УДК 004.02 DOI https://doi.org/10.35546/kntu2078-4481.2024.2.15

Д. Я. БЕЙРАК
аспірант кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
ORCID: 0009-0006-5089-3603

Т. А. ВАКАЛЮК
доктор педагогічних наук, професор,
завідувач кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
ORCID: 0000-0001-6825-4697

ПІДХОДИ ДО МІЖПРОЦЕСНОЇ КОМУНІКАЦІЇ У ПОБУДОВІ МІКРОСЕРВІСНИХ СИСТЕМ В НАУКОВІЙ ЛІТЕРАТУРІ

Сьогодні продовжує актуалізовуватися питання побудови архітектури мікросервісів, що дозволяє проектувати системи з низькою зв'язністю, які мають низку переваг перед монолітами: можливість горизонтального масштабування, краще розділення системи на складові частини (сервіси), кожен окремий з яких простіше розвивати та підтримувати, можливість більш ефективного використання ресурсів тощо. Зазначені вище та низка інших причин призводять до зростання популярності такого типу архітектури в індустрії, що позначається на виборі архітекторів та інженерів програмного забезпечення стосовно впровадження мікросервісної архітектури як при побудові нових систем, так і у якості вектору розвитку успадкованих монолітних систем, які все частіше переводяться на мікросервіси. Проблематика, що стосується питань проектування мікросервісних систем містить в собі велику кількість різноманітних аспектів, і одним з таких аспектів є вибір типу міжпроцесної комунікації разом із низкою супутніх патернів, технологій та інструментів. Вплив такого вибору неможливо переоцінити: здатність сервісів ефективно обмінюватися даними є основою організації функціональності системи, що водночас має значний вплив і на такі нефункціональні характеристики, як масштабованість, надійність, складність розробки та підтримки тощо. В даній статті розглядаються методи, принципи та інструменти, призначені для організації міжпроцесної комунікації у мікросервісних системах, висвітлюються патерни, що дозволяють зменшувати вплив обмежень та недоліків тих чи інших методів та інструментів, наводяться як усталені в індустрії, так і відомі здебільшого в академічній спільноті принципи та підходи. Зазначені вище аспекти розглядаються в контексті як синхронної, так і асинхронної комунікації, в межах яких присутня власна субкласифікація за типами протоколів, інструментами, супутніми патернами, типом організації розподіленої транзакційності.

Ключові слова: *мікросервісна архітектура, міжпроцесна комунікація, моделювання, віддалений виклик процедур, обмін повідомленнями, брокер повідомлень, сага, патерни.*

D. YA. BEIRAK
Postgraduate Student at the Department of Software Engineering
Zhytomyr Polytechnic State University
ORCID: 0009-0006-5089-3603

T. A. VAKALIUK
Doctor of Pedagogical Sciences, Professor,
Head of the Department of Software Engineering
Zhytomyr Polytechnic State University
ORCID: 0000-0001-6825-4697

APPROACHES TO INTER-PROCESS COMMUNICATION IN MICROSERVICE SYSTEMS IN THE SCIENTIFIC LITERATURE

Today, the problem of designing a microservice architecture is becoming more and more relevant. Microservices, which allow building loosely coupled systems, have a number of advantages over monoliths: the possibility of horizontal scaling, better separation of the system into its constituent parts (services), each of which is easier to develop and maintain, the possibility of more efficient use of resources, etc. These and a number of other reasons lead to the growing popularity of this type of architecture in the industry, which influences the choice of architects and software engineers regarding the implementation of microservice architecture both when building new systems and as a vector of development of legacy monolithic systems, which are increasingly transferred to microservices. The issues related to the design of microservice systems contain a large number of different aspects, and one of them is the choice of the type of inter-process communication along with a number of accompanying patterns, technologies, and tools. The impact of such

a choice cannot be overstated: the ability of services to exchange data effectively forms the foundation of business logic organization, which at the same time has a significant impact on such non-functional characteristics as scalability, reliability, and complexity of development and maintenance. This article examines the methods, principles, and tools intended for the organization of inter-process communication in microservice systems, highlights the patterns that allow of reducing the influence of limitations and shortcomings of certain methods and tools, provides principles and approaches both established in the industry and known mostly in the academia. The abovementioned aspects are considered in the context of both synchronous and asynchronous communication, within which there are their own subclasses of the types of protocols, tools, accompanying patterns, and distributed transactions.

Key words: *microservice architecture, inter-process communication, modeling, remote procedure invocation, remote procedure call, messaging, message broker, saga, patterns.*

Постановка проблеми

На сьогоднішній день все більше нових проектів створюється на основі мікросервісної архітектури, а також все більше успадкованих систем переводяться на даний тип архітектури. Серйозного значення набуває питання організації міжпроцесної комунікації, тобто комунікації між окремими сервісами, від якого в значній мірі залежить архітектура системи, її продуктивність, надійність, масштабованість, а також низка інших нефункціональних параметрів, таких як затримка, пропускна здатність, складність розробки та підтримки, можливості моніторингу та тестування тощо. Дані фактори роблять актуальними дослідження наявних підходів та інструментів реалізації міжпроцесної комунікації у контексті побудови мікросервісної архітектури.

