УДК 004.415.2
DOI https://doi.org/10.32782/2663-5941/2024.4/31

Тарновецька О.Ю.
Чернівецький національний університет імені Юрія Федьковича

Осадчук Р.Р.
Software Engineer, Ajax Systems

ДОСЛІДЖЕННЯ МЕТОДІВ ВЗАЄМОДІЇ МІЖ СЕРВІСАМИ ПРИ МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Метою даною роботи було вивчення наявних методів взаємодії між компонентами програмного продукту, створеного з використанням мікросервісної архітектури. Автори провели детальний аналіз різноманітних протоколів, що використовуються для взаємодії мікросервісів. У процесі дослідження було визначено загальну структуру цих протоколів, основні характеристики їх алгоритмів роботи, процеси передачі даних та способи застосування в контексті мікросервісної архітектури. Особлива увага була приділена таким протоколам, як HTTP/REST та AMQP, які забезпечують різні рівні продуктивності, надійності та зручності використання.

Для демонстрації можливостей мікросервісної архітектури та різних методів інтеграції між сервісами, було розроблено програмне забезпечення (ПЗ), призначене для моніторингу робочого часу підлеглих на підприємстві. Це програмне забезпечення включає набір мікросервісів, кожен з яких виконує окрему задачу, пов'язану з моніторингом, збиранням та аналізом даних про робочий час співробітників.

Одним з ключових етапів дослідження стало визначення найбільш оптимального протоколу взаємодії між компонентами програмного продукту. Для цього було створено ряд тест-кейсів, які моделюють різні сценарії використання програмного забезпечення в умовах реального підприємства. На основі цих тест-кейсів було проведено стрес-тестування, яке дозволило оцінити поведінку прототипів програмного продукту під різними навантаженнями. Це включало моделювання високих навантажень, перевірку стабільності роботи при тривалих запусках та оцінку відмовостійкості системи.

Отримані дані в результаті тестування були ретельно проаналізовані, а виявлені закономірності обґрунтовані. На основі проведеного аналізу було визначено, які протоколи та методи взаємодії забезпечують найбільшу продуктивність, надійність та стабільність роботи програмного забезпечення в умовах реального використання. Ці результати стали основою для формулювання рекомендацій щодо вибору способів інтеграції мікросервісів в процесі планування та розробки нового програмного продукту (ПП).

Ключові слова: *мікросервіс, протокол, система, взаємодія, подія, передача даних.*

Постановка проблеми. В сучасній розробці ПЗ (програмного забезпечення) все частіше практикується використання мікросервісної архітектури замість звичної монолітної. Оскільки мікросервісне програмне забезпечення базується на розподілі задач між окремими компонентами – мікросервісами, виникає проблема вибору ефективного методу їх інтеграції. Однак, використання певного методу взаємодії між мікросервісами має свої наслідки та впливає на процес розробки, функціонування та підтримки програмного забезпечення. Проведене в цій роботі дослідження та аналіз допоможуть визначити ключові фактори, за яких слід будувати зв'язки між сервісами в мікросервісній архітектурі, використовуючи відповідний підхід до їх комунікації.

Аналіз останніх досліджень і публікацій. Проблеми побудови програмного забезпечення з використання мікросервісного архітектурного підходу, а також процеси взаємодії між компонентами програмного продукту, були описані чималою кількість авторів та вчених у своїх публікаціях та науковій літературя. Зокрема, такий мікросервісну архітектуру розглядали Адам Беламейр (Adam Bellamare) [1], Джон Кернел (John Carnell) [6], Сем Ньюман (Sam Newman) [9]. Використання синхронних запитів та відповідей між компонентами програмного забезпечення розглянули Стефен Томас (Stephen Thomas) [3], Барі Поллард (Barry Pollard) [4] та Джим Веббер (Jim Webber) [5]. Використання асинхронної передачі подій в розробці програмного забезпечення описували Альваро Відела (Alvaro Videla) [2] та Адам Беламейр (Adam Bellamare) [1].

Постановка завдання. Метою статті є дослідження методів взаємодії між сервісами у мікро-

сервісній архітектурі програмного забезпечення, визначення найбільш оптимального методу зв'язку між компонентами для досягнення максимальної продуктивності та стабільності роботи на основі розробленого ПЗ для системи моніторингу робочого часу працівників підприємства. Встановлена мета обумовлює наступні завдання:

– дослідження та аналіз методів взаємодії між мікросервісами;

– розробка архітектури прототипу системи;

– побудова прототипів ПЗ;

– тестування розроблених ПП (прототипів програмного продукту);

– аналіз результатів отриманих в ході тестування.

Результати дослідження можуть служити як основні критерії для вибору методу взаємодії між мікросервісами під час планування розробки нового програмного продукту.

Виклад основного матеріалу. Мікросервісний архітектурний підхід являє собою спеціальний метод розробки програмного забезпечення, що ґрунтується на розробці відокремлених модулів (сервісів) з чітко визначеною поведінкою, що інтегровані між собою. Кожен з мікросервісів побудований задля вирішення конкретних бізнес-потреб і може бути розгорнутий на віддаленому сервері незалежно від інших компонентів програмного продукту. Мікросервіси можуть бути реалізовані за допомогою різних мов програмування, фреймворків бібліотек, баз даних та інших технологій для розробки ПЗ. Використання даного архітектурного підходу стало популярним в останні роки, оскільки дозволяє розробляти програмні продукти гнучкими до використання та їх подальшого розширення [9].

На рис. 1 представлено типову схему програмного продукту реалізованого за допомогою монолітної архітектури. Можна побачити, що всі блоки бізнес-логіки розташовані в одну великому монолітному додатку. На Рис.2, в свою чергу, показано типову схему мікросервісної архітектури, де кожен блок бізнес логіки монолітного програмного продукту з рис. 1 був виокремлений в окремий мікросервіс, кожен з яких являє собою невеликий додаток, що має своє унікальне призначення та чітку спеціалізацію.

