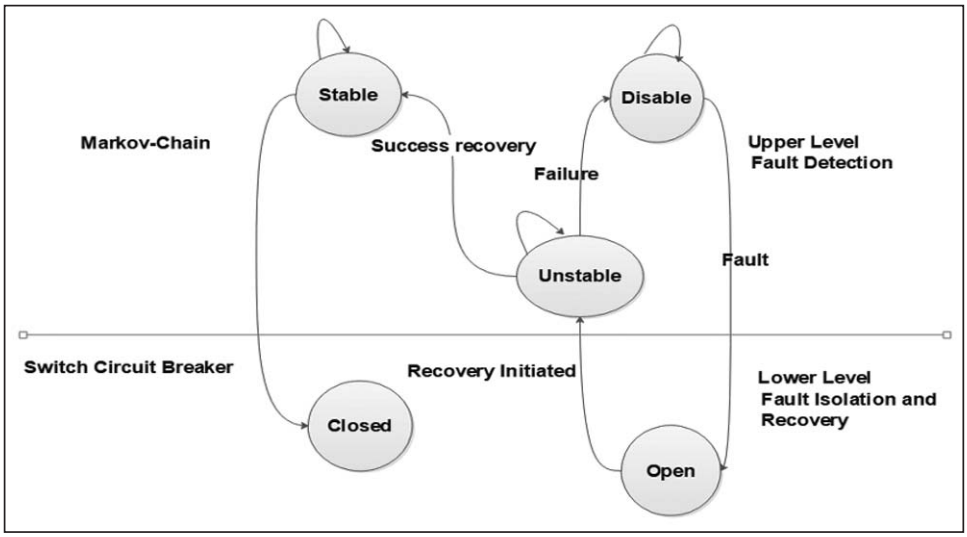

Рис. 1. Граф переходів між станами моделі DFTM, представленої у [12]

звертається до ланцюга Маркова для того, щоб визначити, чи можна повернутися до стану Closed, а також те, якою є на цей момент система — нестабільною, чи то перебуває у стані повної відмови. Це відбувається без додаткових затримок та таймаутів, що прискорює процес роботи патерну.

Для визначення ймовірності переходу між станами ланцюга використано дві матриці — поточну (present matrix) та матрицю переходів (transition matrix), які обчислено за певними правилами та відповідно до показників швидкості запитів у системі. У разі перебування ланцюга у стані Stable стан СВ визначається як Closed. Якщо ланцюг перебуває в стані Unstable, то система використовує механізм Retry для відновлення зв'язку із цільовим сервісом. Перебуваючи у стані Disable, Circuit Breaker переходить у стан Open та використовує дані з кешу для відповіді користувачу. У [12] доведено працездатність зазначеного методу, але не вказано головне — скільки часу система перебуває у стані Disable і чи повторює це той самий принцип таймауту, наявний у традиційному СВ.

Потреба в усуненні цього недоліку стала стимулом до проведення дослідження, описаного у цій статті. Тут автори представили власну версію реалізації покращення патерну СВ. В її основу покладено модель прогнозування стану СВ з використанням метричних показників, збір яких здійснюється під час взаємодії мікросервісів. Опишемо цю модель та її реалізацію.

2. МОДЕЛЬ ТА РЕАЛІЗАЦІЯ

2.1. Затримки в роботі патерну Circuit Breaker. Нагадаємо, що традиційна модель патерну Circuit Breaker має три стани. У стані Open цей патерн протягом наперед визначеного проміжку часу перебуває у режимі очікування і тим самим надає шанс мережі або цільовому сервісу відновити свою роботу. Після очікування протягом певного проміжку часу (зазвичай встановленого в конфігурації застосунку) СВ переходить у стан Half-Open. У цьому стані дозволено здійснення одного або кількох (залежно від реалізації та конфігурації) пробних запитів. Якщо всі пробні запити виявилися помилковими, модель повертається у стан Open. Перебуваючи у ньому, вона продовжує переривати наступні запити впродовж ще деякого часу, деколи навіть більшого за початковий (значення цього параметра також зазвичай налаштовують у конфігурації застосунку). У разі успішності всіх (або деякої частини) пробних запитів під час перебування моделі у стані Half-Open вона переходить у стан Closed. У цьому стані запити дозволяються, а система відновлює свою звичайну роботу.