Аналіз останніх досліджень і публікацій

Питання міжпроцесної комунікації у контексті мікросервісної архітектури розглядались в науковій літературі багатьма вченими. Загальні проблеми, а також велику кількість вузьких питань розглядали Кріс Річардсон (Chris Richardson) [1], Сем Ньюман (Sam Newman) [2], Ендрю Таненбаум (Andrew Tanenbaum), Мартен ван Стін (Maarten van Steen) [3], Мартін Клепман (Martin Kleppmann) [4]. Питання, пов'язані з синхронною комунікацією розглядали Рой Філдінг (Roy T. Fielding) [5], Мартін Фаулер (Martin Fowler) [6], Карл Мастранджело (Carl Mastrangelo) [7], Хуан Крус Віотті (Juan Cruz Viotti), Мітал Кіндерхедія (Mital Kinderkhedia) [8], Лей Чжан (Lei Zhang) [9] та інші. Питання, пов'язані з усуненням перебоїв при синхронній комунікації розглядали Фалаха (Falahah), Кріданто Сурендро (Kridanto Surendro), Вікан Данар Саніндйо (Wikan Danar Sunindyo) [10], Набор К. Мендонка (Nabor C. Mendonça) [11], Мухаммед Міраж (Muhammad Miraj), Ахмад Нурул Фаджар (Ahmad Nurul Fajar) [12] та інші. Патерни, пов'язані з виявленням сервісів розглядали Баасанджаргал Ерденебат (Baasanjargal Erdenebat), Тамаш Козшік (Tamás Kozsik), Баяржаргал Бад (Bayarjargal Bud) [13], Фікрет Сіврикая (Fikret Sivrikaya) та інші. Питання, пов'язані із асинхронною комунікацією розглядали Джон Мур (John Moore) [14], Омкар Касарлевар (Onkar Kasarlewar), П. С. Десаї (P. S. Desai) [15], Рупалі Арун Джайрандж (Rupali Arun Jairange), А. К. Гупта (A. K. Gupta) [16], Філіп Молл (Philipp Moll) [17], Номаун Ратор (Nomaun Rathore), Шрі Кант (Shri Kant) [18] та інші. Роботи, присвячені порівняльному аналізу брокерів повідомлень, виконували Рокін Махарджан (Rokin Maharjan) [19], Гуо Фу (Guo Fu), Янфен Чжан (Yanfeng Zhang), Дже Ю (Ge Yu) [20], Ранджит Хегд (Ranjith Hegde) [21], Шоукат Хоссейн Чай (Showkat Hossain Chy) [22] та інші. Питання, пов'язані з сагами, розглядали Репана Редді Секхар (Repana Reddy Sekhar), Віна Гадад (Veena Gadad) [23], Мустафа Гордслі (Mustafa Gordesli), Ахад Насаб (Ahad Nasab), Асаф Варол (Asaf Varol) [24], Еман Дарагмі (Eman Daraghmi), Ченг-Пу Чжан (Cheng-Pu Zhang), Шиян-Мінг Юань (Shyan-Ming Yuan) [25], Каролін Дюрр (Karolin Dürr), Робін Ліхтенталер (Robin Lichtenthäler), Гвідо Вірц (Guido Wirtz) [26], Мартін Стефанко (Martin Štefanko), Ондрей Чалупка (Ondřej Chaloupka), Бруно Россі (Bruno Rossi) [27] та інші.

Формулювання мети дослідження

Метою статті є детальний огляд інструментів та підходів до реалізації міжпроцесної комунікації у контексті побудови мікросервісної архітектури.

Викладення основного матеріалу дослідження

Мікросервісні системи є розподіленими за своєю природою, а тому взаємодія між їх складовими частинами (сервісами) відбувається через мережу. Цей факт обумовлює виникнення двох типів комунікації: синхронної та асинхронної. Використання кожного з них тягне за собою застосування найбільш прийнятних патернів, що дозволяють зменшити вплив негативних факторів, які має кожен із даних підходів. Варто також зауважити, що інженери та науковці здебільшого надають перевагу асинхронному підходу, так як він дозволяє досягнути найнижчої зв'язності частин мікросервісної системи, хоча й є архітектурно складнішим, ніж синхронний.

Синхронна комунікація реалізується за допомогою патерну, що має назву «віддалений виклик процедур» (remote procedure invocation, RPI; remote procedure call, PRC). Клієнт, щоб викликати певний функціонал на серверній стороні, повинен зробити мережевий запит та дочекатися відповіді. Хоча очікування відповіді на стороні клієнта може бути як блокуючим, так і неблокуючим, незалежно від цього такий спосіб комунікації є саме синхронним, бо клієнт активно очікує на відповідь [1]. З технічної точки зору, віддалені виклики можуть бути реалізовані суто з використанням протоколу HTTP, проте існує ряд технологій, які дозволяють робити це більш ефективно.

На сьогоднішній день, однією із найпопулярніших технологій реалізації міжпроцесної комунікації залишається REST (representational state transfer – передача репрезентативного стану). В основі даної концепції лежить поняття ресурсу, що зазвичай представляє собою певний об'єкт предметної області, яким можна керувати за допомогою API. Найчастіше така комунікація відбувається через протокол HTTP з використанням його дієслів для диференціації операцій, а у якості відповіді надходить об'єкт JSON або XML, хоча можуть бути застосовані і інші, в тому числі, бінарні формати [1]. На сьогодні запропоновано багато варіацій REST, а також існує багато поглядів та різночитань щодо найбільш правильного визначення даного архітектурного стилю [5]. Леонард Річардсон (Leonard Richardson) запропонував модель зрілості REST [6], яка має важливе значення з точки зору практичного застосування. Дана модель складається з чотирьох рівнів: рівень 0 – клієнти викликають сервіс за допомогою HTTP-запитів POST, а кожен запит вказує на дію, яка має бути виконана над певною зазначеною ціллю (об'єктом, сутністю) з урахуванням певних параметрів; рівень 1 – сервіс реалізує підтримку ресурсів, а клієнт, виконуючи запит POST, вказує лише виконувану дію та параметри; рівень 2 – сервіс використовує методи HTTP для здійснення операцій (GET для отримання, POST для створення, PUT для оновлення, PATCH для часткового оновлення, DELETE для видалення), при цьому параметри передаються через рядок запиту або у тілі запиту; рівень 3 – організація API будується навколо принципу HATEOAS (hypertext as the engine of application state – гіпермедіа як рушій стану додатку), основна ідея якого полягає в тому, що запит GET повертає посилання з операціями, які можуть бути виконані над даним ресурсом [1]. У якості набору правил закріпилося також поняття RESTful-архітектури (тобто архітектури, яка повністю відповідає ідеї REST), які запропонував Чезаре Паутассо (Cesare Pautasso) [3]. До даних правил входять наступні: 1) ресурси повинні ідентифікуватися за допомогою єдиної схеми найменування; 2) усі сервіси мають однаковий інтерфейс, що складається з не більше ніж чотирьох операцій (GET, POST, PUT та DELETE); 3) повідомлення, що надсилаються сервісам або сервісами мають бути повністю самоописаними; 4) після виконання операцій, сервіс забуває про того, хто його викликав.

