Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури

Виконав студент 5 курсу, групи 1КІ-22м
спеціальності 123 — Комп'ютерна інженерія
Грядченко А. О.
Керівник к.т.н., доц. каф. ОТ
Кадук О. В.
«11» 12 2023 р.
Опонент к.т.н., доц. каф. МБІС
Грицак А. В.
«_____» __________ 2023 р.

Допущено до захисту
Завідувач кафедри ОТ
_______ д.т.н., проф. Азаров О.Д.
«15» 12 2023 р.

ВНТУ 2023

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Галузь знань — Інформаційні технології
Освітній рівень — магістр
Спеціальність — 123 Комп'ютерна інженерія
Освітн я програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри обчислювальної техніки
д. т. н, проф. О. Д. Азаров
"26 09" вересня 2023 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту **Грядченку Антону Олексійовичу**

1 Тема роботи «Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури» керівник роботи Кадук Олександр Володимирович к.т.н., доцент, затверджено наказом вищого навчального закладу від 18.09.2023 року **№ 18.09.2023 р.**

2 Строк подання студентом роботи **13.12.2023 р**.

3 Вихідні дані до роботи: розподілені системи, організація асинхронної комунікації, маршрутизація повідомлень за метаданими

4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): вступ, аналіз актуальності теми, розповсюдження розподілених систем, наявні методи асинхронної комунікації, можливість маршрутизації повідомлень.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): технічне завдання, сценарії використання інформаційної технології, лістинг програми.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти розділів

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Кадук Олександр Володимирович к.т.н., доцент		
5	Небава Микола Іванович к.е.н, професор		

7 Дата видачі завдання **19.09.2023 року.**

8 Календарний план виконання МКР приведений в таблиці 2.

Таблиця 2 — Етапи МКР

№ з/п	Назва етапів МКР	Строк виконання	Підпис
1	Постановка задачі	**19.09.2023**	
2	Аналіз актуальності теми	**02.10.2023**	
3	Аналіз наявних способів асинхронної взаємодії	**09.10.2023**	
4	Моделювання сценаріїв використання	**16.10.2023**	
6	Вибір технологічного стеку	**23.10.2023**	
7	Створення коду прототипу інформаційної системи	**30.10.2023**	
8	Розрахунок економічної частини	**06.11.2023**	
9	Оформлення пояснювальної записки та ілюстративного матеріалу	**13.11.2023**	
10	Виконання магістерської кваліфікаційної роботи	**20.11.2023**	
11	Перевірка якості виконання магістерської кваліфікаційної роботи та усунення недоліків	**27.11.2023**	
12	Підписи супроводжувальних документів у керівника, опонента, нормоконтролера	**10.12.2023**	
13	Перевірка «антиплагіат»	**11.12.2023**	
14	Попередній захист	**15.12.2023**	

Студент Грядченко Антон Олексійович

Керівник к.т.н., доц. Кадук Олександр Володимирович

АНОТАЦІЯ

УДК 004.75.032.34

Грядченко А. О. Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна Інженерія, Вінниця: ВНТУ, 2023 — 103 с. Українською мовою. Бібліогр.: рис.: 17; табл. 7.

У роботі розглянуто способи асинхронної комунікації у розподілених системах. В рамках дослідження запропоновано спосіб організації асинхронної взаємодії між компонентами розподіленої системи що дозволяє інтегрувати різні інфраструктурні рішення у систему без втручання у код додатку. Запропоновано прототип інформаційної системи що надає уніфікований програмний інтерфейс для взаємодії з різними типами черг та систем повідомлень. В рамках прототипу запропоновано механізм маршрутизації повідомлень, який дозволяє гнучко розподіляти повідомлення в залежності від певних умов з метою більш оптимального використання обчислювальних ресурсів.

Ключові слова: асинхронна комунікація, брокер повідомлень, повідомлення, розподілена система.

ABSTRACT

УДК 004.75.032.34

Griadchenko Anton. Information Technology of Asynchronous Interaction in a Distributed System Based on Microservice Architecture. Master's qualification work in specialty 123 — Computer Engineering, Vinnytsia: VNTU, 2023 — 103 pages. In Ukrainian. Bibliography: figures: 17; tables: 7.

This work discusses methods of asynchronous communication in distributed systems. Within the scope of the research, a method of organizing asynchronous interaction between components of a distributed system is proposed, allowing the integration of various infrastructure solutions into the system without interfering with the application code. A prototype of an information system is proposed, providing a unified software interface for interaction with different types of queues and messaging systems. Within the prototype, a message routing mechanism is suggested, allowing the flexible distribution of messages depending on certain conditions to optimize the use of computational resources more effectively.

Key words: asynchronous communication, message broker, message, distributed system.