Цей принцип є досить логічним, оскільки рішення про дозвіл або заборону виконання запитів ухвалюється на основі інформації, отриманої (або не отриманої) від цільового сервісу. Проте він має такий недолік: час очікування у стані Open встановлюється заздалегідь за допомогою конфігурування спеціального застосунку або бібліотеки. Звісно, деякі бібліотеки мають реалізації алгоритмів збільшення тривалості затримки з кожним наступним циклом переходу від стану Half-Open до стану Open. Це означає, що цільовий сервіс або мережа є дуже навантаженими або перебувають у стані відмови, і тому з кожною наступною невдалою спробою запиту зростає час затримки. Проте навіть ці алгоритми не дають упевненості у тому, що за кожну наступну одиницю часу, протягом якої СВ «простоює» у стані очікування, цільовий сервіс ще не відновився.

2.2. Модель прогнозування переходу між станами. Для розв'язання проблеми, описаної у п. 2.1, потрібно з'ясувати, в який спосіб можна більше покладатися на реальний стан системи у той момент, коли застосунок здійснює спробу отримати у СВ дозвіл на зовнішній запит.

Нагадаємо, що головна задача СВ — передбачити стан сервісу, до якого надсилається запит. Традиційна модель, яку використано у дослідженнях [7–9], містить три раніше зазначені стани, а рішення щодо надання доступу на здійснення запиту після виявлення відмови ухвалюють шляхом виконання тестового запиту через конкретний встановлений заздалегідь час.

У цій роботі запропоновано власну модель прогнозування стану системи. Основна ідея полягає у тому, що обчислення часу затримки між переходом у стан Open та готовністю надавати доступ на здійснення запитів після виявлення відмови, здійснюватиметься у динамічному режимі залежно від попередніх результатів запитів, а також метричних даних, зібраних під час роботи застосунку.

Пропонована модель містить лише два стани — закритий (Closed) та відкритий (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 фактично визначає рейтинг працездатності системи, або рейтинг готовності СВ дозволити користувачу запит на цільовий сервіс. Якщо рейтинг більший за налаштоване заздалегідь значення, то СВ переходить у стан Closed та дозволяє запит. У разі помилкового припущення про працездатність цільового запиту спрацьовує механізм fallback-методу, який надає підготовлену відповідь, що може містити дані з кешу, деяке повідомлення про помилку, або результат будь-якої іншої бізнес-логіки для оброблення цієї ситуації. До того ж, щоб унеможливити занадто тривале перебування СВ у стані Open, поточний час перебування інструмента у цьому стані порівнюється з деяким заздалегідь встановленим максимальним значенням.

Схему роботи запропонованої моделі прогнозування переходу та набір станів представлено на рис. 2.

Прогнозне значення розраховують за допомогою формули, в якій взято до уваги наведені нижче метричні показники (далі — метрики), вибрані з міркувань їхнього можливого впливу на рішення щодо працездатності системи.

• Коефіцієнт відмов (failure rate) — відношення кількості помилкових запитів до загальної кількості останніх запитів. До помилкових запитів не відносять помилки, які містять виключення бізнес-логіки, адже такі результати запитів можуть мати вплив на поведінку системи зсередини та викликати свою