Альтернативою REST є протокол gRPC, що будується на основі HTTP/2. На відміну від REST, цей протокол є бінарним, підтримує потокову передачу даних та не є обмеженим з позиції кількості доступних HTTP-дієслів. У якості формату повідомлень в gRPC зазвичай використовується мова опису інтерфейсів Protocol Buffers. Робота з Protocol Buffers відбувається таким чином, що на перший план постає опис API, на основі якого генеруються моделі цільовою мовою програмування для серверної та клієнтської частин. Не дивлячись на те, що gRPC підтримує двонаправлену потокову передачу даних, цей протокол залишається синхронним, а тому, як і у випадку REST, такий тип комунікації передбачає використання допоміжних патернів, що дозволяють обробляти випадки часткової невдачі при передачі даних (partial failure), а також вирішувати проблеми виявлення одних сервісів іншими [1].

Протокол gRPC також підтримує інші варіанти серіалізації: протоколи Apache Thrift, Apache Avro, Flatbuffers, Cap'n Proto, а також JSON, або відсутність серіалізації взагалі [7]. Формати Thrift та Protocol Buffers мають схожу внутрішню структуру, хоча, у порівнянні з Thrift, Protocol Buffers не має окремого типу даних для позначення масиву (це виконується за допомогою маркеру «repeated»). Особливістю Apache Avro, у порівнянні з Protocol Buffers та Apache Thrift, є відсутність нумерації полів, що дозволяє використовувати цей протокол серіалізації для генерування динамічних схем [4]. До переваг Flatbuffers та Cap'n Proto відносяться ефективний процес десеріалізації та читання даних, а також невеликий розмір самої бібліотеки з мінімальною кількістю залежностей [8].

Група науковців (Лей Чжан (Lei Zhang) та ін.) запропонувала механізм комунікації, в основі якого лежить поєднання звичайного RPC з Protocol Buffers та неблокуючою моделлю комунікації – RPCX [9]. В ньому використовуються технології динамічного проксування віддаленого сервісу та конфігурації анотацій, що спрощує API та робить його більш захищеним. Мережева модель комунікації заснована на асинхронній архітектурі фреймворку Netty, а в якості формату передачі даних використовується Protocol Buffers. Такий підхід продемонстрував кращі результати у порівнянні з протоколом gRPC за такими параметрами, як середній час виконання коду та кількість транзакцій в секунду [9].

У розподілених системах також розповсюдженою є проблема коротких мережевих перебоїв, коли недоступний при певному запиті сервіс може стати доступним при наступному зверненні. Для адекватного вирішення таких ситуацій використовують патерн «retryer», який дозволяє надсилати певну кількість повторних запитів з перервами, розраховуючи на те, що проблема, що унеможливила доступ до сервісу, не є систематичною та зникне, якщо повторити запит через невеликий час [2]. Проте, даний патерн не вирішує проблеми в тому випадку, коли низхідний ресурс є недоступним протягом значної кількості часу. У клієнта даного ресурсу накопичуються власні запити, які призводять до неможливості їх виконання, так як потребують звернення до ресурсу, що не відповідає. В такому випадку можливе виникнення каскадного збою, коли непрацездатність одного сервісу спричиняє відмову всієї системи, або значної її частини. Для запобігання такого сценарію застосовується патерн «автоматичний вимикач» (circuit breaker), що спрацьовує після певного періоду часу, який характеризує системну відмову в роботі низхідного ресурсу, та відразу надсилає клієнтам відповідь про помилку впродовж деякого часу [1, 2]. Варто зауважити, що існують варіації реалізацій «автоматичного вимикача»: на клієнтській стороні, на серверній стороні, та на стороні проксі (тобто такий, що розгортається окремо) [10].

Хоча розглянуті патерни дуже часто зустрічаються в науковій літературі, існують і інші варіанти забезпечення стійкості до мережевих відмов. Так, патерн «timeout» дозволяє реалізувати механізм відмови очікування будь-якої відповіді взагалі, якщо час очікування перевищує задане значення. Патерн «fallback» дозволяє повернути клієнту запасний, заздалегідь підготовлений, результат, якщо отримати справжній не вдалося. Патерн «hedging» дозволяє виконати деякі додаткові дії, якщо виконання основного запиту займає більший час, ніж очікується: наприклад, запустити ідентичний паралельний запит, виходячи з розрахунку, що один з них виконається раніше, а інший можна буде відкинути [28]. Патерн «bulkhead» дозволяє виділяти частину потоків певного мікросервісу для конкретних «важких» операцій, тим самим забезпечуючи працездатність системи для більш «легких» у випадку, коли на «важкі» прийдеться неочікувано велика кількість запитів [2]. Також широкого вжитку мають такі патерни як «throttling» та «обмеження швидкості» (rate limiting), що обмежують кількість запитів до певного сервісу [29].

Існують дослідження (Набор Мендонка (Nabor C. Mendonça) та ін. [11], Мухаммед Міраж (Muhammad Miraj), Ахмад Нурул Фаджар (Ahmad Nurul Fajar) [12]), які показують, що правильно застосовані патерни, що забезпечують стійкість до відмов, істотно зменшують змагання за володіння ресурсами на клієнтській стороні у порівнянні з більш примітивною реалізацією повторення запитів, а також зменшують час відповіді.