Все більш часто в сучасній розробці надається перевага використанню мікросервісною архітектури за її численні переваги, серед них можна виділити основні:

1. Гнучкість – мікросервіси можуть бути реалізовані, за допомогою різних мов програмування, бази даних і навколишнього середовища програмного забезпечення. Це дозволяє кожному сервісу бути розгорнутим і працювати незалежно від інших. Таким чином, будь-яка проблема з мікросервісами не впливатиме на цілу систему, і відмова певного мікросервісу може бути компенсована відносно швидко [6].

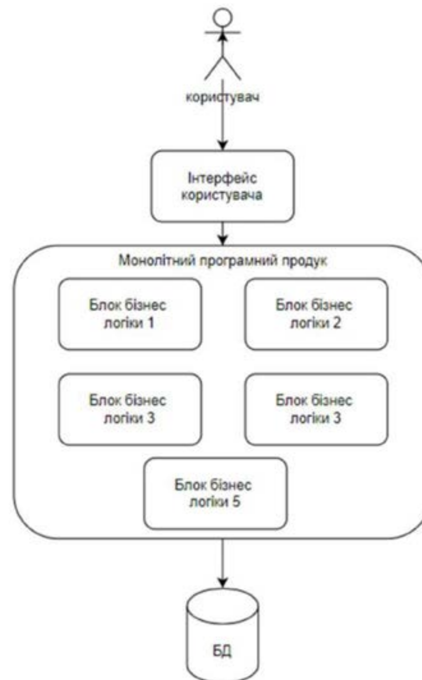

Рис. 1. Типова схема структури монолітного програмного продукту

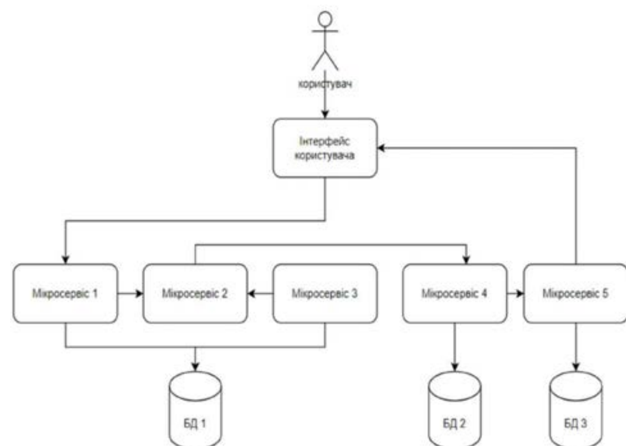

Рис. 2. Типова схема структури програмного продукту написаного з використанням мікросервісного архітектурного підходу

2. Зосередження на бізнес-вимогах – мікросервісна архітектура рекомендує розробникам зосереджуватися навколо створення конкретного бізнес функціоналу замість того, щоб писати проміжний код. Іншими словами, групи розробників відштовхуються від бізнес-вимог, які потрібно реалізувати будь-яким способом, а не від технологій.

3. Простота – кожен мікросервіс являє собою невеликий додаток, що зосереджений на конкретному переліку бізнес задач.

4. Тестування – оскільки мікросервіси реалізують конкретний аспект бізнес функціоналу, то процес тестування стає значно легшим. Кожний окремий мікросервіс може бути протестований індивідуально, з метою виявлення внутрішніх помилок, що потребують додаткового налагодження.

5. Автономність – мікросервісний архітектурний підхід є дуже зручним, якщо у розробці програмного продукту беруть участь декілька команд. Кожна з них може розробляти свій окремий мікросервіс і не залежати від сусідньої допоки не потрібно буде налаштовувати інтеграцію між всіма сервісами.

Однак в мікросервісна архітектура також не є ідеалом у розробці ПЗ і приносить із собою ряд недоліків, з якими розробники змушені миритися та вживати необхідні заходи:

1. Надлишкова зв'язаність – кількість залежностей в середині програмного продукту, розробленого за допомогою мікросервісів, безпосередньо корелює з кількістю використаних сервісів [6].

2. Складність розгортання – мікросервіси вимагають досвідчених DevOps-інженерів, що здатні автоматизувати розгортання, керувати всім життєвим циклом та підтримувати працездатність мікросервісного продукту.

3. Безпека – разом із збільшенням кількості мікросервісів, збільшується кількість зв’язків комунікацій між ними, то і кількість місць у системі, які є вразливими до втрати чи перехоплення певних даних також стрімко збільшується, що значно збільшує шанси отримати несанкціонований доступ до ресурсів ПЗ.

4. Швидкодія – мікросервіси можуть стати причиною затримки у швидкості обробки дій користувача. Якщо певний мікросервіс викликає ще ряд інших мікросервісів, то загальний час обробки дій користувача триватиме довше ніж в монолітному програмному продукті в якому не буде затримки під час передачі даних між різними компонентами системи.

Правильна інтеграція є одним із найбільш важливих аспектом під час проектування архітектури ПЗ із використання мікросервісного підходу. При вдалій реалізації мікросервіси залишаться автономними, що дасть змогу легко впроваджувати новий функціонал і в кожен з них незалежно від всієї системи. Існує два основних підходи реалізації комунікації між мікросервісами: синхронний та асинхронний обмін повідомленнями.

Синхронний метод взаємодії між мікросервісами реалізований за допомогою RESTful архітектурного стилю. The Representational State Transfer (REST) – передача репрезентативного стану, насправді не є протоколом, а архітектурним стилем. Він був вперше представлений Роєм Філдінгом в 2000 році і широко використовується з тих пір. Як правило, RESTful сервіси використовують безпечний і надійний протокол HTTP, який є перевіреним у мережі Інтернет. Безпека гарантується шляхом використання протоколів TLS/SSL [3].