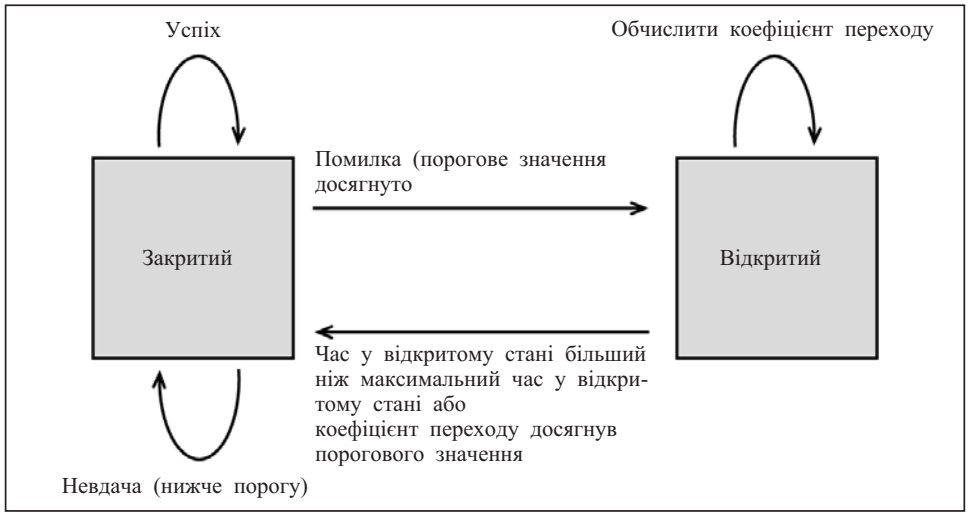

Рис. 2. Граф переходів між станами моделі прогнозування переходів

бізнес-логіку оброблення таких виключень. Також вони не означають відмову системи чи мережі, тому їх не слід враховувати у цьому показнику.

• Коефіцієнт повільних запитів (slow call rate) — відношення кількості повільних запитів (тобто тих, відповідь на які тривала довше встановленого часового порогу) до загальної кількості останніх запитів. Такі запити можуть свідчити про перенавантаження сервісу або мережі.

• Коефіцієнт успішних викликів (success call rate) — відношення кількості успішних запитів до загальної кількості останніх запитів. Успішні запити також можуть бути повільними, і враховуватись у попередньому показнику.

• Поточний час перебування СВ у стані Open (time in open state). Це значення також слід порівнювати з окремим встановленим максимальним пороговим значенням. У разі перевищення цього значення СВ повинен примусово дозволити запит. Це роблять для того, щоб обмежити час перебування в цьому стані.

Звісно, кількість та склад метрик, які впливають на рішення щодо працездатності сервісу, можуть змінюватися, але у межах цього дослідження вирішено обмежитися представленим набором для того, щоб зосередитися на створенні саме нового підходу.

У результаті обчислень СВ повинен отримати значення прогнозованого або очікуваного рейтингу працездатності системи, яке потрібно порівняти з деяким початковим значенням. Тому вирішено обмежити отримувані значення діапазоном $rating \in [0,1]$, в якому значення рейтингу має бути дійсним числом.

Відповідно до цих умов значення метрик у загальній формулі обчислення рейтингу слід множити на унікальні для кожної метрики коефіцієнти та додавати між собою, а самі коефіцієнти повинні в сумі дорівнювати одиниці. Коефіцієнти також потрібні для того, щоб можна було налаштовувати ступінь впливу кожної метрики на результат обчислення рейтингу. Наприклад, метрика failure rate, очевидно, може бути більш важливою для визначення стану цільового сервісу, ніж метрика slow call rate, тому коефіцієнт, на який множать цю метрику, буде більшим.

Тому отримано таку формулу обчислення прогнозованого значення рейтингу працездатності:

$$rating = c1 \cdot (1 - FR) + c2 \cdot (1 - SCR) + c3 \cdot SR + c4 \cdot \frac{TOS}{MOT}$$

$$\text{для } c1 + c2 + c3 + c4 = 1,$$

$$FR, SCR, SR, \frac{TOS}{MOT} \in [0,1],$$

де FR — метрика failure rate, SCR — slow call rate, SR — success call rate (частота успішних запитів), TOS — time in open state, MOT — maximum open time, тобто максимальний час перебування у стані Open, заданий конфігурацією застосунку, $c1$, $c2$, $c3$, $c4$ — відповідні коефіцієнти, задані конфігурацією застосунку.

