УДК 004.4
DOI https://doi.org/10.32782/2663-5941/2023.6/15

Котенко М.М.
Державний університет «Житомирська політехніка»

Вакалюк Т.А.
Державний університет «Житомирська політехніка»

ТЕОРЕТИЧНИЙ АНАЛІЗ ПРИНЦИПІВ ВЗАЄМОДІЇ МІКРОСЕРВІСНИХ КОМПОНЕНТІВ

В контексті постійного розвитку розподілених систем дана робота аналізує динамічну взаємодію мікросервісних компонентів, зокрема, враховуючи рівновагу між синхронними та асинхронними методами комунікації. Взаємосервісна комунікація є фундаментальним елементом архітектури мікросервісів, відіграючи ключову роль у визначенні ефективності, надійності та масштабованості системи. Робота систематично аналізує основні парадигми, розкриваючи різницю між прямими взаємодіями, характерними для синхронних систем, та характеристиками асинхронних систем, які ґрунтуються на подієвій моделі. Такі диференціації важливі для глибокого розуміння впливу кожної з них на архітектуру системи та її продуктивність. Асинхронна комунікація характеризується здатністю ініціювати події, на які можуть реагувати кілька учасників, що сприяє гнучкості, адаптивності та резистентності системи до зовнішніх змін. Децентралізація у відносинах між ініціаторами та слухачами подій відображає принципи динамічного обміну інформацією, підкреслюючи її ключове значення в контексті сучасних технологічних реалій. Однак перехід до цієї парадигми не обходиться без перешкод. Спеціалісти, звиклі до традиційних підходів управління компонентами, можуть зіткнутися з концептуальними та технічними труднощами при адаптації до цієї моделі. У сценаріях, де синхронна комунікація є оптимальною для випадків, які потребують миттєвої взаємодії, її обмеженості в архітектурі мікросервісів, такі як потенційні затримки, надмірне навантаження та нестабільність служб, стають помітними. Хоча асинхронні механізми відзначаються своєю ефективністю в багатозадачних операціях, їх адаптація супроводжується власними викликами, включаючи забезпечення доставки, інтегральність послідовностей та обробку помилок. Даний аналіз акцентується на необхідності обережного підходу до вибору методів комунікації в мікросервісах. Врахування переваг та обмежень кожного методу може сприяти розробці надійних, масштабованих та ефективних систем. Перспективи розподіленого обчислення тісно пов'язані з розумінням та застосуванням цього балансу.

Ключові слова: *мікросервіси, взаємодія компонентів, синхронна комунікація, асинхронна комунікація, подієво-орієнтоване спілкування, брокер подій, балансування навантаження.*

Постановка проблеми. В епоху стрімкого технологічного розвитку, коли організації змушені адаптуватися до непередбачувано швидких змін, вибір відповідної архітектури програмних систем стає основою конкурентоспроможності. Мікросервісна архітектура, що використовує автономні служби, кожна з яких спеціалізується на конкретній функції і спілкується через відкриті інтерфейси, відіграє тут ключову роль.

Основний акцент у мікросервісному підході ставиться на взаємодію цих компонентів. Ця взаємодія – це не просто технічна характеристика; вона відображає глибокі організаційні принципи, командну динаміку і, в кінцевому підсумку, бізнес-стратегію. Здатність компонентів ефективно взаємодіяти, адаптуватися до помилок, обмінюватися даними швидко і надійно, масштабуватися під зростаючі потреби – це те, що визначає реальну цінність та потужність мікросервісної архітектури.

Важливість правильної взаємодії між мікросервісами важко переоцінити. Якщо взаємодія оптимізована, система може швидко реагувати на зміни, надаючи високу продуктивність і стабільність. З іншого боку, недоліки у координації можуть призвести до затримок, помилок і, як наслідок, до зниження довіри клієнтів.

Аналіз останніх досліджень і публікацій. Огляд наукових досліджень та публікацій демонструє неперервну еволюцію парадигми мікросервісної архітектури. Згідно з концептуалізацією, запропонованою Льюїсом і Фаулером, мікросервіси виступають як адаптивна структурна одиниця, що фасилітує прискорене розгортання та оптимізацію процесів технічної підтримки програмного забезпечення [1]. Річардсон, у своїй монографії