При синхронній комунікації у мікросервісних додатках важливу роль відіграють патерни, пов'язані з виявленням сервісів. Так як дуже незручно, навіть використовуючи протокол DNS, статично задавати або визначати поточні IP-адреси екземплярів сервісів, враховуючи розгортання у хмарному середовищі, необхідність масштабування тощо, використовуються спеціальні механізми динамічного виявлення сервісів. Один із способів це зробити полягає у самореєстрації кожного сервісу на початку його роботи у спеціальному централізованому «реєстрі сервісів» (service registry). Клієнти, перед тим, як надіслати запит тому чи іншому сервісу, роблять запит до реєстру, щоб отримати список екземплярів сервісів, та обирають один з екземплярів за допомогою певного алгоритму балансування навантаження. Такий підхід поєднує в собі, відповідно, патерни «самореєстрація» (self registration) та «виявлення на клієнтській стороні» (client-side discovery).

Інший підхід передбачає реєстрацію сервісів при їх розгортанні на стороні хмарного провайдера реєстратором (registrar), який є частиною інфраструктури. Сервіси-клієнти, при цьому, відразу роблять запити використовуючи DNS-ім'я, яке розв'язується маршрутизатором запитів, який, в свою чергу, опитує реєстр сервісів та виконує балансування навантаження. Відповідно, такий підхід поєднує патерни «реєстрація третьої сторони» (3rd party registration) та «виявлення на серверній стороні» (server-side discovery) [1].

На сьогоднішній день, в цій сфері науковці працюють над такими задачами, як перехід від виявлення сервісів в архітектурі, побудованій з використанням віртуальних машин, до архітектури, побудованої навколо контейнерів (Баасанджаргал Ерденебат (Baasanjargal Erdenebat), Баяржаргал Бад (Bayarjargal Bud) та Тамаш Козшік (Tamás Kozsik) [15]), виявлення сервісів в контексті архітектури IoT (Internet of things – інтернет речей) (Фікрет Сіврикая (Fikret Sivrikaya) та ін. [30]) тощо.

Асинхронна комунікація реалізується у мікросервісних системах за допомогою механізму передачі повідомлень між сервісами. Зазвичай, такі системи використовують брокер повідомлень, який виступає посередником при передачі повідомлення від одного сервісу до іншого, проте існують і варіанти безброкерних архітектур. Концепція повідомлення передбачає наявність заголовку та тіла, а також середовища передачі – каналу, що являє собою абстракцію над інфраструктурою доставки повідомлень. Розрізняють декілька типів повідомлень: документ – загальний тип повідомлень, які приймальна сторона інтерпретує на власний розсуд; команда – еквівалент RPC-запиту, що вказує операцію, яку необхідно виконати, та її параметри; подія – повідомлення, що сповіщає зацікавлені сторони про те, що відбулося дещо визначне (зазвичай це доменні події, які характеризують зміну стану об'єкту предметної області). Щодо каналів, вони поділяються на канали типу «точка-точка» (point-to-point), який доставляє повідомлення до виключно одного споживача, та канали типу «публікація-підписка» (publish-subscribe), які доставляють повідомлення до всіх споживачів, що прослуховують даний канал [1].

Архітектура, що базується на обміні повідомленнями, відкриває можливість асинхронної комунікації, хоча і дає змогу за потреби реалізувати синхронну. Вона дозволяє імплементувати такі способи комунікації, як «асинхронні запит/відповідь» без необхідності блокування клієнта під час очікування відповіді, «односпрямовану комунікацію», що дозволяє надіслати повідомлення одній стороні (тип каналу «точка-точка»), «публікацію-підписку», що дозволяє доставляти повідомлення багатьом споживачам, «публікацію-асинхронну відповідь», що є поєднанням «публікації-підписки» з «асинхронними запитом/відповіддю», а також «pipeline», що дозволяє доставляти повідомлення одному із багатьох потенційних споживачів [1, 3].

Безброкерний варіант асинхронної комунікації дозволяє сервісам виконувати обмін повідомленнями напряму. Однією з популярних технологій такого обміну є ZeroMQ, що водночас є і специфікацією, і бібліотекою, побудованою поверх протоколу TCP [1, 3], на основі якої, в тому числі, створено нові технології, такі як Zest, що використовується для абстрагування ідей протоколу CoAP (Constrained Application Protocol – протокол обмежених додатків), що застосовується в контексті IoT (Джон Мур (John Moore) та ін. [14]). Інші розробки в цьому напрямку стосуються підвищення захищеності безброкерної комунікації типу «публікація-підписка».

Так, в роботі Омкара Касарлевара (Onkar Kasarlewar) та П. С. Десаі (P. S. Desai) [18] був запропонований підхід на основі криптографії на еліптичних кривих та сервера ключів, що збирає їх від видавця та підписника. В роботі науковців Рупалі Арун Джайрандж (Rupali Arun Jairange) та А. К. Гупта (A. K. Gupta) [16] запропоновано варіант з використанням алгоритму шифрування зворотного кола. Також існують розробки даної проблематики в контексті NDN-мереж (Named Data Networking – мережі іменованих даних) (Філіп Молл (Philipp Moll) та ін. [17]) та в контексті блокчейну (Номаун Ратор (Nomaun Rathore) та Шрі Кант (Shri Kant) [21]).

Хоча безброкерна архітектура має свої переваги, такі як менші витрати мережевого трафіку, виключення можливості єдиної точки збою або вузького місця та менша операційна складність, її недоліком є необхідність застосування механізмів виявлення сервісів (як при синхронній архітектурі), зниження доступності системи через те, що як передавач, так і приймач повідомлень мають бути активними в даний момент часу, більша складність в реалізації механізмів гарантованої доставки тощо. Через це широкого використання набули системи, побудовані на основі передачі повідомлень через той чи інший брокер повідомлень. У якості переваг такої комунікації виділяють низьку зв'язність частин системи, гнучкість у виборі певного стилю комунікації та явну для розробника відмінність (яка не створює помилкове відчуття безпеки) від абстрагованого стилю RPC-комунікації, що здебільшого приховує той факт, що виклик відбувається через мережу. У якості недоліків комунікації за допомогою брокера зазначаються додаткові операційні витрати та потенційна наявність єдиної точки відмови або вузького місця [1].