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

2.3. Реалізація моделі прогнозування Circuit Breaker. Реалізовано модель з використанням фреймворку Resilience4j, вебфреймворку Spring Boot 3, та мови програмування Java версії 17. Вибір фреймворку зумовлений тим, що він містить всі необхідні конфігурації, аспекти, реалізації патернів та методів для того, щоб інтегрувати той чи інший пропонований патерн у застосунок. До того ж цей фреймворк має відкритий код та активну підтримку.

Отже, реалізація описаного методу ґрунтується на імплементації інтерфейсів, які пов'язані з патерном CB з бібліотеки Resilience4j.

На рис. 3 наведено структуру проєкту, де реалізація покращеного Circuit Breaker знаходиться в модулі engine. Головним класом імплементації методу є ThresholdCircuitBreaker. Проєкт також містить класи, необхідні для тестування ефективності методу та для порівняння його з традиційною моделлю.

Бібліотека Resilience4j також містить набір анотацій, які дають змогу позначати методи, що будуть обгорнуті у необхідний реалізований нею патерн. Це, зокрема, анотації @CircuitBreaker, @RateLimiter, @Bulkhead тощо [13]. Створено схожу анотацію @ThresholdCircuitBreaker, яка обгортає цільовий метод (наприклад, призначений для відправлення запиту до іншого сервісу) у ThresholdCircuitBreaker. Для того, щоб контекст фреймворку Spring мав усі необхідні створені об'єкти та щоб це працювало, у цьому дослідженні розроблено аспект ThresholdCircuitBreakerAspect. Цей аспект сканує методи проєкту на наявність зазначеної анотації та створює або перевикористовує необхідний ThresholdCircuitBreaker всередині ThresholdCircuitBreakerRegistry — контейнера з наявними у застосунку екземплярами цього класу. Клас цього контейнера теж довелося реалізувати. Описану частину створених файлів схематично зображено на рис. 4.

Описуючи реалізацію ThresholdCircuitBreaker, варто зазначити, що вона ґрунтується на використанні метрик, збір яких здійснюється під час виконання та отримання результатів запитів. У бібліотеці Resilience4j є два варіанти збору метрик. Це методи «Time-based Sliding Window» та «Count-based Sliding Window» [13]. Власне «ковзне вікно» (sliding window) являє собою деякий масив з об'єктами, у які записуються дані з кожного запиту. Цими даними є такі метрики: час виконання запиту, кількість повільних запитів, кількість помилкових запитів, кількість повільних помилкових запитів, кількість запитів. Кількість об'єктів у цьому масиві налаштовують конфігуруванням бібліотеки. Користувач може перевизначити цю величину, як і більшість інших параметрів. Метод Time-based Sliding Window дає змогу записати зазначені метрики за всіма запитами, що відбулися за певний часовий проміжок, у відповідний об'єкт у масиві. За допомогою методу Count-based Sliding Window записують ці метрики для кожного запиту, у такий спосіб беручи до уваги певну кількість останніх з них. Коли об'єктів із цими даними стає більше, ніж заданий розмір ковзного вікна (sliding window size), найстаріший об'єкт видаляється.

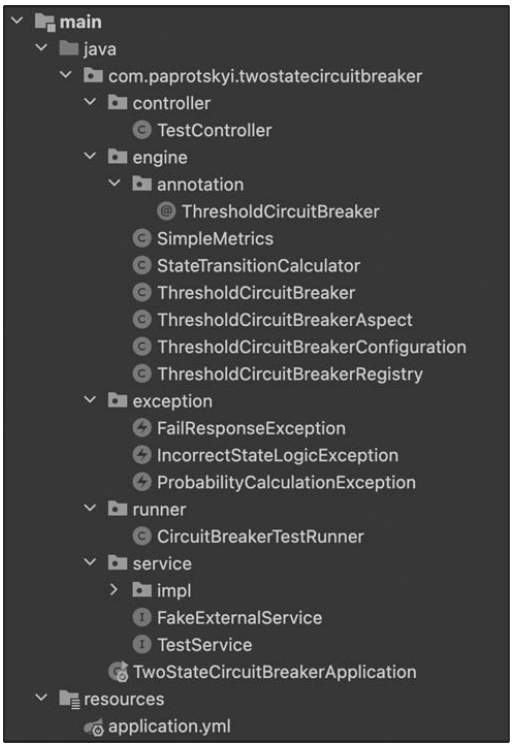

Рис. 3. Структура проєкту

У цій статті реалізовано роботу з метриками методом Count-based Sliding Window через прозорість в оцінюванні результатів. Безперечно, пропонований метод реалізації CB може працювати і з Time-based Sliding Window.