“Microservices Patterns”, виокремлює архітектурні патерни та методологічні підходи, які підсилюють дизайн та імплементацію мікросервісів [2]. Робота Мітри та співавторів оцінює мікросервісну архітектуру через призму ключових принципів, втілених практик та корпоративної культури, що є необхідними для ефективної інтеграції [3]. Шерманн, Цітo та Лайтнер проводять аналіз реальних індустріальних стратегій, використовуючи мікросервіси як інструментарій для модернізації об’ємних монолітних систем [4]. Критичний розгляд трансформаційних процесів від монолітичної архітектури до мікросервісно орієнтованої та виявлення взаємозв’язків між компонентами представлено у дослідженні Мазламі, Цітo та Лайтнера [7]. Різноманітність методологічних підходів до реалізації комунікації між сервісами піддається деталізованому аналізу в роботі Сліви та Панчика, які проводять порівняльний огляд програмних інтерфейсів REST API, GraphQL і gRPC [12]. Обговорення важливості компетентного управління навантаженням у мікросервісних архітектурах знаходить своє відображення у широкому спектрі наукових публікацій [13, 14, 15, 18, 19], які підкреслюють імператив оптимізації розподілу запитів та часу відгуку між сервісами. Тежас і Хемаваті присвячують своє дослідження використанню Apache Kafka як інструменту для ефективної взаємодії між мікросервісами, підкреслюючи його роль у забезпеченні високої пропускної здатності [22]. Асрі та коавтори розглядають імплементацію асинхронних мікросервісів в контексті "Smart Village", з акцентом на зростанні ефективності через асинхронний режим обробки даних [23]. Асинхронний шаблон запит-відповідь презентує модель для створення децентралізованих систем, здатних виконувати запити у фоновому режимі, тим самим підвищуючи пропускну спроможність та еластичність системи [24].

Мета дослідження – теоретичний аналіз принципів взаємодії мікросервісних компонентів, що полягає у вивченні механізмів спілкування між мікросервісами, виявленні потенційних викликів цієї взаємодії та розгляді методів їх оптимізації.

Виклад основного матеріалу. У світі інженерії програмного забезпечення архітектурні рішення постійно еволюціонують, пристосовуючись до змінних бізнес-потреб, досягнень у технологіях та розмірів організацій. Серед безлічі варіантів архітектури виокремився мікросервісний підхід, який глибоко перетворює процес проектування, розробки та розгортання програмного забезпечення.

У минулому багато програмних додатків стартували як монолітні структури. У монолітній архітектурі всі функціональні компоненти, від елементів інтерфейсу користувача до модулів обробки даних та операцій з базами даних, були тісно інтегровані та об’єднані в єдиний, незмінний блок. Незважаючи на те, що ця модель пропонує простоту на початкових етапах розробки та розгортання додатку, з часом вона ставала все більшою та менш гнучкою зі зростанням додатка. Масштабування конкретних функцій, внесення змін чи впровадження нових технологій в монолітичну систему стає складним та ризиковим процесом, і часто потребує обширних змін та комплексного тестування всього додатка.

Мікросервіси виникли, щоб вирішити ці проблеми, сприяючи впровадженню розподіленого підходу до проектування програмного забезпечення [2, с. 4–8]. У цій архітектурі програми розкладаються на невеликі, незалежні та слабко зв’язані сервіси. Кожен сервіс відповідає за свій власний функціонал, функціонує автономно і взаємодіє з іншими сервісами через чітко визначені інтерфейси, такі як HTTP/REST або черги повідомлень.

З появою більш очевидних переваг мікросервісів – таких як швидші цикли випуску, покращена масштабованість, стійкість та можливість використання різних технологій – цей архітектурний стиль почав отримувати все більше популярності. Використання контейнерів та інструментів для оркестрації, таких як Docker і Kubernetes, додатково сприяло поширенню мікросервісів, надаючи легке, стандартизоване та розширюване середовище для розгортання окремих сервісів.

Слід зауважити, що термін «мікросервіси» не має універсально прийнятого визначення. Тим не менше, є декілька ключових особливостей, які програмна система має мати, щоб заслуговувати на класифікацію як система, заснована на парадигмі мікросервісів. Серед цих характеристик можна вказати на:

– **Невеликий розмір**. В мікросервісному домені розмір сервісу є умовним і підвладним різноманітним факторам. Якість сервісу може визначатися на основі основних бізнес-операцій або ж спрямовуватися на конкретні завдання. Проте за результатами дослідження [4], що включало аналіз 40 організацій, існує загальна схильність уникати екстремально коротких сервісів з менше ніж 100 рядків коду, а також дуже об’ємних, які містять понад 1000 рядків коду. Цей аналіз підкреслює тенденцію до розробки сервісів із чітко визначеною метою та специфікацією для ефективного виконання певної функції.

– **Децентралізація.** Мікросервісні системи активно поширюють ідею розподілених процесів, але вони не є єдиною архітектурою з такою концепцією. Архітектура орієнтована на сервіс (SOA), яка є попередницею в області програмної інженерії, вже використовувала такий розподілений підхід. Основна відмінність мікросервісів полягає у збільшеному ступені децентралізації, зокрема в аспекті управління даними. У мікросервісному підході кожна бізнес-задача та її асоційовані дані існують в рамках окремого, самодостатнього сервісу. Такий сервіс може незалежно від інших розгортатися та працювати в своєму власному хостинговому середовищі [5].