ЗМІСТ

АНОТАЦІЯ **4**

ВСТУП **9**

1 АНАЛІЗ ТЕХНОЛОГІЙ КОМУНІКАЦІЇ У РОЗПОДІЛЕНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ ПОБУДОВАНИХ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ **11**

1.1 Огляд сучасного стану розподілених комп'ютерних систем 11

1.2 Аналіз підходів у комунікації у розподілених комп'ютерних системах 15

1.3 Основні методи асинхронної комунікації 18

1.3.1 Web-hook (Зворотній виклик) 18

1.3.2 Використання спільного сховища 20

1.3.3 Обмін повідомленнями 22

1.3.4 Комбінація різних способів комунікації 24

1.4 Гарантії доставки повідомлень у асинхронних методах комунікації 26

1.5 Формулювання проблемних питань та завдань дослідження 29

2 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ІНФОРМАЦІЙНОЇ СИСТЕМИ ДЛЯ АСИНХРОННОЇ ВЗАЄМОДІЇ У РОЗПОДІЛЕНІЙ КОМП'ЮТЕРНІЙ СИСТЕМІ **31**

2.1 Виклики асинхронних методів взаємодії між компонентами розподіленої комп'ютерної системи 31

2.2 Вимоги до компоненту асинхронної комунікації у розподілених комп'ютерних системах 32

2.3 Обрані критерії ефективності 37

3 ПРОГРАМНІ ЗАСОБИ ДЛЯ РЕАЛІЗАЦІЇ КОМПОНЕНТУ 'АСИНХРОННОЇ КОМУНІКАЦІЇ У РОЗПОДІЛЕНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ **39**

3.1 Вибір технологічного стеку 39

3.1.1 Java 40
3.1.2 C# 41
3.1.3 Node.js (JavaScript) 43
3.1.4 Golang 44
3.1.5 C++ 45
3.1.6 Rust 47
3.1.7 Python 48
3.1.8 Узагальнення та вибір мови програмування 49
3.2 Розробка архітектури програмного компонента 51
3.2.1 Проектування предметної області програмного компонента 51
3.2.2 Сценарії використання 53
3.3 Реалізація компонента асинхронної взаємодії з розподіленою системою 57
3.3.1 Конфігурація компонента асинхронної взаємодії. 57
3.3.2 Ключові типи, що використовуються у компоненту асинхронної комунікації 59
3.3.3 Ключові методи компоненту асинхронної комунікації 63
4 ВЕРИФІКАЦІЯ ТА ДОСЛІДЖЕННЯ КОМПОНЕНТУ АСИНХРОННОЇ КОМУНІКАЦІЇ 67
4.1 Методики тестування та верифікації асинхронної взаємодії 67
4.2 Верифікація сумісності з різними інфраструктурними компонентами .. 72
4.3 Верифікація можливості маршрутизації повідомлень 77
5 ЕКОНОМІЧНА ЧАСТИНА 81
5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки 81
5.2 Розрахунок витрат на проведення науково-дослідної роботи 83
5.2.1 Витрати на оплату праці 83
5.2.2 Відрахування на соціальні заходи 85
5.2.3 Сировина та матеріали 85
5.2.4 Розрахунок витрат на комплектуючі 86

5.2.5 Спецустаткування для наукових (експериментальних) робіт 86
5.2.6 Програмне забезпечення для наукових (експериментальних) робіт 86
5.2.7 Амортизація обладнання, програмних засобів та приміщень 87
5.2.8 Паливо та енергія для науково-виробничих цілей 87
5.2.9 Службові відрядження 88
5.2.10 Витрати на роботи, які виконують сторонні підприємства, установи і організації 89
5.2.11 Інші витрати 89
5.2.12 Накладні (загальновиробничі) витрати 89
5.3 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором 91
ВИСНОВКИ 96
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 98
ДОДАТОК А Технічне завдання **.......... 100**
ДОДАТОК Б Порівняльна таблиця мов програмування **.......... 104**
ДОДАТОК В Схема взаємодії при надсиланні повідомлення **.......... 105**
ДОДАТОК Г Схема взаємодії при підписці на повідомлення **.......... 106**
ДОДАТОК Д Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка **.......... 107**
ДОДАТОК Е Початковий код програми **.......... 108**
ДОДАТОК Ж Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень. **.......... 147**

ВСТУП

Актуальність теми, що розглядається у даному дослідженні обумовлена тим, що. сучасний світ IT переживає безпрецедентне зростання популярності розподілених систем, що стає фундаментальною основою для більшості технологічних процесів, від масштабних корпоративних мереж до хмарних обчислень.

Особливу увагу у цьому дослідженні приділено асинхронним методам комунікації, які набувають все більшої популярності завдяки своїм перевагам, таким як зниження затримок, підвищення ефективності обробки даних та здатності ефективно масштабуватися. Значущість асинхронних методів полягає у їх здатності підтримувати високу пропускну спроможність та надійність у умовах розподілених та складних систем.

Робота також охоплює різноманітність підходів до асинхронної комунікації, яка відображається у великій кількості доступних рішень, таких як Kafka, RabbitMQ, AWS SQS, AWS SNS, ZeroMQ та інші. Ця різноманітність свідчить про важливість та складність вибору оптимального рішення для конкретних бізнес-потреб та технічних вимог.