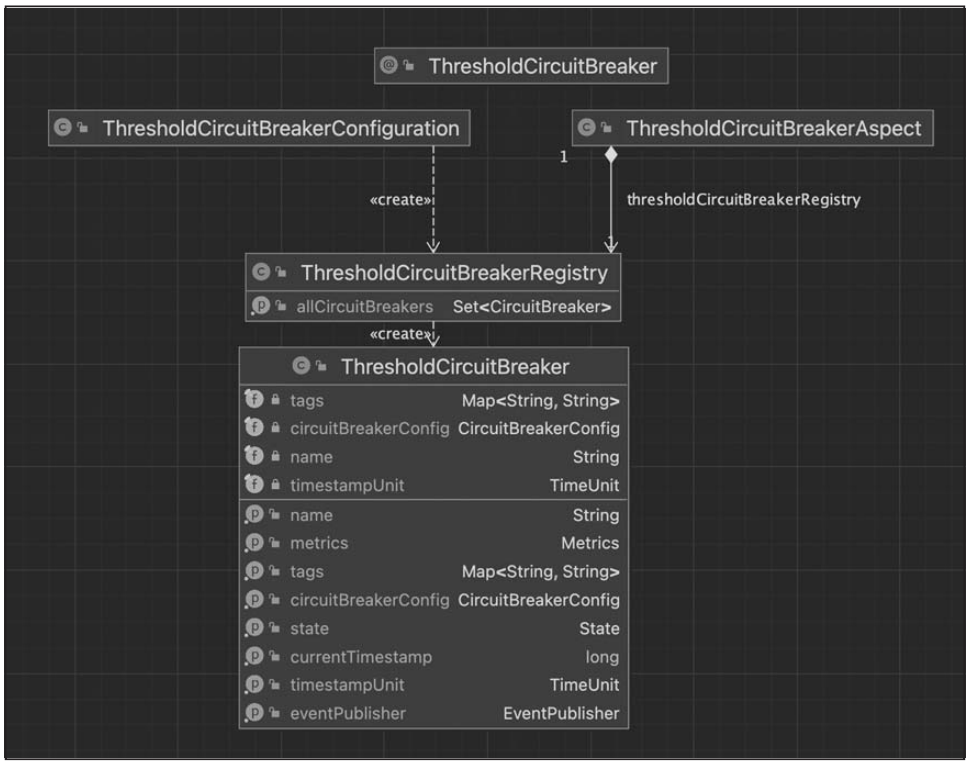

Рис. 4. Класи для створення та конфігурування ThresholdCircuitBreaker

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

Інтерфейс SimpleState декларує спільні методи для двох класів ClosedState та OpenState (внутрішніх у ThresholdCircuitBreaker). Об'єкт класу ClosedState у багатопотоковому режимі перевіряє, чи дійсно можна надавати користувачу доступ на надіслання запиту, а також переходить в стан OpenState у разі перевищення порогових значень, налаштованих у застосунку. Під час виклику аналогічного методу (під назвою tryAcquirePermission()) на спробу отримання доступу на надіслання запиту у стані OpenState, викликається метод класу StateTransitionCalculator, який і пропонує головну концепцію підходу — обчислення рейтингу за запропонованою вище формулою.

Автори цієї статті не використовували явно зазначеного постійного таймауту у стані OpenState, а маніпулювали лише максимальним значенням часу, більше якого сервіс не може очікувати у цьому стані. Тому час очікування залежить від результатів останніх виконаних запитів, що дає можливість динамічного реагування на зміни у стані системи.