– **Обмежений контекст (Bounded Context).** У рамках архітектури мікросервісів кожен сервіс ретельно розробляється навколо конкретної бізнес-функції, а набір таких функцій формує домен [6, с. 40–42]. Ці виокремлені сервіси функціонують у межах своїх відповідних доменів, забезпечуючи їх захист від складностей більшої системи. Такий підхід за своєю природою сприяє підвищенню автономії, сприяє децентралізованому управлінню та спрощує процеси оновлення або заміни сервісів [7]. Якщо сервісу потрібна взаємодія за межами свого рідного домену, вона координується через чітко визначений інтерфейс на загальному рівні домену. Цей принцип проектування забезпечує чітке відокремлення завдань і зберігає цілісність контекстуальних меж кожного мікросервісу.

При розробці систем на основі мікросервісної архітектури важливо правильно вибрати метод комунікації [8, с. 39]. Міжпроцесорне спілкування в мікросервісах має особливе значення порівняно з монолітними системами [2, с. 65–66]. У монолітних системах модулі легко взаємодіють завдяки узгодженим мовним структурам. Натомість мікросервісна архітектура передбачає поділ на ізольовані сервіси, які можуть функціонувати в різних обчислювальних середовищах. Основним критерієм вибору способу міжпроцесорної взаємодії в таких системах є його синхронність чи асинхронність. Такий вибір значущо впливає на продуктивність, надійність та гармонійність архітектури системи.

У контексті мікросервісів, синхронна комунікація традиційно базується на моделі «запит/відповідь». В цій схемі, мікросервіс ініціює запит до іншого сервісу та очікує його відгуку. Зазвичай, відправник тимчасово зупиняє свою діяльність, очікуючи відповіді від приймача. Ця взаємодія часто здійснюється за допомогою HTTP. Тобто, клієнт робить виклик, передаючи свій запит, зазвичай у форматі JSON або XML через HTTP, а потім чекає відповіді. Для реалізації такої синхронної комунікації в мікросервісних архітектурах на передовій стоять технології, як REST API, gRPC та GraphQL.

REST (Representational State Transfer) API – це архітектурний підхід, який базується на протоколі HTTP та визначає правила для створення веб-сервісів. Системи, побудовані за цією концепцією, характеризуються легкістю, адаптивністю та гнучкістю масштабування. RESTful API використовує HTTP-запити для взаємодії з ресурсами, які зазвичай представлені через URL, здійснюючи дії створення, читання, оновлення та видалення (CRUD). Основна перевага цього підходу полягає в тому, що кожен запит містить всі необхідні дані для його правильної обробки. Використовуючи загальноприйняті методи HTTP, REST API забезпечує інтуїтивно зрозумілі точки доступу та використовує HTTP-статуси для вказівки на результат обробки запиту [9, с. 2–9].

gRPC – це відкрита високопродуктивна платформа для викликів віддалених процедур (RPC), спочатку розроблена Google. Вона використовує передові можливості HTTP/2 та Protocol Buffers (Protobuf) для надання платформи двійкової серіалізації, що не залежить від мови. Засновуючись на принципах мультиплексованих, двосторонніх потоків, gRPC значно підвищує продуктивність порівняно з традиційними комунікаційними протоколами [10]. Адаптивність цього фреймворку до різноманітних платформ та його підтримка потокової передачі роблять його незамінним інструментом для розробки додатків у хмарному середовищі, особливо в мікросервісних архітектурах.

GraphQL – мова запитів до API, створена Facebook у 2012 році та оприлюднена у 2015 році, яка надає стійке та гнучке середовище для доступу до специфічних даних. Відрізняючись від традиційних REST API з їхніми фіксованими кінцевими точками, GraphQL дозволяє користувачам звертатися саме до тих даних, які їм необхідні, оптимізуючи мережеві запити. Її основа базується на системі типів, що визначає структуру даних та можливі операції над ними, стаючи чітким контрактом між клієнтом і сервером. Такий схемний підхід, доповнений інтроспективними функціями GraphQL, надає можливості для глибокої валідації, автозаповнення та розширених інструментів [11, с. 28–29].