Нарешті, робота підкреслює потребу у спрощенні взаємодії з різними системами асинхронної комунікації. Така потреба обумовлена швидким розвитком технологій та необхідністю інтеграції різних систем у єдине цілісне рішення. Крім того, важливим аспектом є маршрутизація повідомлень у межах розподілених систем, що дозволяє оптимізувати процеси обробки даних та забезпечує ефективність та надійність комунікацій.

Метою дослідження є вдосконалення процесу розробки розподілених систем що використовують асинхронні методи комунікації шляхом розробки інформаційної технології, яка спрощує інтеграцію компонент розподіленої системи.

Об’єктом дослідження є методи та засоби асинхронної взаємодії між компонентами розподіленої системи, включаючи, але не обмежуючись, такими

як повідомлення в черзі (message queuing), подієве сповіщення (event notifications), та інші шаблони для підвищення відмовостійкості, масштабованості та незалежності сервісів у розподіленій системі.

Предметом дослідження є організація взаємодії між компонентом розподіленої системи та інфраструктурним способом яке забезпечує передачу та отримання повідомлення у асинхронний спосіб.

Новизна результатів полягає у тому, що покращено метод організації асинхронних комунікацій у розподілених системах, що дозволяє обирати оптимальне рішення для кожного конкретного випадку та комбінувати їх з метою досягнення оптимальних результатів.

Практичне значення даного дослідження полягає у тому, що запропонована інформаційна система асинхронної комунікації покращує процес інтеграції інфраструктурних рішень для обміну повідомленнями. Система дозволяє легко інтегрувати різні типи інфраструктурних рішень шляхом зміни конфігурації. Це дозволяє замінювати інфраструктурне рішення тим самим оптимізуючи витрати та зменшуючи ризики залежності від одного постачальника послуг.

Апробація теми даного дослідження відбувалась шляхом публікації на конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)» (м. Вінниця, 2023 р.)

1 АНАЛІЗ ТЕХНОЛОГІЙ КОМУНІКАЦІЇ У РОЗПОДІЛЕНИХ КОМП’ЮТЕРНИХ СИСТЕМАХ ПОБУДОВАНИХ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Огляд сучасного стану розподілених комп’ютерних систем

Комп’ютерні технології пройшли стрімкий шлях розвитку від першого програмованого комп’ютера ENIAC, який було створено у 40-х роках 20-го століття до сучасного часу, коли кількість обчислювальних пристроїв обчислюється мільярдами. Наприклад, згідно дослідження компанії Gartner, кількість персональних комп’ютерів, ноутбуків, планшетів та мобільних телефонів, що використовуються у світі у 2022 році має досягнути 6.4 мільярди пристроїв [1]. Всі ці пристрої використовуються користувачами для доступу до веб-додатків, які розміщені в мережі Інтернет, що вимагає від популярних додатків адаптуватись до зростаючого навантаження через ріст кількості користувачів.

Звичайно, для розгортання веб-додатків можна використовувати більш потужні сервери, але цього недостатньо, оскільки навіть найпотужніші сервери не здатні обслуговувати запити від користувачів популярних веб-додатків. Іншим підходом для масштабування є створення розподілених систем, які, завдяки своїм характеристикам, дозволяють створювати системи що можуть обслуговувати запити великої кількості користувачів.

Існує велика кількість досліджень, присвячених розподіленим системам. Різні дослідники по-різному визначають розподілені системи. Так, наприклад, Maarten van Steen та Andrew S. Tanenbaum визначають розподілені системи як набір незалежних комп’ютерів, які виглядають для користувача як єдина система [2]. Такі дослідники, як George Coulouris, Jean Dollimore, Tim Kindberg та Golangrdon Blair визначають розподілену систему як таку, де апаратні та програмні компоненти розміщені у комп’ютерах поєднаних у мережу та координують свої дії тільки шляхом обміну повідомленнями [3].

Проаналізуємо основні характеристики, які дозволяють розподіленим

системам задовільняти потребам сучасних веб-додатків.

Масштабованість це характеристика розподіленої системи що дозволяє використовувати переваги горизонтального масштабування, коли при великій кількості запитів до системи від користувачів у системі функціонує деяка кількість серверів, що дозволяє без перешкод обслуговувати потрібну кількість запитів від клієнтів. Крім того, горизонтальне масштабування дозволяє налаштувати систему таким чином, що кількість ресурсів (серверів) які обслуговує запити клієнтів адаптується до кількості запитів. Це дозволяє гнучко реагувати на зміну навантаження на веб-додаток: динамічно збільшувати кількість серверних ресурсів, які обслуговують запити користувачів у пікові періоди та зменшувати у періоди зменшення активності користувачів.

Прикладами таких періодів збільшення кількості запитів можуть бути:

— зміна кількості запитів у зв’язку зі зміною часу доби;

— сезоні зміни;

— збільшення навантаження у зв’язку з маркетинговими акціями що можуть призвести до зростання зацікавленості користувачів у веб-додатку.