Основними RESTful принципами є:

– Client-Server (з англ. – клієнт-сервер). Сервіс є клієнтом, коли надсилає запит з метою отримання інформації про певний ресурс, власником якого є сервіс-сервер [5] (див. рис. 3).

– Stateless (з англ. – без стану). Даний принцип вимагає, щоб кожного запиту не залежала від стану системи.

– Cacheable (з англ. – здатний до кешування). Дані, що надсилаються між сервером та клієнтом, можуть бути закешовані кожною із сторін.

– Uniform interface (з англ. – однотипний інтерфейс). Якщо застосувати до систем інженерний принцип спільності/одноманіття, то архітектура всього додатка стане простішою, а взаємодія стане прозорою та зрозумілою [8].

– Layered system (з англ. – багаторівнева система). Система має складатись з багаторівневих компонентів так, що кожен з них має доступ тільки до компонентів, що розташовані на сусідніх рівнях [5].

Говорячи в контексті мікросервісної архітектури, REST метод взаємодії між сервісами не завжди може бути корисним. Чим більше у системі мікросервісів тим важче їх інтегрувати одним з одним так, щоб мінімізувати кількість залежностей, що є проблемою тісносної зв’язаності (див. рис. 4).

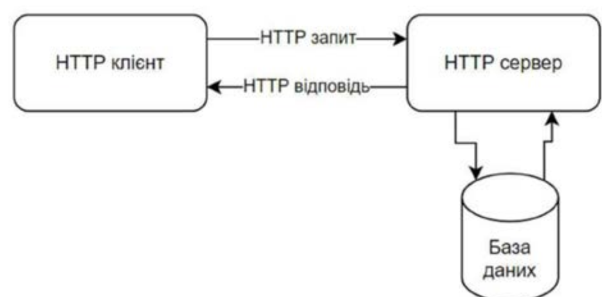

Рис. 3. Схема взаємодії клієнта і сервера з використання RESTful підходу

Кількість максимальних залежностей мікросервісів один від одного пропорційна їх кіль-

кості у системі. Таким чином, можна обчислити кількість зв'язків у системі можна за допомогою наступної формули:

$$К-сть\ зв'язків = N * (N - 1)/2,$$

де N – це кількість мікросервісів у систем.

Використовуючи дану формулу можна побудувати графік залежності максимальної кількості зав'язків у системі від кількості мікросервісів (див. рис. 5).

Ще однією проблемою є блокування – після надсилання HTTP запиту на мікросервіс-сервер, мікросервіс-клієнт стає недоступними до роботи допоки він не отримує відповідь на свій запит.

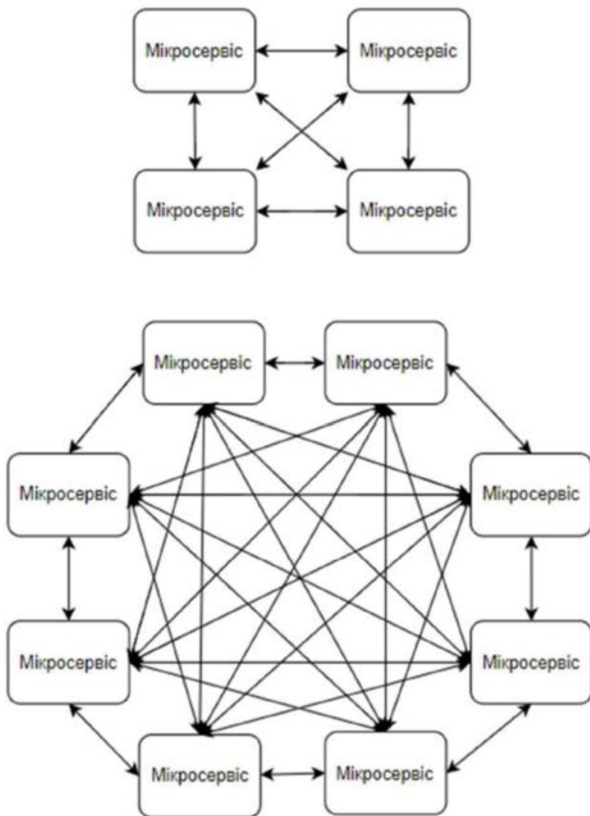

Рис. 4. Діаграма максимальної кількості зв'язків між мікросервісами при використанні REST підходу

Проблеми такого роду можуть бути подолані разом із використанням асинхронного методу взаємодії між мікросервісами, що базуються на Event-oriented (з англ. – подіє-орієнтований) архітектурному підході. Головною суттю даного підходу є наявність брокеру повідомлень (від англ. message broker) у системі мікросервісів, що являє собою проміжне програмне забезпечення, яке полегшує обмін повідомленнями між різними системами або компонентами в рамках розподіленої архітектури [1]. Основні функції брокера повідомлень включають:

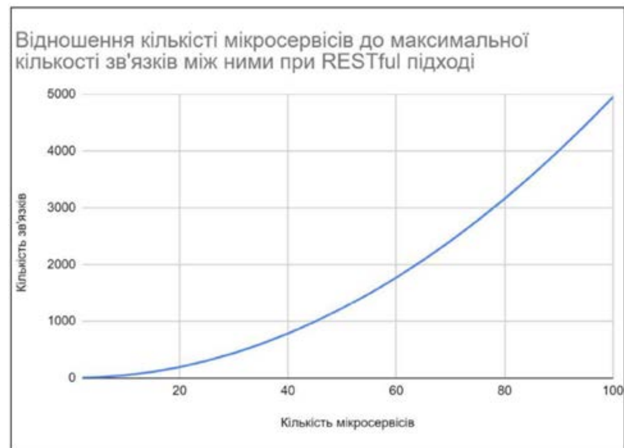

Рис. 5. Графік відношення кількості мікросервісів до максимальної кількості зв'язків між ними

– Маршрутизація повідомлень: Прийом повідомлень від відправників і пересилання їх до відповідних одержувачів.