Вибираючи синхронну технологію для взаємодії мікросервісів, слід звернути увагу на висно-

вки дослідження [12], де були показані різниці в ефективності різноманітних технологій обміну даними. Відзначено, що gRPC є найбільш ефективним для швидкого обміну даними між клієнтом та сервером, роблячи його оптимальним вибором для додатків, що вимагають швидкої передачі даних без урахування їх об'єму. GraphQL, за даними [12], мав труднощі при роботі з меншими наборами даних, але продуктивність зростала із збільшенням об'єму даних. Дослідження також вказує, що REST має стабільну і збалансовану продуктивність в різних умовах, що підкреслює його широке застосування. Особливий акцент зроблено на кореляцію між використанням пам'яті та продуктивністю: gRPC, незважаючи на високу швидкість, потребує значних ресурсів пам'яті. На підставі вищезазначеного [12], можна зробити висновок, що вибір технології повинен базуватися на специфіці потреб сервісів. gRPC ідеально підходить для тих, хто шукає швидкість, тоді як REST залишається універсальним рішенням. GraphQL рекомендується для сервісів, що працюють із складними структурами даних і операціями на основі запитів.

Синхронні виклики у контексті мікросервісної архітектури асоціюються з простотою та інтуїтивністю, відображаючи традиційний послідовний підхід до програмування. Більшість розробників звикли до синхронного стилю, використовуючи його для створення SQL чи HTTP запитів. Тому при переході до розподіленої архітектури цей стиль виглядає знайомим. Проте, не дивлячись на зрозумілість та простоту імплементації, синхронна взаємодія приховує ряд обмежень та ризиків. Основна проблема полягає в тому, що при збою одного мікросервісу, інші, які на нього посилаються, можуть заблокуватися, очікуючи відгуку. Така поведінка може створити ланцюговий ефект, коли недоступність одного компонента веде до проблем в усій системі. До того ж, можливі затримки у взаємодії між мікросервісами можуть погіршити загальну продуктивність і збільшити час очікування відгуку [8, с. 96].

Для вирішення вищезазначених викликів часто використовують балансування навантаження. Використання різних методів та технік балансування навантаження може забезпечити стабільність розподіленої системи, розподіляючи завдання між різними мікросервісами та ресурсами. Це забезпечує уникнення "вузьких місць", максимізуючи ефективність системи. Таким чином, замість того, щоб допустити, що одна помилка впливає на усю систему, балансування навантаження розподіляє запити, зменшуючи ризики та підвищуючи загальну продуктивність та відгук системи [13].

Балансування навантаження зазвичай поділяється на дві категорії: статичне балансування навантаження та динамічне балансування навантаження.

Статичні стратегії балансування навантаження передбачають розподіл трафіку між сервісами на основі типової діяльності системи або заздалегідь визначених ресурсів. Водночас, рішення про перерозподіл навантаження приймається без врахування актуального стану системи [15]. Хоча статичні алгоритми балансування навантаження є простими у реалізації та впровадженні, вони можуть викликати проблеми в продуктивності системи через ризик перевантаження чи недостатнього завантаження [14].

Найпопулярнішими статичними методами балансування навантаження є наступні:

– **Round-Robin** – алгоритм циклічного планування. Його головна ідея полягає в цілеспрямованому розподілі завдань між вузлами в послідовному порядку. Починаючи з довільно вибраного вузла, завдання розподіляються послідовно. Кожний сервер обслуговує процеси без попереднього призначення пріоритетів, що забезпечує відсутність затримок. При рівному розподілі навантаження це забезпечує швидке реагування. Але існують випадки, коли процеси потребують різного часу на виконання: окремі вузли можуть перевантажуватися, а інші залишатися невикористаними [15].

– Weighted Round-Robin – удосконалена версія алгоритму циклічного планування. Визначає вагу кожного сервера з урахуванням його потужності та можливостей [15]. За допомогою цих ваг завдання розподіляються так, що сервери із вищою потужністю отримують більше завдань [14]. У випадку, коли всі сервери мають однакову вагу, трафік рівномірно розподіляється між ними [15]. Незважаючи на ефективність цього методу у розподілі завдань відповідно до потужності сервера, він може бути трохи повільнішим в порівнянні із звичайною версією цього алгоритму [14]. Зазвичай, адміністратори визначають ваги серверів, виходячи з їхньої оцінки потужності сервера та його спроможності обробляти трафік.

– IP Hashing – за допомогою IP-адреси клієнта, цей підхід генерує специфічний хеш-ключ, щоб вибрати найкращий сервер для обробки запиту. Інтегруючи IP-адреси клієнта та сервера, метод створює унікальний ключ для спрямування

клієнта на відповідний сервер [17]. У випадках, коли сеанс розривається, ключ може бути повторно згенерований, гарантуючи, що запит клієнта залишається у відповідності з початковим сервером [16, 17]. Ця система необхідна, коли потрібно підтримувати активні сеанси або зберігати інформацію під час ряду звернень клієнта [16]. Такий підхід є ключовим елементом балансування навантаження, що забезпечує стабільність у співпраці між клієнтом та сервером.