Окрім більш якісного обслуговування запитів користувачів, гнучке масштабування дозволяє більш ефективно використовувати ресурси та зменшувати вартість інфраструктури під час спадів навантаження на веб-додаток за рахунок зменшення кількості серверних ресурсів, що використовуються.

Стан системи у період зменшення та збільшення кількості запитів зображено на рисунках 1.1 та 1.2 відповідно.

Рисунок 1.1 — Стан системи у період меншої кількості запитів

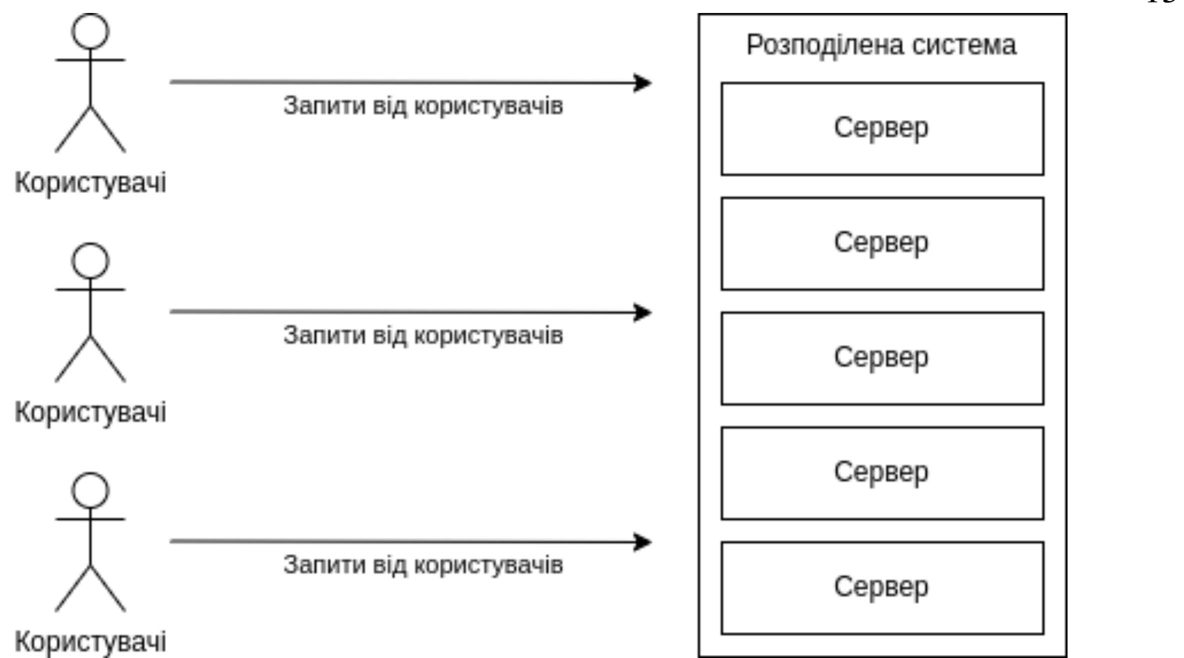

Рисунок 1.2 — Стан системи у період збільшення кількості користувачів

Централізовані системи є уразливими до відмов. У випадку збою в одному місці вся система може стати недоступною. Розподілені системи використовують реплікацію та декомпозицію, щоб забезпечити стабільність роботи навіть у випадку відмови декількох компонентів. На рисунку 1.3 зображена система де один з серверів знаходиться у несправному стані, проте в цілому система продовжує функціонувати та обслуговувати запити користувачів.