– Асинхронність: Відправники можуть надсилати повідомлення без необхідності чекати на відповідь, що дозволяє компонентам працювати незалежно один від одного.

– Буферизація: Зберігання повідомлень до тих пір, поки одержувачі не будуть готові їх отримати.

– Перетворення повідомлень: Зміна формату повідомлень для сумісності між різними системами.

– Надійність і гарантія доставки: Забезпечення гарантованої доставки повідомлень навіть у випадку збою системи або мережі.

Прикладами брокерів повідомлень є RabbitMQ, Apache Kafka, ActiveMQ та Microsoft Azure Service Bus. Вони використовуються для побудови масштабованих і надійних систем, де компоненти можуть працювати незалежно і асинхронно обмінюватися даними [7].

Мікросервіс, що відправляє повідомлення брокеру, прийнято називати постачальниками. Будь-яка кількість постачальників має змогу відправляти повідомлення в певну чергу, так само і будь-яка кількість мікросервісів-підписників може вичитувати повідомлення з черги брокера [1]. Типова схема передачі даних за допомогою підписок і брокеру повідомлень зображена рис. 6.

Використання асинхронного способу до реалізації комунікації між мікросервісами має кілька переваг над синхронним підходом:

– Мала зв'язаність. Кожний мікросервіс залежить лише від брокера повідомлень. Саме тому кількість зв'язків у системі завжди буде дорівнювати кількості мікросервісів (див. рис. 7), а графік

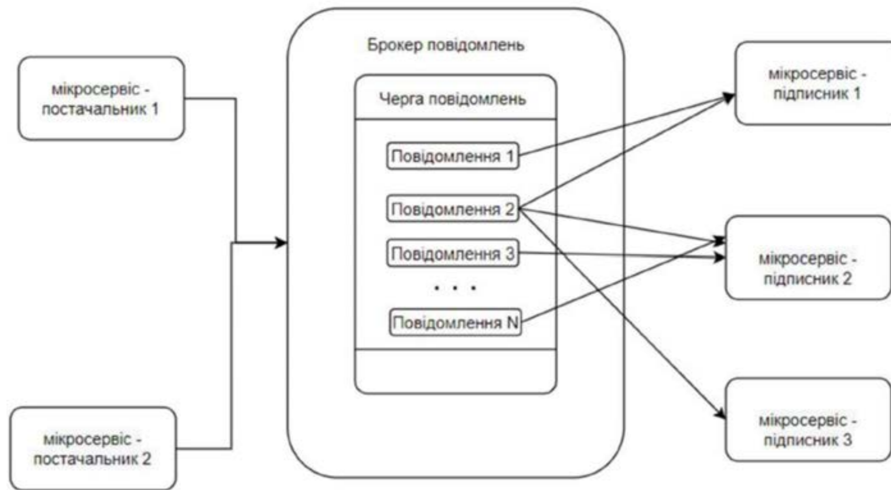

Рис. 6. Схема прикладу взаємодії мікросервісів із використанням брокеру повідомлень

відношення кількості мікросервісів до зв'язків у системі матиме вигляд лінійної функції f(x) = x, де x – це кількість мікросервісів, f(x) – кількість зв'язків (див. рис. 8).

– Відсутність блокувань. Мікросервіс-постачальник надіславши повідомлення не повинен чекати допоки це повідомлення буде оброблене мікросервісом-підписником.

– Здатність до масштабування системи. З подієорієнтованим підходим дуже легко проводити впровадження нового мікросервісу в систему, адже йому не потрібно зав'яхуватись на особливості всіх існуючих мікросервісів.

Однак, асинхронна комунікація мікросервісів також має свої недоліки:

– Неочевидність порядку виконання бізнес процесу. Мікросервіси комунікують через брокер повідомлень, а не напряму один з одним, тому стає важко прослідкувати послідовність роботи мікросервісів для певного бізнес-процесу.

– Кінцева узгодженість. З використанням асинхронного методу взаємодії можлива ситуація, коли стан системи є неузгоджений в контексті різних мікросервісів. Однак, коли всі події, необхідні для виконання певного бізнес процесу, будуть оброблені, то стан системи знову стане узгодженим.

– Складність у налагодженні. Оскільки порядок виконання бізнес процесу є неочевидний, то і виконувати процес налагодження складніше.

Для порівняння методів взаємодії між мікросервісами було розроблено 2 прототипи програмного продукту системи моніторингу роботи працівників. Продукт складається із клієнтського додатку для працівника та серверної частини. Основними функціональними вимогами для клієнта є:

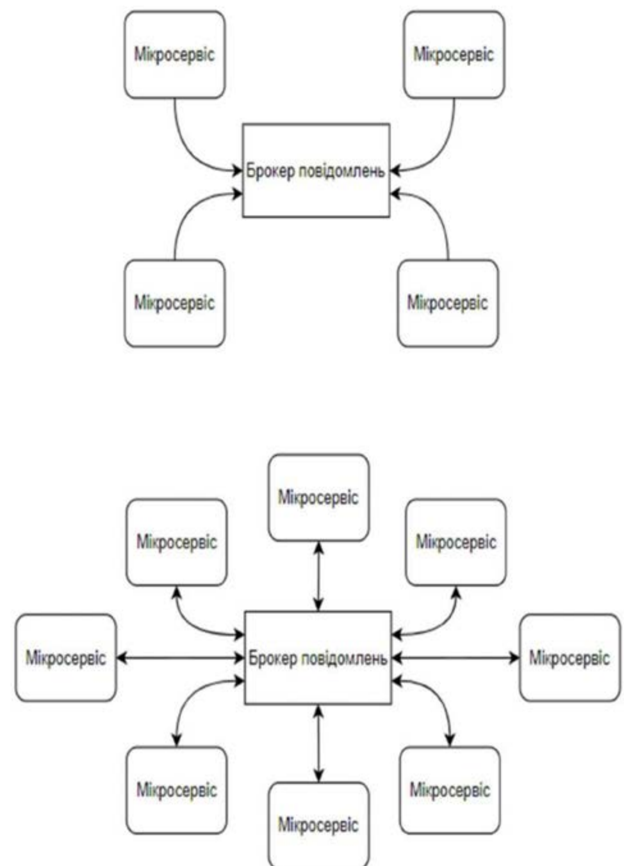

Рис. 7. Схема зв'язків між мікросервісами при подіє-орієнтованому підході

– авторизація працівника;

– онлайн трекер робочого часу;

– надсилання знімків екрану з робочого комп'ютера працівника на сервер;

– відстеження бездіяльності працівника.