На сьогоднішній день існує велика кількість брокерів повідомлень з низкою акцентів на різні архітектурні та інші характеристики. Серед найбільш розповсюджених можна виділити RabbitMQ (який є імплементацією протоколу AMQP), Apache Kafka, Apache RocketMQ, ActiveMQ Artemis, Apache Pulsar, а також, як варіант реалізації брокеру, розглядають відповідні можливості Redis. Крім того, існують хмарні рішення, такі як Amazon SQS, Amazon EventBridge, Azure Storage Queues та Azure Service Bus.

У роботі Рокін Махарджана (Rokin Maharjan) та ін. [19] проведено аналіз чотирьох рішень брокерів повідомлень з наступними результатами. Apache Kafka показав найкращу пропускну здатність (throughput). Redis продемонстрував найнижчий показник затримки (latency). За необхідності забезпечення низької затримки, у випадках, коли можна знехтувати пропускною здатністю, варто звернути увагу на ActiveMQ Artemis або RabbitMQ. В ситуаціях, коли важлива як низька затримка, так і висока пропускна здатність, варто звернути увагу на Apache Kafka, так як за показником затримки цей брокер наближається до Redis [19].

Враховуючи результати порівняльних досліджень інших вчених (Гуо Фу (Guo Fu) та ін. [20], Ранджит Г. Хегде (Ranjith G. Hegde), Нагараджа Г. С. (Nagaraja G. S.) [21], Шоукат Хоссейн Чай (Showkat Hossain Chy) та ін. [22]), можна резюмувати наступне. Apache Kafka, маючи високу пропускну здатність завдяки технології zero-copy, добре підходить для систем реального часу, обробки великих даних (big data), потокової передачі даних, передачі відеоданих, широкомасштабних банківських систем. Проте, так як Apache Kafka завжди використовує запис на диск журналів реального часу та сторінок, зі збільшенням розміру повідомлення в ньому зростає затримка, а зі збільшенням тем (topic) та розділів (partition) ще й споживання ресурсів. RabbitMQ вважається брокером повідомлень загального вжитку. Не будучи таким швидким як Apache Kafka, він має дуже широкий функціонал (пріорітетні черги, черги недоставлених повідомлень (dead-letter queue), час життя повідомлень та черг тощо), який може бути розширеним за допомогою плагінів, а також застосовується у сферах, критичних до відсутності втрати даних. Брокер повідомлень на основі Redis можна використовувати в тих випадках, коли стовідсотковою гарантією доставки можна знехтувати на користь швидкості, тому таке рішення також підходить для сервісів передачі відео, наповнення стрічок соціальних мереж тощо. Також Redis підходить для вбудованих (embedded) систем, які можуть передавати великі об'єми дрібних порцій даних.

Перевагою RocketMQ перед Apache Kafka є те, що зі збільшенням тем не знижується продуктивність його роботи, проте RocketMQ не підходить для сценаріїв обробки великих за об'ємом повідомлень з високою швидкістю надходження без більш тонкого налаштування та використання таких засобів як зворотний тиск (backpressure). ActiveMQ Artemis демонструє найнижчий показник затримки у сценаріях з невеликим навантаженням, проте є вимогливим до ресурсів. Apache Pulsar є найбільш збалансованим інструментом з точки зору динаміки споживання ресурсів (як процесорних потужностей, так і оперативної пам'яті), але платою за це є можливий у ряді сценаріїв використання компроміс за продуктивністю на великих повідомленнях, що пояснюється активною роботою garbage collector, а також підвищена затримка, яка, як і у випадку Apache Kafka, пояснюється постійним записом журналів реального часу та сторінок на диск.

Наступна серія патернів, що розглядається в контексті комунікації в розподілених системах з асинхронною архітектурою, стосується поняття транзакційності. Деякі автори [1, 2] заперечують проти використання механізму розподілених транзакцій, тому що вони здатні принести у проект більше проблем ніж користі, так як їх використання змушує блокувати дані кожного сервісу на час, поки триває транзакція, який є доволі довгим у розподілених системах, а тому це відразу знижує швидкість роботи та доступність системи. Крім того, вони часто не підтримуються сучасними інструментами (сховищами NoSQL та брокерами повідомлень). Проте питання консистентності для операцій, що охоплюють декілька сервісів одночасно все одно має бути певним чином вирішено, тому замість розподілених транзакцій використовують патерн «сага» (saga).

Сага – це набір локальних транзакцій, що разом утворюють велику розподілену транзакцію, та запобігають довготривалим блокуванням даних, яке притаманне розподіленим транзакціям. Також використання саг змушує архітекторів та розробників у явному вигляді моделювати бізнес-процеси, що є їх додатковою перевагою. Питання, що викликає певні труднощі під час використання саг стосується ізольованості транзакцій. Хоча локальні транзакції цілком можуть бути ACID-транзакціями, вся сага є лише ACD-транзакцією. Це проявляється в тому, що у випадках, коли під час виконання локальної транзакції стається помилка, сага має застосовувати компенсаційні транзакції (compensating transactions), що дозволяють відмінити попередні зміни кожної окремої локальної транзакції. Крім того, локальні транзакції саги мають певним чином координуватися, для чого використовується один з двох механізмів: хореографія (choreography) або оркестрація (orchestration) [1, 2].

Використання хореографії передбачає відсутність координатора, який керував би процесом: учасники саги виконують обмін повідомленнями, підписуються на необхідні та певним чином на них реагують. Фактично, це зводиться до приймання сервісом деякого повідомлення, виконання необхідних операцій, оновлення даних цього сервісу та відправки наступного повідомлення, яке можуть підхопити інші сервіси [1, 2]. Через таку конфігурацію обміну повідомленнями, у літературі зустрічається опис саг, що базуються на хореографії, як таких, що працюють за принципом «довіряй, але перевіряй» (trust-but-verify) [2].