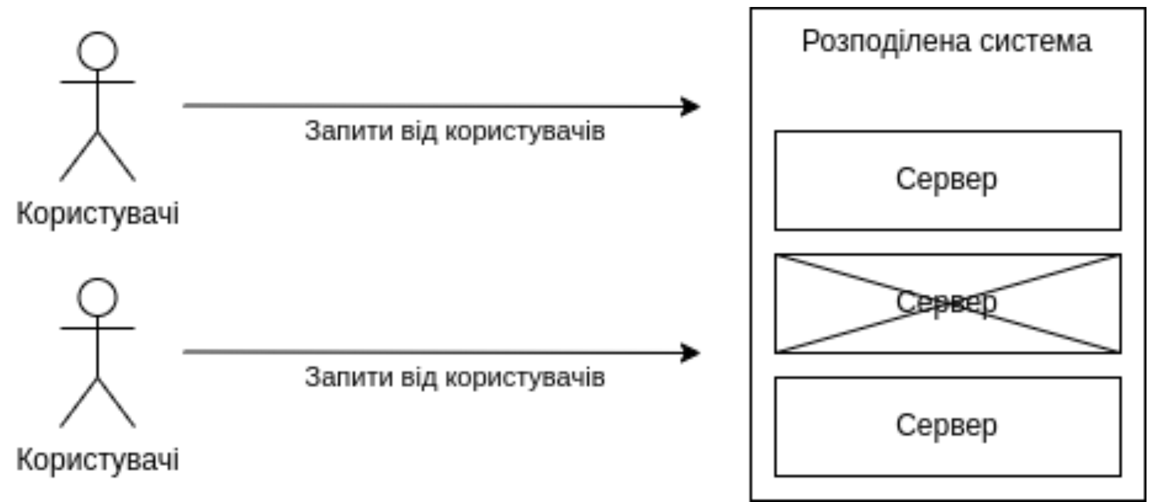

Рисунок 1.3 — Розподілена система у випадку відмови одного з компонент

Доступність системи характеризується її здатністю бути доступною для

використання в будь-який момент часу. Крім того, компоненти розподілених систем можуть бути розгорнуті у різних географічних регіонах що вкрай важливо для глобального бізнесу. Це дозволяє розмістити компоненти системи якнайближче до кінцевого користувача, тим саме зменшуючи фізичну відстань між користувачем та сервером, що, в свою чергу, покращує швидкодію додатку та покращує користувацький досвід. На рисунку 1.4 зображено схематично розподілену систему, компоненти якої розміщено у різних географічних регіонах, які знаходяться фізично ближче до кінцевого користувача.

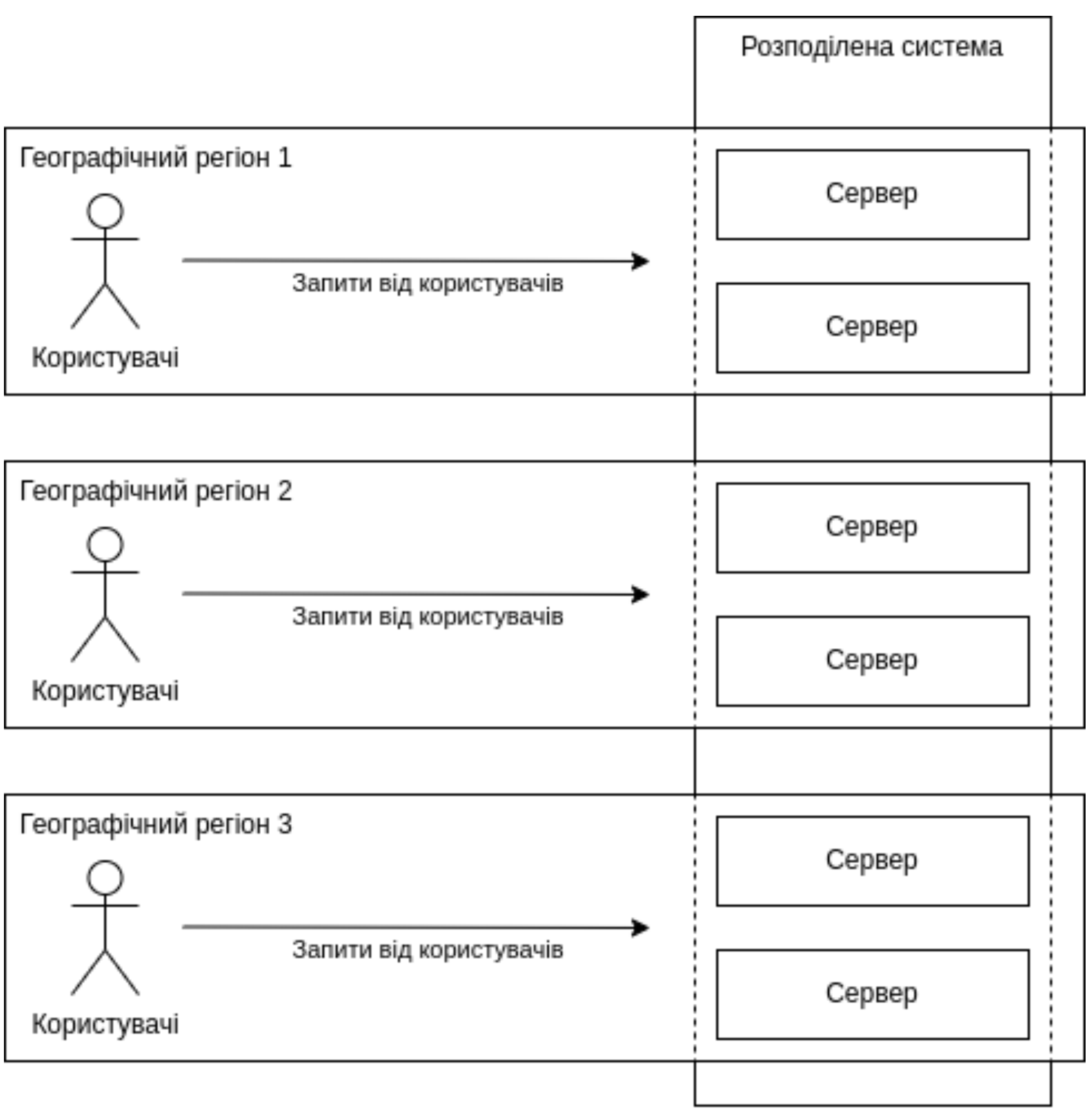

Рисунок 1.4 — Розподілена система у різних географічних регіонах

Використання розподілених систем дозволяє підвищувати безпеку веб-додатку за рахунок розділення відповідальності між різними компонентами та