Функціональні вимоги до серверної частини:

– збереження даних по обліку часу з клієнтів;

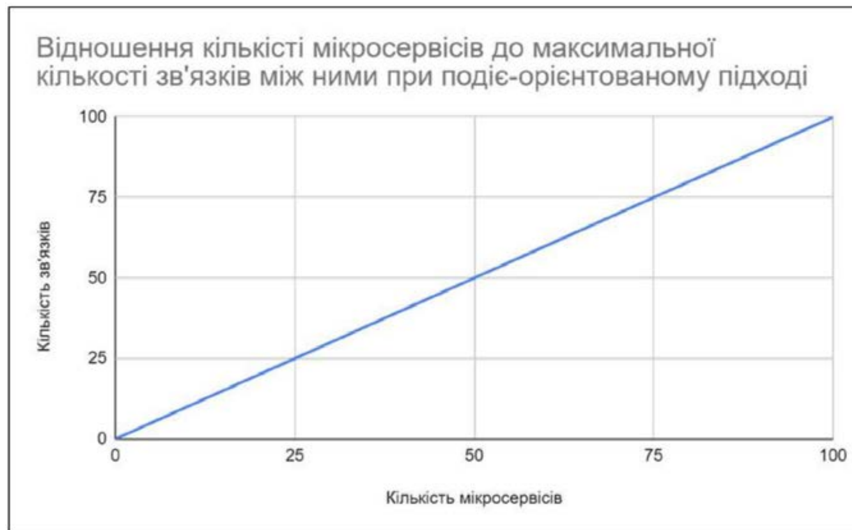

Рис. 8. Графік відношення кількості мікросервісів до максимальної кількості зв'язків у системі при подіє-орієнтованому підході

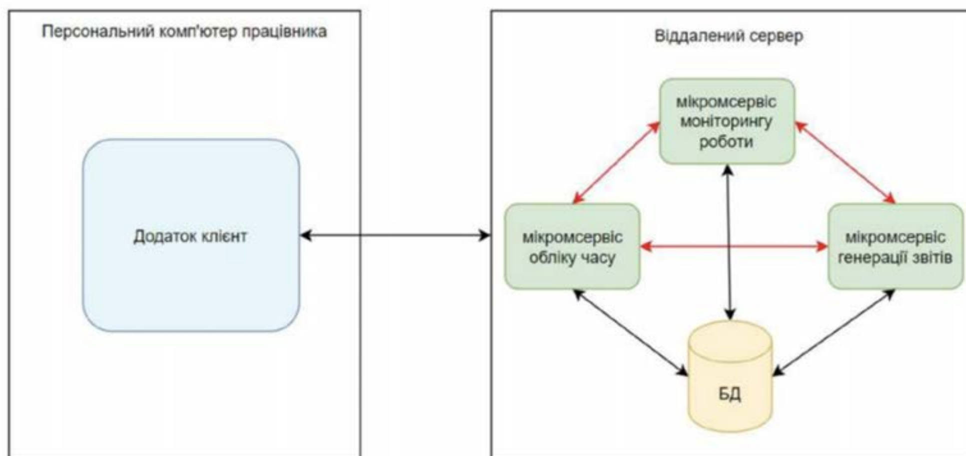

Рис. 9. Діаграма архітектури СМРПП із використанням синхронного підходу зв'язку між мікросервісами

– збереження та перегляд знімків екрану;

– наявність веб інтерфейсу для адміністрування;

– генерація звітів про роботу працівників за різний період часу;

– додавання, редагування, та видалення працівників.

Для реалізації програмного продукту, було використано наступні програмні засоби:

1. База даних: PostgreSQL
2. Мова програмування: Java
3. Фреймворки: JavaFX (для клієнта), Spring Boot (для сервер)
4. Бібліотеки: Spring Data, Spring WEB, Spring AMQP (тільки для асинхронної взаємодії), Spring MVC, Spring Security, Thymeleaf
5. Брокер повідомлень: RabbitMQ (тільки для асинхронної взаємодії)
6. Середовища розробки: PgAdmin, IntelliJ IDEA
7. Інструменти для тестування: Apache JMeter, IntelliJ IDEA Profiler

Перший продукт складається із клієнтського додатку та серверної частини, що являє собою структуру мікросервісів, реалізовану за допомогою синхронної взаємодії між мікросервісами, та базу даних PostgreSQL (див. рис. 9). Комунікація між мікросервісами була реалізована за допомогою протоколу HTTP, що надає підтримку таких типів запитів як GET, POST, PUT, DELETE, і т.д., що забезпечують ресурсорієнтовану систему обміну повідомленнями [8]. Функції HTTP

можуть бути повністю використані в архітектурі REST, включаючи кешування, аутентифікацію та узгодження типу вмісту [4].

Другий продукт також містить із додаток-клієнт та серверної частини, що являє собою структуру мікросервісів, асинхронна взаємодія яких реалізована за допомогою брокера повідомлень RabbitMQ (див. рис. 10). Даний брокер використовує широкорозповсюдженого протокол передачі даних AMQP – Advanced Message Queue Protocol (з англ. розширений протокол черги повідомлень). Даний протокол виник у фінансовій галузі, як правило використовує надійний транспортний протокол TCP/IP для передачі повідомлень [2]. AMQP забезпечує асинхронну комунікацію за допомогою публікації/підписки повідомлення. Його головна перевага – це функція зберігання та передачі даних, яка забезпечує надійність навіть після збоїв в роботі мережі [10].