Варто також зазначити, що всі необхідні операції у класі ThresholdCircuitBreaker виконуються з використанням класу AtomicReference (наприклад, для роботи з об'єктами станів), а там, де потрібно — із застосуванням ключового слова synchronized. Це дає можливість працювати у багатопотоковому режимі, чого й потребує робота більшості вебзастосунків.

Як результат, описані вище класи дають змогу використовувати ThresholdCircuitBreaker на такому самому рівні, що і стандартна реалізація CircuitBreaker, над бібліотекою Resilience4j.

3. ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ЇХНІ РЕЗУЛЬТАТИ

Важливою метою модифікації було зменшення часу очікування, за який цільовий сервіс може відновитися. Тоді, у разі використання 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 у разі застосування традиційного методу.

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

Для організації такої поведінки в експериментах реалізовано штучний віддалений сервіс, який імітується Java-класом та має вбудований метод, до якого звертаються виклики, обгорнуті в реалізації СВ. Цей клас розроблено так, щоб його екземпляр міг за таймером з псевдовипадковою ймовірністю переходити у стан, в якому він не може надавати відповідь. Через деякий псевдовипадковий час у визначених межах екземпляр переходить у початковий стан, в якому надає відповідь через псевдовипадковий інтервал у ще одних зазначених межах, імітуючи час на оброблення своєї логіки. Кожне застосування псевдовипадкових значень виконується з використанням однакового seed-числа для кожного експерименту. Це гарантує однакову послідовність штучних «відмов» та однакові часові проміжки, потрібні для штучного відновлення й відповіді для обох варіантів використання СВ.

Варто зазначити, що експерименти також полягають у підборі величин для коефіцієнтів у формулі (1) обчислення рейтингу, а також порогового значення, з яким порівнюється результат застосування формули. Цей процес здійснюють шляхом ручного підбору коефіцієнтів. Надалі можна застосовувати методи машинного навчання для автоматизованого підбору цих параметрів всередині працюючої системи.

Оскільки описаний процес підбору значень та умов для СВ в обох реалізаціях потребував багато часу та великої кількості спроб, різниця між якими полягала лише у зміні одного чи двох значень коефіцієнтів, порогу рейтингу, умов конфігурації для самого СВ, або поведінки тестового сервісу, варто навести лише деякі фінальні дані щодо найцікавіших показників. Умови та результати цих експериментів представлено в табл. 1.

З табл. 1 видно, що запропонований Threshold Circuit Breaker працює краще ніж бібліотечна реалізація за показником кількості запитів, які успішно виконуються в умовах періодичних відмов цільового сервісу. Дійсно, перші два рядки табл. 1 відрізняються між собою деякими умовами відмови цільового сервісу та коефіцієнтами, але в обох випадках спостерігається більш вдала пропускна здатність патерну СВ.

Проте третій рядок наведено спеціально для демонстрації того, що у разі зменшення таких параметрів як розмір ковзного вікна (sliding window size) (яким цей рядок і відрізняється від другого), Threshold Circuit Breaker має менше інформації про останні запити (10 записів замість 20) на момент прийняття рішення щодо надання дозволу на надіслання наступного запиту і відповідно показує гірший результат. Схожа ситуація виникає й тоді, коли коефіцієнти, порогове значення або параметри конфігурації СВ налаштовані неправильно: під час численних повторів експерименту зі зміною їхніх значень можна було також спостерігати погіршення результатів застосування Threshold Circuit Breaker відносно результатів застосування стандартної реалізації.