надання кожному компоненту обмежених прав, що унеможливлює отримання інформації, що не є необхідною для роботи кожного окремого компоненту. Це дозволяє ізолювати різні компоненти та ускладнити доступ зловмиснику до інформації в системі в загалом. При отриманні неправомірного доступу до окремого компоненту системи зловмисник має доступ тільки до частини системи та, відповідно, частини інформації.

У розподілених системи дозволяють використовувати технологічну гнучкість, де кожен компонент може відповідати за свою окрему функцію. Це дозволяє робити компоненти сфокусованими на окремій функції та використовувати для реалізації цієї функції найбільш доречний інструменти: мову програмування, бібліотеку та інше. Наприклад для реалізації веб-сервісів доречно використовувати такі технології як Java, C#, Python, Node.js завдяки можливостям швидкої розробці які надають такі технології, а компонент системи який відповідальний за складні математичні обчислення доречно реалізувати з використанням таких технологій як C або C++ що дозволяє найбільш ефективно використати обчислювальні ресурси.

1.2 Аналіз підходів у комунікації у розподілених комп’ютерних системах

В розподілених системах комунікація між компонентами є ключовим фактором для ефективної роботи. Сучасні розподілені системи можуть складатись з сотень та тисяч компонент, які можуть бути розміщені у різних географічних регіонах, та вибір правильного способу комунікації може стати вирішальним фактором працездатності веб-додатку.

Існують різні способи комунікації, і вони можуть бути класифіковані за рядом критеріїв, одним з яких є синхронність.

Синхронна комунікація це спосіб комунікації між компонентами системи що передбачає, що відправник буде очікувати поки запит не буде прийнятий отримувачем. Дуже часто відправник також очікує на відповідь від отримувача

що додає до часу очікування також час, необхідний на обробку запиту, формування та надсилання відповіді.

Синхронні методи комунікації досить прості для використання та розуміння що робить їх привабливим варіантом для вибору як основного методу комунікації, проте вони також мають ряд недоліків, які необхідно враховувати. Серед таких недоліків можна виділити наступні:

— синхронні запити можуть бути менш ефективним за рахунок очікування відповіді від отримувача;

— синхронні методи комунікації вразливі до затримок, які можуть виникнути в процесі передачі інформації мережею що знижує швидкодію системи в цілому;

— синхронні запити передбачають, що отримувач у момент запиту має функціонувати, проте це може бути не завжди вірно, адже у момент запиту компонент отримувача може знаходитись у непрацездатному стані з багатьох причин, що призведе до виникнення помилки на стороні відправника.

Серед прикладних протоколів, які реалізують синхронні методи комунікації є такі протоколи як:

— HTTP — основний протокол функціонування інтернету та використовується для великої кількості веб-додатків в мережі Інтернет;

— GraphQL — протокол для зручних запитів на сервер який дозволяє гнучко отримувати потрібну інформацію з урахуванням фільтрації даних, які будуть отримані та зв’язків;

— gRPC — високопродуктивний бінарний протокол який має широку підтримку у різних мовах програмування;

— SOAP — протокол обміну повідомленнями з використанням мови розмітки XML для формування повідомлень, популярний у корпоративних застосунках.

Асинхронні методи комунікації передбачають, що відправник не очікує

відповіді від отримувача та може продовжувати функціонувати. Використання асинхронних методів комунікації ускладнює систему, адже вона має бути спроектована таким чином, щоб використовувати всі переваги асинхронних методів комунікації. Зазвичай у асинхронних методах комунікації використовується додатковий компонент який виступає посередником між компонентами системи. Серед переваг асинхронних методів комунікації можна виділити наступні.

— асинхронна комунікація дозволяє гнучко масштабувати систему, адже компоненти системи можуть обробляти повідомлення незалежно один від одного;

— асинхронні методи комунікації дозволяють розподіляти навантаження між компонентами за рахунок використання черг повідомлень;

— за рахунок використання черг повідомлень, системи побудовані на асинхронних методах комунікації можуть бути більш стійкими до збоїв, адже дозволяють зберігати повідомлення незалежно від працездатності решти системи та доставляти їх коли отримувач буде готовий обробити повідомлення.