Було проведено стрес-тестування сценарію по обробці події про початок роботи працівника серверної частини обох прототипів програмного продукту. Діаграми послідовності даного процесу в синхронній та асинхронній реалізації наведені на рис. 11 та на рис. 12 відповідно. Під час тестування серверний додаток був розгорнутий на

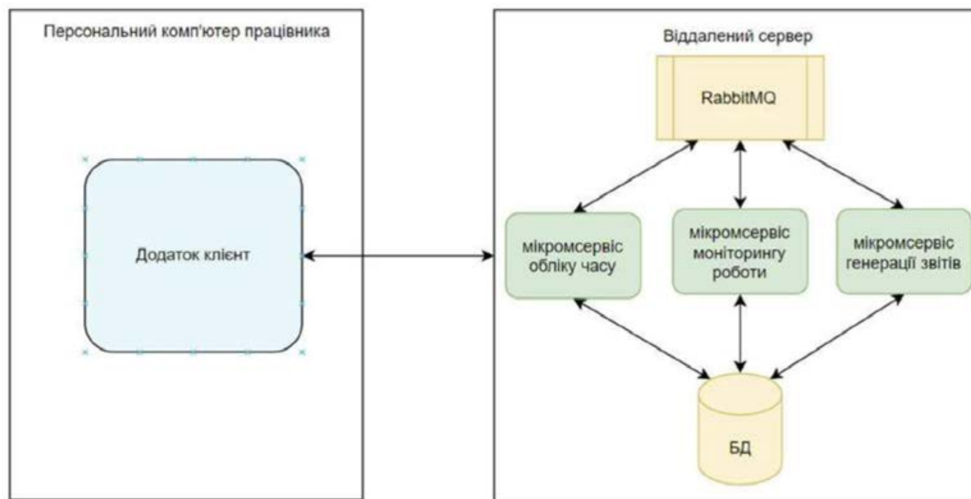

Рис. 10. Архітектура ПП із асинхронною зв'язками між мікросервісами

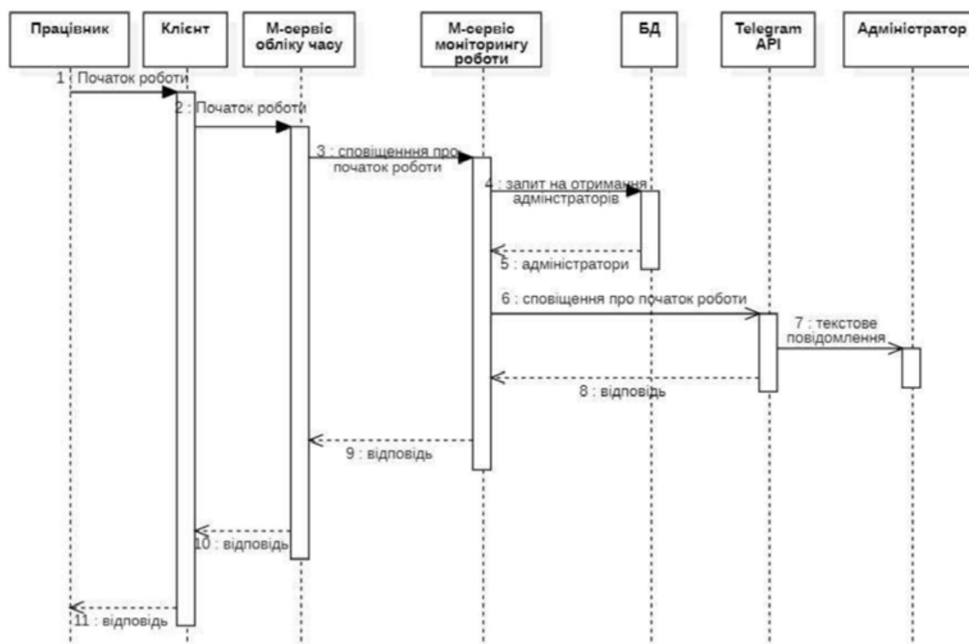

Рис. 11. Діаграма взаємодії процесу обробки початку роботи працівника із використанням синхронної взаємодії між мікросервісами