Так як для хореографічних саг важливим є атомарне оновлення бази даних та публікація подій, вони мають використовувати механізми транзакційної відправки повідомлень (transactional messaging), до яких відносяться «публікація подій» (transactional outbox), «видавець опитувань» (polling publisher) та «відслідковування транзакційного журналу» (transaction log tailing). Патерн «публікація подій» передбачає використання бази даних, що підтримує повноцінні ACID-транзакції у якості тимчасової черги повідомлень. Для кожної сутності у базі даних створюється поле, куди потрапляють усі повідомлення, що чекають публікації. Після цього виникає питання вибирання та відправлення даних повідомлень їх споживачам. «Видавець опитувань» є проміжною ланкою між базою даних з повідомленнями та брокером повідомлень, та працює методом опитування (polling): він виконує запит щодо відповідних полів, знаходить повідомлення, які чекають відправки, та видаляє їх після того, як вони були успішно надіслані [1]. Даний підхід є простим у реалізації та добре працює в системах, що не потребують великої масштабованості. З іншого боку, є важчим тестування функціональності, що задіяна в хореографії [1, 23]. Ще одним негативним фактором є потенційна необхідність частого опитування бази даних, яке може виявитися ресурсозатратним. Більш складним, але ефективним рішенням в даному випадку, може виявитися відслідковування журналу транзакцій бази даних, що реалізується відповідним патерном: аналізатор журналу транзакцій читає журнал транзакцій бази даних, аналізує його та перетворює релевантні записи на повідомлення, які публікує у брокер повідомлень [1].

Використання оркестрації передбачає наявність оркестратору – класу, що керує сагою шляхом контролювання процесу її проходження. Сага складається з необхідної кількості кроків, на кожному з яких оркестратор надсилає команду відповідному сервісу, який потім асинхронним чином надсилає відповідь. Базуючись на відповіді, оркестратор приймає рішення про виконання подальших операцій [1, 2]. Така конфігурація обміну повідомленнями характеризується принципом «командуй та контролюй» (command-and-control) [2]. У якості апарату моделювання процесу оркестрації використовують кінцеві автомати, які також можуть бути ефективно протестовані [1].

Оркестровані саги мають перед хореографічними ряд переваг: нижча зв'язність через те, що оркестратор бере на себе частину знань щодо форматів повідомлень, які очікують сервіси (проте з точки зору сервісів системи в цілому, підхід з використанням хореографії не має залежності на оркестратор, що, з цієї позиції, навпаки додає зв'язності оркестрації); покращення розділення обов'язків та спрощення бізнес-логіки завдяки локалізації координуючої логіки в оркестраторі; відсутність можливості виникнення циклічних залежностей. До недоліків можна віднести надмірну централізацію процесу виконання саги, що може стати причиною концентрації значної частини бізнес-логіки в оркестраторі, а також потенційно більшу інфраструктурну складність через наявність додатково сервісу [1, 23]. Однією із рекомендацій, яку можна знайти в науковій літературі є комбінований підхід: написання саг з оркестраторами для складніших випадків, та використання хореографії у простіших сагах [1, 2]. У якості способу позбавлення надмірної централізації рекомендується застосування в ролі оркестратора різних сервісів для різних функціональних сценаріїв [2].

Через те, що транзакційна модель саги, через її розподілену природу, позбавлена ізоляції, перебої у виконанні саги можуть призводити до виникнення аномалій. До них відносяться:

- втрачені оновлення (lost updates) – одна сага перезаписує зміни іншої, не вичитуючи їх при цьому;
- dirty reads – сага читає незавершені оновлення іншої саги;
- нечітке читання (fuzzy reads) – два різних кроки саги читають однакові дані, але отримують різні результати, тому що якась інша сага внесла в них зміни [1].

Задля запобігання або мінімізації зазначених небажаних ефектів, використовують контрзаходи, які можуть виконуватися на певних кроках саги у якості компенсаційних транзакцій, що відміняють попередні локальні транзакції. До контрзаходів відносяться:

• семантичне блокування (semantic lock) – блокування на рівні додатку за допомогою «прапорців», які проставляють у відповідних записах під час створення чи оновлення;

• комутативні оновлення (commutative updates) – проектування операцій оновлення так, щоб їх можна було виконувати у будь-якому порядку;

• pessimistic view – реорганізація кроків саги таким чином, щоб зменшити ризики dirty reads;

• повторне читання (reread value) – впровадження повторного читання записів бази даних перед оновленням для запобігання втрачених оновлень;

• файл версій (version file) – спосіб перетворення оновлень на комутативні: операції над записами у базі даних окремо фіксуються задля того, щоб їх можна було переставити місцями;

• за значенням (by value) – вибір механізму виконання операції (сага або розподілена транзакція) в залежності від контексту запиту, бізнес-ризиків тощо [1].

Часто при реалізації саг використовується відразу декілька контрзаходів, які доповнюють один одного [1].

Науковці Мустафа Гордслі (Mustafa Gördesli), Ахад Насаб (Ahad Nasab) та Асаф Варол (Asaf Varol) пропонують у своїй роботі варіацію реалізації механізму відміни локальних транзакцій саги з використанням сервісу контролю відповідей (response control service) [24]. На відміну від хореографічного підходу, в якому сервіси самі відповідають за відміну власних локальних транзакцій при отриманні відповідного повідомлення, та оркестраційного підходу, де цим процесом керує оркестратор, автори пропонують винести цей функціонал в окремий сервіс. Цей сервіс підписується на всі повідомлення всіх мікросервісів системи та надсилає результуючі відповіді у разі успіху всіх етапів саги; якщо ж під час її виконання виникає помилка, він відповідає за запуск дій, що скасовують зміни на стороні всіх сервісів, які цього потребують, а клієнту відправляє повідомлення з помилкою. Керування транзакціями відбувається за рахунок їх унікальної ідентифікації [24].