Динамічні методи балансування навантаження, на відміну від статичних, спрямовані на виявлення найменш завантажених серверів або віртуальних машин у системі, щоб гарантувати ефективний розподіл завдань [14, 15]. Використання таких методів передбачає взаємодію в реальному часі з мережею, що може спричинити зростання трафіку. Основною відмінністю методів динамічного балансування є їх гнучкість: ці методи мають можливість реагувати на поточний стан системи, перерозподіляючи задачі від перевантажених до менш зайнятих ресурсів\серверів у реальному часі [15].

Наступні методи динамічного балансування навантаження найбільше зустрічаються на практиці:

– **Least Connection** – це стратегія балансування навантаження направляє користувацькі запити до сервера з найменшим числом активних з'єднань [17]. Відповідно до цього методу, сервери з меншим числом з'єднань вважаються менш завантаженими та краще підходять для обробки нових запитів. Система рівномірно розподіляє навантаження, запобігаючи можливим перевантаженням. Метод враховує, що всі з'єднання потребують приблизно однакових ресурсів, тому він особливо ефективний для серверів з аналогічними характеристиками, що мають тривалі активні з'єднання [18].

– **Weighted least connection** – базується на методі мінімального з'єднання. При цьому підході, коли балансувальник навантаження отримує новий запит на завдання, він вибирає сервер на основі його поточної кількості з'єднань відносно призначеного йому вагового коефіцієнта в межах групи серверів [19]. Адміністратори мають можливість призначати різні вагові коефіцієнти кожному серверу, враховуючи, що деякі сервери можуть обробляти більшу кількість з'єднань, ніж інші. Цей метод враховує різні можливості кожного серверу та вплив кількості з'єднань, що підвищує ефективність системи.

– **Weighted response time** – ця стратегія враховує середній час відповіді кожного сервера у мережі, щоб приймати обґрунтовані рішення щодо маршрутизації. Вивчаючи як час відповіді, так і поточну кількість відкритих з'єднань, які підтримує кожен сервер, алгоритм може більш точно визначити, куди направляти вхідний трафік [20, 21]. Основна перевага цього підходу полягає в тому, що він постійно прагне направляти користувачів до серверів, які демонструють найшвидший час відповіді, забезпечуючи таким чином оптимальний та швидкий досвід обслуговування для кінцевих користувачів.

– **Resource-based** – ресурсна стратегія оцінює реальні доступні ресурси кожного сервера у мережі. Для цього кожен сервер оснащений спеціалізованим програмним забезпеченням, яке часто називають «агентом» [21]. Цей агент відповідає за постійний моніторинг та звітність про доступні обчислювальні ресурси сервера, такі як використання ЦПУ та доступну пам'ять. Перед направленням трафіку балансувальник навантаження спілкується з цим агентом, щоб визначити поточну ємність сервера [20]. В результаті трафік направляється до серверів на основі їх реальної ємності та готовності обробляти запити, забезпечуючи збалансоване навантаження та запобігання перевантаженню будь-якого окремого сервера.

Враховуючи попередньо описані труднощі синхронної взаємодії, слід зазначити, що синхронне спілкування між сервісами не завжди задовольняє потреби, особливо у системах, що постійно змінюються, займаються обробкою великих потоків даних чи вимагають високу надійність. Тому індустрія все більше звертається до асинхронних методів спілкування. Ці методи дозволяють службам взаємодіяти без постійного очікування відповідей від інших учасників системи [22]. Служби можуть займатися паралельними завданнями, а отримавши повідомлення або запит вони негайно приступають до його аналізу та опрацювання. Така модель не лише оптимізує роботу, але й допомагає уникнути численних проблем, характерних для синхронної комунікації. Важливо підкреслити, що асинхронність сприяє зменшенню прямих залежностей між мікросервісами, що в свою чергу покращує стабільність системи, сприяє легшому розширенню та масштабуванню мікросервісної архітектури.

При детальному вивченні асинхронних методів комунікації можна виявити ряд специфічних підходів до передачі інформації, кожен з яких має свої характерні переваги та рекомендовані області застосування.

Комунікація через спільні дані – у цій моделі вищестоящий сервіс змінює спільно використовувані дані, які після цього використовуються

одним або декількома нижчестоящими сервісами для визначення конкретних операцій. Цей метод відзначається своєю простотою та універсальністю. Його можна без зусиль реалізувати, використовуючи добре відомі технології, такі як читання/запис файлів або операції з базами даних. Такий підхід сприяє не лише легкості впровадження, але й забезпечує взаємодію між широким спектром систем [8]. Ця модель ідеально підходить для обробки великих об'ємів даних, але може мати затримки через механізми опитування, що робить її менш оптимальною для обробки даних в реальному часі. Хоча можна розширити цю модель додатковими викликами для сповіщення нижчестоящих сервісів про нові дані, для негайної обробки даних ефективніше використовувати технології, такі як Kafka [22]. Однак основний виклик цього способу комунікації полягає в потенційних залежностях від спільного сховища даних. Його зміни можуть впливати на стабільність взаємодії сервісів. Надійність цієї моделі прямо залежить від стабільності та цілісності сховища даних [8, с. 102–104].