Протоколи, що реалізують асинхронну комунікацію:

— AMQP (Advanced Message Queuing Protocol) — стандарт обміну повідомленнями для міжпроцесної комунікації;

— MQTT (Message Queuing Telemetry Transport) — протокол обміну повідомленнями, який зазвичай використовується у пристроях з обмеженими ресурсами або у мережах з низькою пропускною здатністю;

— Apache Kafka — високопродуктивна платформа для обробки потоків даних у реальному часі;

— STOMP (Simple Text Oriented Messaging Protocol) — простий текстовий протокол для обміну повідомленнями між клієнтами через шину повідомлень;

— WebSockets — протокол для двосторонньої комунікації через одне,

відкрите з'єднання, що дозволяє надсилати повідомлення з сервера до клієнта та навпаки без повторного встановлення з'єднання.

Підсумуємо основні риси синхронних та асинхронних методів комунікації у таблиці 1.1

Таблиця 1.1 — Основні риси синхронних та асинхронних методів комунікації

Синхронна комунікація	Асинхронна комунікація
Простий для використання.	Вимагає врахування особливостей асинхронної комунікації.
Проста архітектура, взаємодія відбувається напряму між відправником та отримувачем.	Зазвичай вимагає додаткового компоненту між відправником та отримувачем — черги повідомлень.
Вимагає доступності отримувача та, зазвичай, відповіді від отримувача.	На момент відправки повідомлення отримувач може бути недоступний, повідомлення може бути доставлено пізніше.
Може створити додаткові затримки за рахунок очікування відповіді від отримувача.	Відправник не очікує відповіді від отримувача та може продовжити роботу.
Приклади реалізації: HTTP, GraphQL, gRPC, SOAP	Приклади реалізації: AMQP, MQTT, Apache Kafka, STOMP, WebSockets

1.3 Основні методи асинхронної комунікації

Асинхронна комунікація між компонентами інформаційної системи може здійснюватись різними способами. Розглянемо основні з них.

1.3.1 Web-hook (Зворотній виклик)

Одним з найбільш простих способів асинхронної комунікації є використання веб-хуків. Веб-хук є по суті зворотнім викликом, який компонент, у якому відбувається подія, має викликати щоб повідомити інший компонент про те що подія настала. Зазвичай веб-хуки реалізуються за допомогою використання HTTP протоколу, коли зворотній виклик здійснюється шляхом надсилання HTTP

запиту після настання певної події. Для того щоб компонент, де відбувається подія, міг повідомити інший компонент зворотній виклик має бути зареєстровано заздалегідь.

Схематично це відображено на рисунку 1.5.

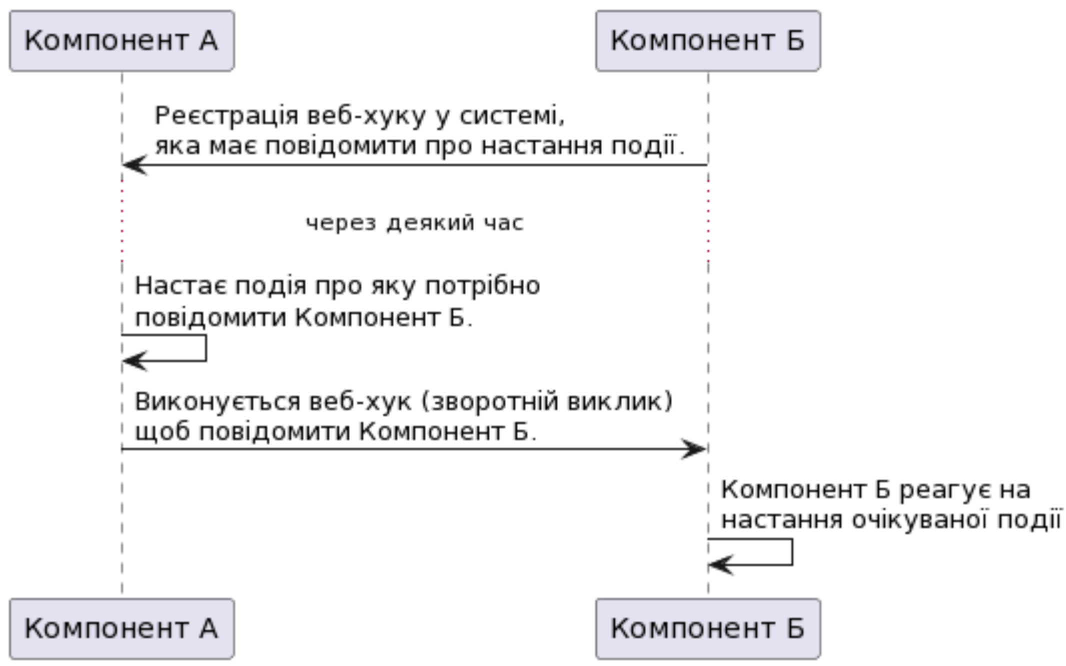

Рисунок 1.5 — Схема комунікації з використанням веб-хуку

Веб-хуки прості у реалізації. Для того, щоб прийняти веб-хук достатньо простого веб-серверу, який здатний обробити вхідний HTTP-запит. Крім того для обробки веб-запитів наявні вже напрацьовані технології для забезпечення безперервної роботи веб-серверу, такі як обмеження кількості викликів за певний проміжок часу, обмеження запитів з певних IP-адрес або мереж, фільтри які дозволяють заборонити потенційно небезпечні запити які містять шкідливі дані (наприклад SQL ін’єкції).