Таблиця 1. Порівняння результатів деяких проведених експериментів

№ рядка	Поведінка віддаленого сервісу	Конфігурація обох варіантів Circuit Breaker	Коефіцієнти	Результати (Success call rate)
1	Мінімальний час успішної відповіді: 50 мс Максимальний час відповіді: 6 с Мінімальний час перевірки стану за таймером: 1 с Максимальний час перевірки стану за таймером: 10 с Ймовірність переходу у стан відмови: 0.2 Мінімальний час відновлення зі стану відмови: 1 с Максимальний час відновлення зі стану відмови: 10 с	Кількість запитів: 100 Розмір sliding window: 20 Час очікування у стані Open (для традиційного CB): 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 % **Поріг CB**: **74 %**
2	Мінімальний час успішної відповіді: 50 мс Максимальний час відповіді: 3 с Мінімальний час перевірки стану за таймером: 1 с Максимальний час перевірки стану за таймером: 10 с Ймовірність переходу у стан відмови: 0.3 Мінімальний час відновлення зі стану відмови: 1 с Максимальний час відновлення зі стану відмови: 10 с	Кількість запитів: 100 Розмір sliding window: 20 Час очікування у стані Open (для традиційного CB): 5 с Failure rate поріг: 40 % Slow call поріг: 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: 51% **Поріг CB:** 61 %
3	Мінімальний час успішної відповіді: 50 мс Максимальний час відповіді: 3 с Мінімальний час перевірки стану за таймером: 1 с Максимальний час перевірки стану за таймером: 10 с Ймовірність переходу у стан відмови: 0.3 Мінімальний час відновлення зі стану відмови: 1 с Максимальний час відновлення зі стану відмови: 10 с	Кількість запитів: 100 Розмір sliding window: 10 Час очікування у стані Open (для традиційного CB): 5 с Failure rate поріг: 40 % Slow call поріг: 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 %** Поріг CB: 36 %

За результатами проведених експериментів було отримано такі коефіцієнти: порогове значення рейтингу: 0.4–0.45; коефіцієнт впливу метрики

failure rate: 0.4; коефіцієнт впливу метрики slow call rate: 0.15; коефіцієнт впливу метрики success call rate: 0.35; коефіцієнт впливу метрики time in open state: 0.1; максимальний час у стані Open: 10 с.

Важливою є умова переходу у стан Closed у разі перевищення максимального часу перебування у стані Open. Якщо її не дотримуватися в моделі, то цей компонент у наведеній вище загальній формулі перевищить одиницю, що негативно вплине на правильність обчислень.

ВИСНОВКИ

Досліджено сучасні практики забезпечення відмовостійкості мікросервісів. Основну увагу приділено відомому патерну Circuit Breaker, а саме спробі покращення його стандартної імплементації з погляду надійності та продуктивності у подоланні каскадних відмов. Запропоновано модифікацію Dynamic Fault Tolerance Model, яка забезпечує зменшення кількості часових затримок, що виникають під час переходу між станами, та кількості станів самого Circuit Breaker за рахунок переходу до двостанової моделі прогнозування стабільності системи на основі метрик, збір яких здійснюється під час роботи застосунку. Важливим результатом модифікації є зменшення часу очікування, за який цільовий сервіс може відновитися. Описано реалізацію вдосконаленої моделі Circuit Breaker. Наведено детальний аналіз порівняльних експериментів для перевірки правильності припущень щодо ефективності цієї моделі.

Метод реалізовано шляхом використання механізмів бібліотеки відмовостійкості Resilience4j (з розширенням її функціоналу) та мови програмування Java.

Для підтвердження правильності судження про можливість покращення патерну Circuit Breaker у цей спосіб проведено експерименти в однакових умовах для обох варіантів реалізації: традиційної (бібліотечної) та розробленої. На основі результатів порівняння пропускної здатності зовнішніх запитів до сервісу, який схильний до періодичної відмови, для обох реалізацій доведено, що за умови знаходження правильних коефіцієнтів та порогових значень рейтингу розроблена модель показує кращі результати ніж бібліотечна реалізація.

Запропоновану модель поведінки Circuit Breaker можна використати як розширення бібліотеки Resilience4j для застосунків, що працюють на JVM, або написати іншими мовами з використанням інших бібліотек чи фреймворків. Її застосування, особливо з правильним підбором параметрів, дасть змогу підвищити відмовостійкість систем з мікросервісною архітектурою, та пришвидшити їхню роботу порівняно з традиційною реалізацією.