Комунікація по принципу «запит-відповідь» – цей метод взаємодії застосовується, коли від отримувача очікується відгук, але не негайно. В такій асинхронній комунікації сервіси не спілкуються безпосередньо; замість цього вони обмінюються повідомленнями за допомогою брокера. Сервіс надсилає повідомлення до брокера, а інші сервіси підписуються, щоб отримувати відповідні повідомлення [23]. Такий підхід зменшує складність безпосереднього спілкування мікросервісів, полегшуючи оновлення та масштабування, дозволяє інформувати відправника про результати обробки повідомлення у неблокуючий спосіб та чудово підходить для виконання довготривалих завдань.

Як приклад цього виду комунікації можна розглянути ситуацію з області електронної комерції, де система замовлень резервує товари в системі інвентаризації. Замість того, щоб безпосередньо запитувати систему інвентаризації, система замовлення надсилає запит, який ставиться в чергу. Система інвентаризації, коли їй зручно, обробляє цей запит і надсилає відповідь. Цю відповідь можна помістити в іншу чергу, яку пізніше перевіряє система замовлення [8, с. 106].

Однак, такий метод має свої складнощі. Поєднання відповідей з їхніми початковими запитами може бути непростим, особливо із затримками або коли є кілька шляхів обслуговування [24]. Крім того, асинхронна модель «запит-відповідь» вимагає стратегій для вирішення можливих затримок, щоб уникнути нескінченного очікування відповіді, а також більш просунутого механізму обробки помилок.

Комунікація, орієнтована на конкретні події (подієво-орієнтована комунікація), є методологією проектування, де специфічні дії або комунікаційні процеси ініціюються на основі попередньо встановлених подій [8, с. 111]. Цей підхід представляє зсув парадигми від традиційних схем «запит-відповідь», акцентуючи увагу на більш гнучкому та адаптованому механізмі комунікації [25].

Основний принцип подієво-орієнтованого спілкування полягає в розумінні подій як важливих змін у стані інформації [25]. Замість традиційних способів обміну повідомленнями, де служби мають чітке уявлення про структуру та вміст повідомлення, в даному методі комунікація не завжди потребує такої деталізації. Служби або компоненти комунікують між собою, відправляючи події, які вони генерують самостійно. Ці події діють як сигнали або активатори для інших частин системи чи окремих систем, свідчачи про виконання певної умови або здійснення конкретної дії [23].

У цій конфігурації ключову роль відіграють посередники або брокери подій. Вони служать мостом, обробляючи підписки на події та забезпечуючи доставку сповіщень усім релевантним підписникам [8, с. 110; 26]. Особливість цієї системи полягає в тому, що вона децентралізована: видавці подій, зазвичай, не мають інформації про підписників і навпаки. Така схема підтримує моделі комунікації типу «один до багатьох» або «багато до багатьох», де численні видавці можуть ініціювати події, на які, в свою чергу, можуть відгукуватися багато підписників [25].

Центральною особливістю у цій комунікації є роз'єднаність між видавцями подій та підписниками, що дозволяє взаємодіям базуватися на моделі публікації та підписки. Така структура підсилює гнучкість системи, додає їй адаптивності та стійкості до непередбачених змін. Це знижує пряму залежність від конкретних подій, відкриваючи шлях для більш динамічного обміну інформацією [23].

У світі сучасних технологій, особливо у контексті мікросервісів, такий підхід до комунікації зарекомендував себе як надзвичайно ефективний. Він забезпечує слабку взаємозалежність між компонентами, що є критично важливим для складних розподілених систем. Це особливо корисно у сценаріях, де необхідно передавати динамічну

інформацію між різними службами або коли компоненти мають здійснювати незалежні дії, базуючись на зовнішніх подіях [8, с. 115]. Інновації у сфері Інтернету речей [26], в свою чергу, демонструють, як цей підхід може бути адаптований для різноманітних додатків, оптимізуючи комунікацію та масштабованість.

Впровадження подієво-орієнтованого спілкування приносить не лише переваги, але й виклики. Для команд і фахівців, які вперше стикаються з таким стилем взаємодії, можуть виникнути певні складнощі у сприйнятті та адаптації. Основна зміна полягає у тому, що фокус зсувається з прямого управління компонентами до самостійного визначення їх дій на основі зовнішніх подій. Це ставить перед розробниками завдання не лише технічного характеру, а й концептуального, оскільки вони повинні переосмислити спосіб взаємодії компонентів системи.

Поглиблюючи розуміння типів взаємодії мікросервісних компонентів, стає важливим докладніше вивчити випадки, коли варто вибрати синхронні чи асинхронні механізми комунікації.