Інший варіант боротьби з наслідками відсутності ізоляції транзакцій запропонували Еман Дарагмі (Eman Daraghmi), Ченг-Пу Жанг (Cheng-Pu Zhang) та Шуан-Мінг Юань (Shyan-Ming Yuan) [25]. Вони запропонували застосування кешування даних для усунення проблем з ізоляцією читання даних та уникнення накладних витрат на відміну саги: CRUD-операції спершу виконуються через кеш, а лише потім, після остаточного валідування, фіксуються в локальних базах даних; компенсуючі транзакції, відповідно, теж будуть виконуватися лише на рівні кешу. Остаточна фіксація даних відбувається завдяки сервісу синхронної кінцевої фіксації (eventual commit sync service), який і запускає процес запису в локальні бази даних, коли всі етапи саги пройшли успішно [25].

Наостанок варто зауважити, що вибір одного з наявних фреймворків для реалізації саг може виявитися нетривіальним. Так, окрім аналізу таких показників, як підтримувані мови програмування, продуктивність та зручність синтаксису, варто врахувати, що не всі фреймворки підтримують реалізацію обох типів саг (наприклад, Netflix Conductor, Camunda та MicroProfile LRA не підтримують хореографічні саги [26]), а інші можуть накладати специфічні умови використання (наприклад, Axon та Eventuate ES вимагають в системі реалізацію патерну CQRS (command and query responsibility segregation – розділення відповідальності командних запитів) [27]).

Висновки

Проаналізувавши наявні на сьогоднішній день підходи до реалізації міжпроцесної комунікації у мікросервісній архітектурі, можна відзначити, що і індустрії існує широкий вибір інструментарію та підходів, які стосуються цієї тематики. Не дивлячись на велику кількість усталених підходів, академічна спільнота працює над їх удосконаленням та розвитком, що може виявитися перспективним у майбутньому. Виконуються також дослідження, пов’язані з застосуванням інструментів у різних сценаріях використання, що дозволяє робити більш точний вибір для розв’язання конкретних задач, а також надає зворотний зв’язок розробникам. Таким чином, до перспектив подальших досліджень відносимо подальшу апробацію розглянутих підходів та інструментів, виявлення в них вузьких місць та обмежень, а також отримання зворотного зв’язку від архітекторів та інженерів.

Список використаної літератури

1. Richardson C. Microservices patterns: with examples in Java. Shelter Island, New York: Manning Publications, 2019. 490 p.

2. Newman S. Building microservices: designing fine-grained systems. Second Edition. Beijing: O’Reilly Media, 2021. 586 p.

3. Steen M. van, Tanenbaum A.S. Distributed systems. 4th edition, Version 01 (January 2023). Maarten van Steen, 2023. 669 p.

4. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 611 p.

5. Fielding R.T. et al. Reflections on the REST architectural style and “principled design of the modern web architecture” (impact paper award) // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn Germany: ACM, 2017. P. 4–14.

6. Fowler M. Richardson Maturity Model [Electronic resource]. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html.

7. Mastrangelo C. gRPC + JSON [Electronic resource]. 2018. URL: https://grpc.io/blog/grpc-with-json/.

8. Viotti J.C., Kinderkhedia M. A Survey of JSON-compatible Binary Serialization Specifications: arXiv:2201.02089. arXiv, 2022.

9. Zhang L. et al. High performance microservice communication technology based on modified remote procedure call // Sci Rep. 2023. Vol. 13, № 1. P. 12141.

10. Falahah, Surendro K., Sunindyo W.D. Circuit Breaker in Microservices: State of the Art and Future Prospects // IOP Conf. Ser.: Mater. Sci. Eng. 2021. Vol. 1077, № 1. P. 012065.

11. Mendonca N.C. et al. Model-Based Analysis of Microservice Resiliency Patterns // 2020 IEEE International Conference on Software Architecture (ICSA). Salvador, Brazil: IEEE, 2020. P. 114–124.

12. Miraj M., Fajar A.N. Model-Based Resilience Pattern Analysis For Fault Tolerance In Reactive Microservice. Vol. 2022. № 9.

13. Erdenebat B., Bud B., Kozsik T. Challenges in service discovery for microservices deployed in a Kubernetes cluster – a case study // Infocommunications journal. 2023. Vol. 15, № Special Issue. P. 69–75.

14. Moore J. et al. Zest: REST over ZeroMQ: arXiv:1902.07009. arXiv, 2019.

15. Kasarlewar O., Desai P.S. Secure Broker-less Publish/Subscribe System using ECC. 2015. Vol. 6.

16. Jairange R.A., Gupta A.K. Secure Brokerless System for Publisher/Subscriber Relationship in Distributed Network. 2016. Vol. 6, № 7.

17. Moll P. et al. Resilient Brokerless Publish-Subscribe over NDN // MILCOM 2021–2021 IEEE Military Communications Conference (MILCOM). San Diego, CA, USA: IEEE, 2021. P. 438–444.

18. Rathore N., Kant S. Enhanced Blockchain Application for Pub-Sub Model // Advances in Electromechanical Technologies / ed. Pandey V.C., Pandey P.M., Garg S.K. Singapore: Springer Singapore, 2021. P. 299–311.

19. Maharjan R. et al. Benchmarking Message Queues // Telecom. 2023. Vol. 4, № 2. P. 298–312.

20. Fu G., Zhang Y., Yu G. A Fair Comparison of Message Queuing Systems // IEEE Access. 2021. Vol. 9. P. 421–432.

21. Hegde R.G. Low Latency Message Brokers. 2020. Vol. 07, № 05.

22. Chy M.S.H. et al. Comparative Evaluation of Java Virtual Machine-Based Message Queue Services: A Study on Kafka, Artemis, Pulsar, and RocketMQ // Electronics. 2023. Vol. 12, № 23. P. 4792.

23. Sekhar R.R., Gadad V. Microservices, Saga Pattern and Event Sourcing: A Survey. 2020. Vol. 07, № 05.

24. Gordesli M., Nasab A., Varol A. Handling Rollbacks with Separated Response Control Service for Microservice Architecture // 2022 3rd International Informatics and Software Engineering Conference (IISEC). Ankara, Turkey: IEEE, 2022. P. 1–4.