Основною складністю впровадження запропонованої моделі є правильний підбір значень коефіцієнтів та порогового значення рейтингу. До того ж підбір параметрів досі залишається складним завданням: очевидно, що внаслідок неправильного вибору параметрів конфігурації цього механізму продуктивність системи може тільки погіршитись.

Тому одним із можливих майбутніх напрямів покращення або розвинення цього підходу до реалізації Circuit Breaker могла б бути спроба знаходження оптимальних значень усіх зазначених параметрів за допомогою алгоритмів машинного навчання.

Також до наведеної у цій роботі формули можна було б додати більше показників метрик та їхніх відповідних коефіцієнтів, проте це може ускладнити обчислення і призвести до збільшення затримок часу, що вплине на кожен запит.

СПИСОК ЛІТЕРАТУРИ

1. The state of developer ecosystem in 2022 infographic. JetBrains. 2022. URL: https://www.jetbrains.com/lp/devecosystem-2022/microservices/.
2. Nadareishvili I., Mitra R., McLarty M., Amundsen M. Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, Incorporated, 2016. 128 p. URL: https://books.google.com.ua/books/about/Microservice_Architecture.html?id=BvUEvgAACAAJ&redir_esc=y.
3. Nygard M.T. Release it! 2nd ed. Design and Deploy Production-Ready Software. Raleigh, North Carolina: The Pragmatic Bookshelf, 2014. 356 p. URL: http://repo.darmajaya.ac.id/4586/1/Release%20It%21_%20Design%20and%20Deploy%20Production-Ready%20Software%20%28%20PDFDrive%20%29.pdf.
4. Brooker M. Exponential backoff and jitter. AWS Architecture Blog. 2015. URL: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/.
5. May F. Microservices and cascading failures. Medium, 2018. URL: https://medium.com/@floyd.may/microservices-and-cascading-failures-16ec91c6ec9b.
6. Fowler M. Circuit breaker. 2014. URL: https://martinfowler.com/bliki/CircuitBreaker.html.
7. Netflix. Hystrix. 2017. URL: https://github.com/Netflix/Hystrix/wiki/.
8. Istio. Documentation. Version 1.17.2. 2023. URL: https://istio.io/latest/docs/.
9. Sentinel. Documentation. 2023. URL: https://sentinelguard.io/en-us/docs/introduction.html.
10. Montesi F., Weber J. Circuit breakers, discovery, and API gateways in microservices. arXiv:1609.05830v2 [cs.SE] 21 Sep 2016. https://doi.org/10.48550/arXiv.1609.05830.
11. Mendonça C., Aderaldo C.M., Cámara J., Garlan D. Model-based analysis of microservice resiliency patterns. *Proc. IEEE International Conference on Sogtware Architecture (ICSA 2020)* (16–20 March 2020, Salvador, Brazil). Salvador, 2020. https://doi.org/10.1109/ICSA47634.2020.00019.
12. Addeen H.H. A dynamic fault tolerance model for microservices architecture: Thesis Master of Science (MS). South Dakota State University, 2019. URL: https://openprairie.sdstate.edu/cgi/viewcontent.cgi?article=4417&context=etd.
13. Resilience4j. Introduction. 2023. URL: https://resilience4j.readme.io/docs/getting-started.

A. Hlybovets, I. Paprotskyi

INCREASING THE FAULT TOLERANCE IN MICROSERVICE ARCHITECTURE

Abstract. Microservice architecture is widely used in the development of distributed applications. Many problems present at the beginning of using this approach have already been solved. However, one of the fundamental problems that seriously affects the system's fault tolerance remains unsolved. This fundamental problem is known as guaranteed delivery of messages between services. The article analyzed standard approaches to solving this problem. Based on the analysis, the Circuit Breaker pattern was isolated, and its modification was carried out, which reduced the delay in the transition between states and, accordingly, the sending of messages by the service. The efficiency of the proposed modification has been confirmed experimentally. The result of the modification is presented in the form of a quick configuration for Spring Boot.

Keywords: microservice architecture, fault tolerance, microservice, programming pattern, Circuit Breaker.

Надійшла до редакції 16.01.2024