Синхронна комунікація є невід'ємною частиною багатьох сучасних систем, особливо там, де швидкість реакції та отримання зворотного зв'язку є критичною. Наприклад, системи обробки запитів у онлайн-базах даних, транзакційні системи фінансових установ або навіть звичайний веб-запит до сервера. У всіх цих випадках користувач або система надсилає запит і очікує негайної відповіді перед тим, як продовжити наступний крок. Проте, не дивлячись на її простоту та ефективність у певних сценаріях, синхронна комунікація може стати проблемою в мікросервісних системах. Затримки мережі, високе навантаження на сервери та недоступність певних сервісів можуть стати перешкодами для її ефективності.

З іншого боку, асинхронна комунікація являється фундаментом систем, що працюють на основі подій чи конкретних тригерів. Такий підхід дозволяє системам відправляти повідомлення, переходити до інших завдань та обробляти відповідь після її отримання. Це робить цю модель підходящою для багатогранних операцій. Наприклад, коли одна дія активує кілька сервісів, асинхронна комунікація гарантує, що затримка в одному компоненті не блокує весь процес. Вона також незамінна в довготривалих завданнях. Уявімо завдання з обробки великого обсягу даних; тут асинхронний підхід гарантує, що ресурси не будуть прив'язані, очікуючи завершення процесу. Проте асинхронність також має свої виклики, зокрема гарантію доставки повідомлень, збереження послідовності та управління повідомленнями які дублюються.

Висновки. У даній роботі були розглянуті та проаналізовані механізми комунікації мікросервісних компонентів, акцентуючи увагу на відмінностях між синхронними та асинхронними методами. Синхронний метод комунікації, характеризований оперативністю реакцій, є есенціальним в контекстах, що потребують негайного зворотного зв'язку. Він демонструє ефективність у визначених умовах, проте може стикаються з викликами в структурах мікросервісів, зокрема через можливі затримки та перебої в обслуговуванні. З іншого боку, асинхронний метод базується на подієвій моделі та специфічних тригерах, забезпечуючи системам можливість паралельної обробки завдань та негайного реагування на вхідні події. Ця методика є оптимальною для комплексних операцій, зокрема, при активації декількох служб однією подією або в контексті тривалих процесів. Втім, асинхронний метод презентує ряд технічних викликів, таких як забезпечення доставки повідомлень, гарантія послідовності та управління дублюванням повідомлень. Вибір конкретного механізму комунікації визначається специфічними потребами системи, що підкреслює важливість глибокого аналітичного осмислення обох підходів.

Список літератури:

1. Lewis J., Fowler M. Microservices: a definition of this new architectural term. URL: https://martinfowler.com/articles/microservices.html (date of access: 20.09.2023).
2. Richardson C. Microservices Patterns: With examples in Java. Manning Publications, 2018. 522 p.
3. Microservice Architecture: Aligning Principles, Practices, and Culture / R. Mitra et al. O'Reilly Media, 2016. 146 p.
4. Schermann G., Cito J., Leitner P. All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. Service-Oriented Computing – ICSOC 2015 Workshops. Berlin, Heidelberg, 2016. P. 36–47. URL: https://doi.org/10.1007/978-3-662-50539-7_4 (date of access: 24.09.2023).
5. Microservices: How To Make Your Application Scale / N. Dragoni et al. Lecture Notes in Computer Science. Cham, 2018. P. 95–104. URL: https://doi.org/10.1007/978-3-319-74313-4_8 (date of access: 01.10.2023).
6. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, Limited, 2021.

7. Mazlami G., Cito J., Leitner P. Extraction of Microservices from Monolithic Software Architectures. 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017. 2017. URL: https://doi.org/10.1109/icws.2017.61 (date of access: 02.10.2023).

8. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Incorporated, 2021. 616 p.

9. Richardson L., Ruby S., Amundsen M. RESTful Web APIs. O'Reilly Media, Incorporated, 2013.

10. Indrasiri K., Kuruppu D. GRPC : up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes. O'Reilly Media, Incorporated, 2020. 204 p.

11. Buna S. GraphQL in Action. Manning Publications Company, 2020. 375 p.

12. Śliwa M., Pańczyk B. Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. Journal of Computer Sciences Institute. 2021. Vol. 21. P. 356–361. URL: https://doi.org/10.35784/jcsi.2744 (date of access: 06.10.2023).

13. Prassanna J., Jadhav P. A., Neelanarayanan V. Towards an Analysis of Load Balancing Algorithms to Enhance Efficient Management of Cloud Data Centres. Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC – 16'). Cham, 2016. P. 143–155. URL: https://doi.org/10.1007/978-3-319-30348-2_13 (date of access: 07.10.2023).