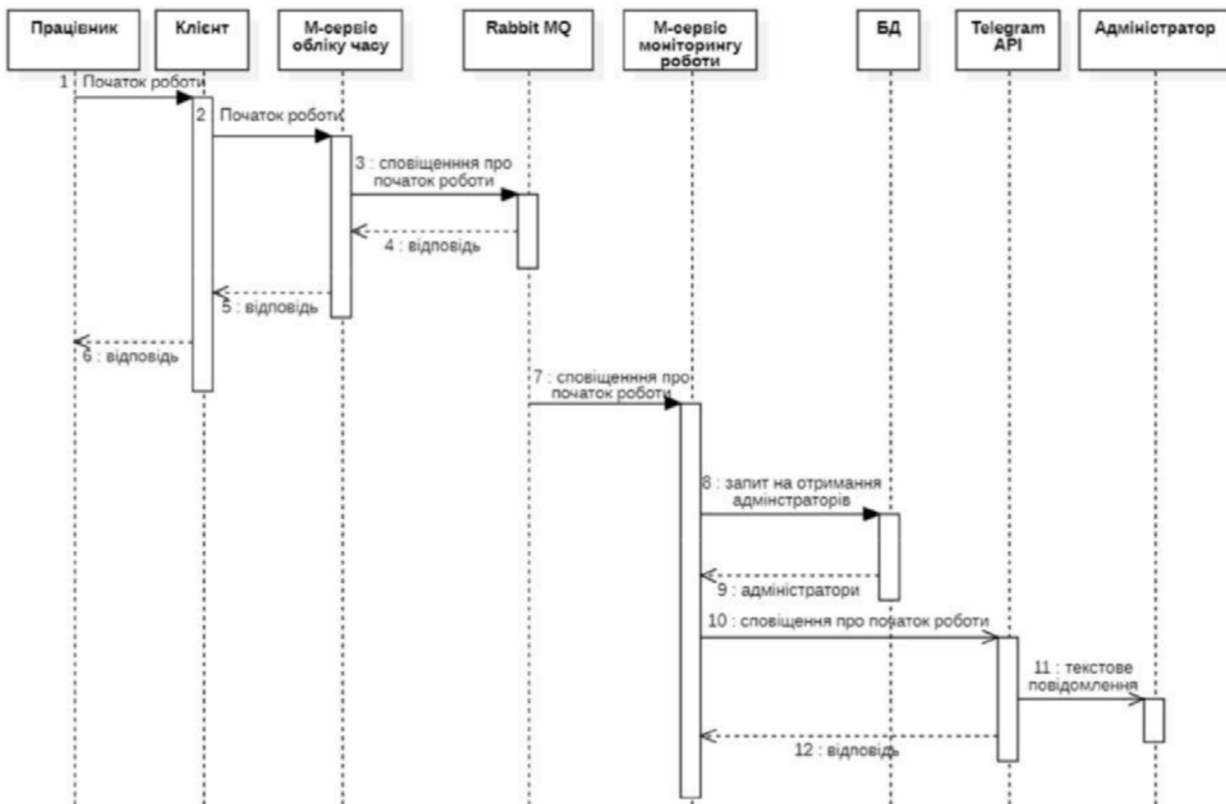

Рис. 12. Діаграма взаємодії процесу обробки початку роботи працівника із використанням асинхронної взаємодії між мікросервісами

системі із операційною системою Windows 10, центральним процесором Intel Core i5-8625U та кількість ОЗП – 8ГБ.

Для автоматизації процесу тестування було обрано програмний засіб JMeter від компанії Apache – це безкоштовний додаток, що дозволяє створювати та виконувати тестові сценарії для проведення тестування навантаженої веб систем, симулюючи поведінку багатьох користувачів в один момент часу, а також отримувати статистику результатів тестування та зберігати її у вихідному CSV чи XML файлі.

Було написано автоматизовані тестові сценарій, для синхронного та асинхронного методу взаємодії. Кожен сценарій виконувався із симуляцією різної кількості користувачів системи: 10, 100, 500 та 1000. Під час виконання стрес-тестування було зібрано наступні метрики:

– Середня тривалість одного запиту.

– Загальний час обробки процесу для одного користувача.

– Використання оперативної пам’яті.

Отримані результати наведені та зображені графічно у відповідних таблицях 1–3 та діаграмах на рис. 13–15.

Таблиця 1

Середній час обробки одного запиту

Тип взаємодії / к-сть користувачів	10	100	500	1000
Синхронна взаємодія	1131 мс	3219 мс	6584 мс	8995 мс
Асинхронна взаємодія	185 мс	436.8 мс	1651 мс	2134 мс

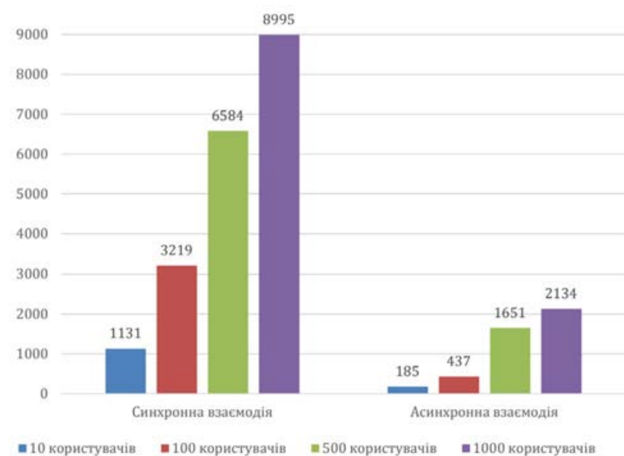

Рис. 13. Діаграма середньої тривалості запиту при синхронній та асинхронній взаємодії. Дані представлені у мілісекундах

Таблиця 2

Результати використання ОЗП

Тип взаємодії / к-сть користувачів	10	100	500	1000
Синхронна взаємодія	145 МБ	197 МБ	248 МБ	335 МБ
Асинхронна взаємодія	165 МБ	174 МБ	187 МБ	201 МБ

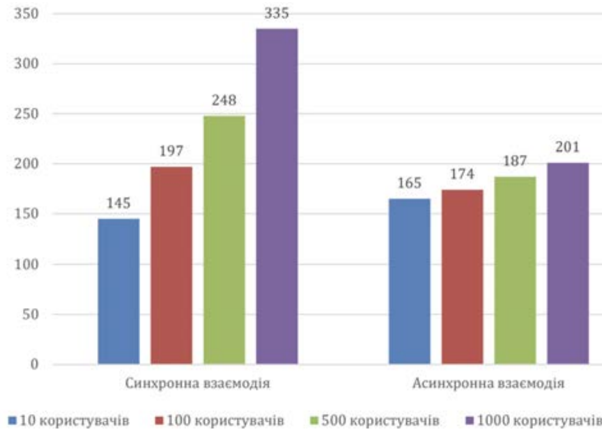

Рис. 14. Діаграма використання ОЗП при синхронній та асинхронній взаємодії. Дані представлені у мегабайтах

Таблиця 3

Загальний час обробки всіх запитів

Тип взаємодії / к-сть користувачів	10	100	500	1000
Синхронна взаємодія	1.387 с	18.74 с	64.952 с	117.43 с
Асинхронна взаємодія	1.106 с	16.84 с	122.95 с	258.9 с

Проаналізувавши отримані дані тестування, бачимо, що розроблений ПП на основі асинхронної взаємодії споживає менше системних ресурсів, однак програє у швидкодії аналогічному ПП, що використовує синхрону взаємодію, коли існує понад 100 користувачів, що одночасно взаємодіють із програмним продуктом.