25. Daraghmi E., Zhang C.-P., Yuan S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture // Applied Sciences. 2022. Vol. 12, № 12. P. 6242.

26. Dürr K., Lichtenthäler R., Wirtz G. Saga Pattern Technologies: A Criteria-based Evaluation: // Proceedings of the 12th International Conference on Cloud Computing and Services Science. Science and Technology Publications, 2022. P. 141–148.

27. Štefanko M., Chaloupka O., Rossi B. The Saga Pattern in a Reactive Microservices Environment: // Proceedings of the 14th International Conference on Software Technologies. Prague, Czech Republic: SCITEPRESS – Science and Technology Publications, 2019. P. 483–490.

28. Resilience strategies [Electronic resource]. URL: https://www.pollydocs.org/strategies/index.html.

29. Cloud Design Patterns [Electronic resource]. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/.

30. Sivrikaya F. et al. Internet of Smart City Objects: A Distributed Framework for Service Discovery and Composition // IEEE Access. 2019. Vol. 7. P. 14434–14454.

References

1. Richardson C. Microservices patterns: with examples in Java. Shelter Island, New York: Manning Publications, 2019. 490 p.

2. Newman S. Building microservices: designing fine-grained systems. Second Edition. Beijing: O'Reilly Media, 2021. 586 p.

3. Steen M. van, Tanenbaum A.S. Distributed systems. 4th edition, Version 01 (January 2023). Maarten van Steen, 2023. 669 p.

4. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 611 p.

5. Fielding R.T. et al. Reflections on the REST architectural style and "principled design of the modern web architecture" (impact paper award) // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn Germany: ACM, 2017. P. 4–14.

6. Fowler M. Richardson Maturity Model [Electronic resource]. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html.

7. Mastrangelo C. gRPC + JSON [Electronic resource]. 2018. URL: https://grpc.io/blog/grpc-with-json/.

8. Viotti J.C., Kinderkhedia M. A Survey of JSON-compatible Binary Serialization Specifications: arXiv:2201.02089. arXiv, 2022.

9. Zhang L. et al. High performance microservice communication technology based on modified remote procedure call // Sci Rep. 2023. Vol. 13, № 1. P. 12141.

10. Falahah, Surendro K., Sunindyo W.D. Circuit Breaker in Microservices: State of the Art and Future Prospects // IOP Conf. Ser.: Mater. Sci. Eng. 2021. Vol. 1077, № 1. P. 012065.

11. Mendonca N.C. et al. Model-Based Analysis of Microservice Resiliency Patterns // 2020 IEEE International Conference on Software Architecture (ICSA). Salvador, Brazil: IEEE, 2020. P. 114–124.

12. Miraj M., Fajar A.N. Model-Based Resilience Pattern Analysis For Fault Tolerance In Reactive Microservice. Vol. 2022. № 9.

13. Erdenebat B., Bud B., Kozsik T. Challenges in service discovery for microservices deployed in a Kubernetes cluster – a case study // Infocommunications journal. 2023. Vol. 15, № Special Issue. P. 69–75.

14. Moore J. et al. Zest: REST over ZeroMQ: arXiv:1902.07009. arXiv, 2019.

15. Kasarlewar O., Desai P.S. Secure Broker-less Publish/Subscribe System using ECC. 2015. Vol. 6.

16. Jairange R.A., Gupta A.K. Secure Brokerless System for Publisher/Subscriber Relationship in Distributed Network. 2016. Vol. 6, № 7.

17. Moll P. et al. Resilient Brokerless Publish-Subscribe over NDN // MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM). San Diego, CA, USA: IEEE, 2021. P. 438–444.

18. Rathore N., Kant S. Enhanced Blockchain Application for Pub-Sub Model // Advances in Electromechanical Technologies / ed. Pandey V.C., Pandey P.M., Garg S.K. Singapore: Springer Singapore, 2021. P. 299–311.

19. Maharjan R. et al. Benchmarking Message Queues // Telecom. 2023. Vol. 4, № 2. P. 298–312.

20. Fu G., Zhang Y., Yu G. A Fair Comparison of Message Queuing Systems // IEEE Access. 2021. Vol. 9. P. 421–432.

21. Hegde R.G. Low Latency Message Brokers. 2020. Vol. 07, № 05.

22. Chy M.S.H. et al. Comparative Evaluation of Java Virtual Machine-Based Message Queue Services: A Study on Kafka, Artemis, Pulsar, and RocketMQ // Electronics. 2023. Vol. 12, № 23. P. 4792.

23. Sekhar R.R., Gadad V. Microservices, Saga Pattern and Event Sourcing: A Survey. 2020. Vol. 07, № 05.

24. Gordesli M., Nasab A., Varol A. Handling Rollbacks with Separated Response Control Service for Microservice Architecture // 2022 3rd International Informatics and Software Engineering Conference (IISEC). Ankara, Turkey: IEEE, 2022. P. 1–4.

25. Daraghmi E., Zhang C.-P., Yuan S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture // Applied Sciences. 2022. Vol. 12, № 12. P. 6242.

26. Dürr K., Lichtenthäler R., Wirtz G. Saga Pattern Technologies: A Criteria-based Evaluation: // Proceedings of the 12th International Conference on Cloud Computing and Services Science. Science and Technology Publications, 2022. P. 141–148.

27. Štefanko M., Chaloupka O., Rossi B. The Saga Pattern in a Reactive Microservices Environment: // Proceedings of the 14th International Conference on Software Technologies. Prague, Czech Republic: SCITEPRESS – Science and Technology Publications, 2019. P. 483–490.

28. Resilience strategies [Electronic resource]. URL: https://www.pollydocs.org/strategies/index.html.

29. Cloud Design Patterns [Electronic resource]. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/.

30. Sivrikaya F. et al. Internet of Smart City Objects: A Distributed Framework for Service Discovery and Composition // IEEE Access. 2019. Vol. 7. P. 14434–14454.