14. Comparative analysis of metaheuristic load balancing algorithms for efficient load balancing in cloud computing / J. Zhou et al. Journal of Cloud Computing. 2023. Vol. 12, no. 1. URL: https://doi.org/10.1186/s13677-023-00453-3 (date of access: 07.10.2023).

15. Shah N., Farik M. Static Load Balancing Algorithms In Cloud Computing: Challenges & Solutions. International Journal of Scientific & Technology Research, Volume 4, October 2015, P. 353-355.

16. Srikanth Dannarapu, Load Balancing Algorithms that can be used in Java Applications. URL: https://medium.com/javarevisited/load-balancing-algorithms-that-can-be-used-in-java-applications-6f605d1bf19 (date of access: 08.10.2023).

17. Pachikkal C. S. Interservice Communication in Microservices. International Journal of Advanced Research in Science, Communication and Technology, Volume 5, Issue 2, May 2021, P. 498-503. URL: https://doi.org/10.48175/568 (date of access: 10.10.2023).

18. Xuan Phi N., Cong Hung T. Load Balancing Algorithm to Improve Response Time on Cloud Computing. International Journal on Cloud Computing: Services and Architecture. 2017. Vol. 7, no. 6. P. 01–12. URL: https://doi.org/10.5121/ijccsa.2017.7601 (date of access: 11.10.2023).

19. Load Balancing Algorithm for Web Server Based on Weighted Minimal Connections. Journal of Web Systems and Applications. 2017. Vol. 1, no. 1. URL: https://doi.org/10.23977/jwsa.2017.11001 (date of access: 11.10.2023).

20. What is load balancing? URL: https://aws.amazon.com/what-is/load-balancing/ (date of access: 12.10.2023).

21. Types of load balancing algorithms. URL: https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/ (date of access: 12.10.2023).

22. Tejas V. G., Hemavathy R. Microservices and its Intercommunication using Kafka. International Research Journal of Engineering and Technology, Volume 7, Issue 4, April 2020, P. 5632-5635. URL: https://www.irjet.net/archives/V7/i4/IRJET-V7I41061.pdf (date of access: 13.10.2023).

23. Implementation of Asynchronous Microservices Architecture on Smart Village Application / S. A. Asri et al. International Journal on Advanced Science, Engineering and Information Technology. 2022. Vol. 12, no. 3. P. 1236. URL: https://doi.org/10.18517/ijaseit.12.3.13897 (date of access: 15.10.2023).

24. Asynchronous Request-Reply pattern. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply (date of access: 16.10.2023).

25. David J. S. Development of a digital twin of a flexible manufacturing system for assisted learning. URL: http://dx.doi.org/10.13140/RG.2.2.26398.08000 (date of access: 20.10.2023).

26. A Publish/Subscribe Protocol for Event-Driven Communications in the Internet of Things / C. Esposito et al. 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)2016, Auckland, 8–12 August 2016. 2016. URL: https://doi.org/10.1109/dasc-picom-datacom-cyberscitec.2016.79 (date of access: 23.11.2023).

Kotenko M.M., Vakaliuk T.A. THEORETICAL ANALYSIS OF MICROSERVICE COMPONENTS INTERACTION PRINCIPLES

Within the context of the continuous evolution of distributed systems, this study delves into the dynamic interplay of microservice components, particularly considering the balance between synchronous and

asynchronous communication methods. Inter-service communication is a fundamental aspect of microservice architecture, playing a pivotal role in determining the system's efficiency, reliability, and scalability. This work systematically examines core paradigms, elucidating the distinctions between direct interactions inherent in synchronous systems and the characteristics of asynchronous systems rooted in an event-driven model. Such differentiations are crucial for a profound understanding of the impact of each on system architecture and its performance. Asynchronous communication is characterized by its capability to initiate events that multiple participants can respond to, fostering flexibility, adaptability, and system resilience to external changes. The decentralization in relations between event initiators and listeners mirrors the principles of dynamic information exchange, underscoring its paramount importance in the realm of contemporary technological realities. However, transitioning to this paradigm is not without challenges. Professionals accustomed to traditional component management approaches may encounter conceptual and technical difficulties when adapting to this model. In scenarios where synchronous communication is optimal for instances requiring immediate interaction, its limitations in microservice architecture, such as potential delays, excessive load, and service instability, become evident. Although asynchronous mechanisms are notable for their efficiency in multitasking operations, their adaptation is accompanied by distinct challenges, including ensuring delivery, sequence integrity, and error handling. This analysis emphasizes the necessity for a judicious approach to selecting communication methods in microservices. Considering the advantages and constraints of each method can aid in developing reliable, scalable, and effective systems. The prospects of distributed computing are intrinsically linked to the comprehension and application of this balance.

Key words: *microservices, component interaction, synchronous communication, asynchronous communication, event-driven communication, event broker, load balancing.*