Висновки. В роботі досліджено використання мікросервісного підходу в архітектурі програмного забезпечення, проаналізовано типи комунікацій між мікросервісам, визначено переваги та недоліки синхронного та асинхронного методів взаємодії мікросервісів.

Розроблено програмні продукти системи моніторингу роботи працівників підприємства із використанням синхронного та асинхронного методів комунікації між мікросервісами. Проведено стрес-тестування двох розроблених програмних продуктів за допомогою Apache JMeter та IntelliJ IDEA Profiler.

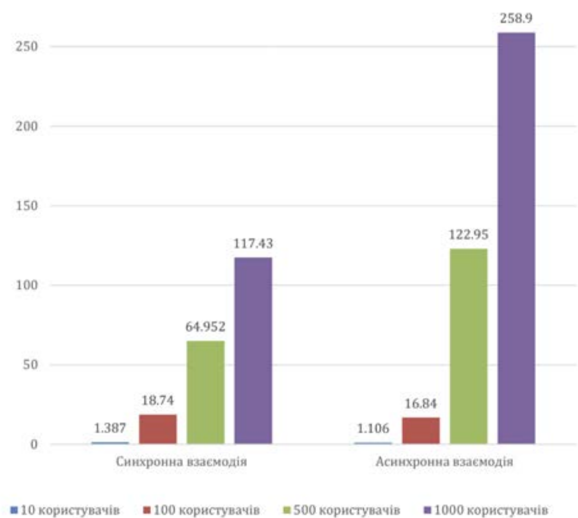

Рис. 15. Діаграма тривалості обробки процесу для всіх користувачів при синхронній та асинхронній взаємодії. Дані представлені у секундах

На основі отриманих результатів, можна зробити висновок, що асинхронний підхід до побудови програмного забезпечення матиме перевагу над синхронним, коли основною вимогою до ПЗ є обробка великої кількості однотипних процесів із мінімальною затратою системних ресурсів. Однак, якщо пріоритетом для програмного продукту є його швидкодія, то доцільно буде використовувати синхронний метод взаємодії між мікросервісами.

Таким чином, для системи моніторингу роботи працівників підприємства краще буде використати асинхронний підхід до зв'язку між мікросервісами, оскільки підтримка великої кількості працівників має важливе значення та є більш вагомою, ніж швидкодія виконання бізнес процесу. Також перевагою асинхронного підходу в даному випадку є менше споживання системних ресурсів, що дозволяє зменшити витрати коштів на оренду віртуального сервера чи хмарного середовища, які будуть використовуватися для розгортання програмного продукту.

Список літератури:

1. Bellemare A. Building Event-Driven Microservices: Leveraging Organizational Data at Scale 1st Edition. Себастопол: O'Reilly Media, Inc., 2020. 322 с.

2. Videla A., Williams J. RabbitMQ in Action. Нью-Йорк: Manning Publications Co., 2012. 558 с.
3. Thomas S. HTTP Essentials: Protocols for Secure, Scaleable Web Sites. Лондон: Wiley, 2001. 336 с.
4. Pollard B. HTTP/2 in Action. Нью-Йорк: Manning Publications Co., 2019. 416 с.
5. Webber J., Parastatidis S., Robinson I. REST in Practice. Себастопол: O'Reilly Media, Inc., 2010. 446 с.
6. Carnell J., Sanchez I. Spring Microservices in Action. Нью-Йорк: Manning Publications Co., 2021. 448 с.
7. AMQP 0-9-1 Model Explained. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html. (дата звернення: 18.06.2024).
8. REST Architectural Constraints. URL: https://restfulapi.net/rest-architectural-constraints/. (дата звернення: 21.06.2024).
9. Newman S. Building Microservices: Designing Fine-Grained Systems. Себастопол: O'Reilly Media, Inc., 2021. 500 с.
10. Spring AMQP. URL: https://spring.io/projects/spring-amqp. (дата звернення: 29.06.2024).

Tarnovetska O.Yu., Osadchuk R.R. RESEARCH OF INTERACTION METHODS BETWEEN SERVICES IN MICROSERVICE SOFTWARE ARCHITECTURE

The main objective of this article was to study the existing methods of interaction between the components of a software product created using microservices architecture. The authors conducted a detailed analysis of various protocols used for microservices interaction. During the research, the general structure of these protocols, the main characteristics of their working algorithms, data transmission processes, and methods of application in the context of microservices architecture were determined. Special attention was given to protocols such as HTTP/REST and AMQP, which provide different levels of performance, reliability, and ease of use.

To demonstrate the capabilities of microservices architecture and various methods of integration between services, software was developed for monitoring the working hours of subordinates in an enterprise. This software includes a set of microservices, each performing a specific task related to monitoring, collecting, and analyzing data on employees' working hours.

One of the key stages of the research was to determine the most optimal interaction protocol between the components of the software product. For this purpose, a series of test cases was created to model different usage scenarios of the software in a real enterprise environment. Based on these test cases, stress testing was conducted, which allowed evaluating the behavior of the software prototypes under different loads. This included simulating high loads, checking the stability of operation during prolonged runs, and assessing the system's fault tolerance.

The data obtained from the testing was thoroughly analyzed, and the identified patterns were substantiated. Based on the analysis, it was determined which protocols and interaction methods ensure the highest performance, reliability, and stability of the software in real-world use conditions. These results formed the basis for formulating recommendations for choosing methods of integrating microservices during the planning and development of a new software product.

Key words: *microservice, protocol, system, interaction, event, data transmission.*