Проте при використанні веб-хуків є ряд недоліків, які слід враховувати при виборі цього методу асинхронної комунікації, а саме: під час виклику веб-хуку компонент, який потрібно повідомити про настання події, може бути недоступний і повідомлення може бути втрачено або має бути відправлено повторно, що вимагає додаткової обробки.

Веб-хуки ефективні для інтеграції з різними веб-сервісами та забезпечують

можливість швидко реагувати на зміни без необхідності постійного опитування зовнішніх сервісів. Це знижує затримки та інтенсивність використання ресурсів, проте вимагає ретельної обробки помилок та забезпечення безпеки передачі даних.

Веб-хуки широко використовуються у сучасній розробці для інтеграції компонент різних систем. Приклади застосування цього методу комунікації:

— сервіс для спільної роботи над початковим кодом GitHub.com використовує веб-хуки для таких задач як повідомлення зовнішніх систем безперервної інтеграції про додання нового коду на git-сервер, повідомлення про Pull Request, розгортання коду та інше [4];

— сервіс керування проектами Jira надає можливість реєстрації веб-хуків для повідомлення сторонніх компонентів про зміни у зареєстрованих задачах, настання певних подій та інше [5];

— сервіс для обміну повідомленнями (месенджер) Slack за допомогою веб-хуків дозволяє надсилати повідомлення у чат, наприклад повідомлення про невдалий запуск тестів на сервісі безперервної інтеграції або про новий звіт про помилку, заведений у Jira [6];

— реалізації протоколу авторизації OAuth 2.0 використовують веб-хук для передачі клієнту даних, необхідних для отримання токену для авторизації [7].

1.3.2 Використання спільного сховища

Іншим способом асинхронної комунікації між сервісами є використання спільного сховища — наприклад бази даних або файлового сховища. Суть цього способу полягає у тому, що компоненти, що комунікують між собою, мають доступ до спільного сховища даних, яким може бути база даних, файлове сховище або будь-який інший компонент який дозволяє зберігати дані. Для того щоб надіслати повідомлення один з компонентів має зберегти дані у спільному сховищі, а другий — прочитати з нього.

Схема організація комунікації з використанням спільного сховища

відображена на рисунку 1.6.

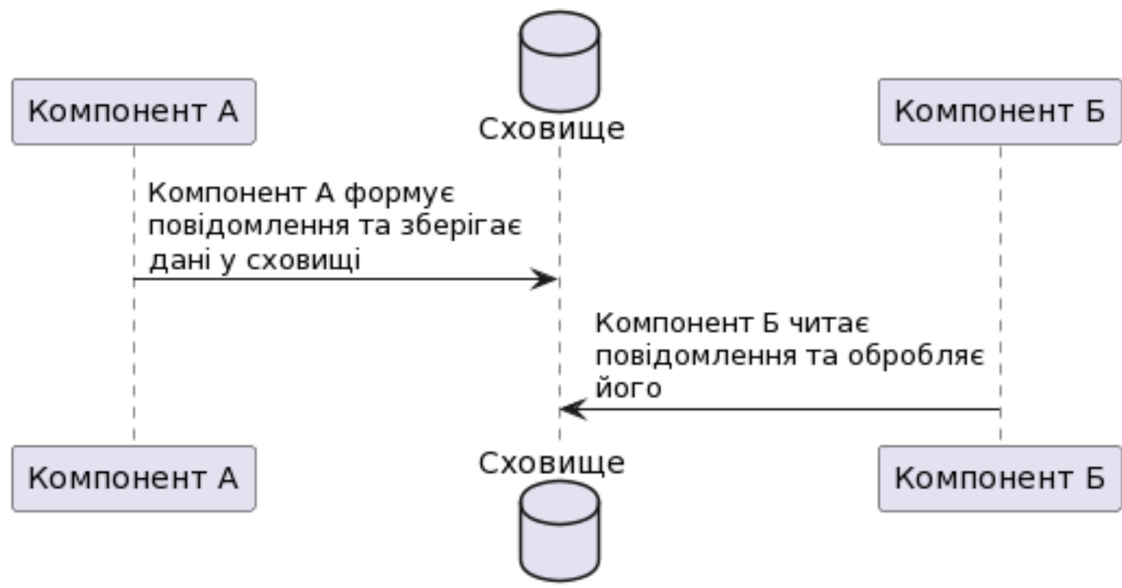

Рисунок 1.4 — Схема комунікації з використанням спільного сховища

Реалізація комунікації за допомогою використання спільного сховища може бути привабливим варіантом завдяки тому що його досить просто реалізувати з використанням стандартних механізмів роботи з файлами (у випадку використання файлового сховища) або з базою даних. Проте таких підхід має ряд недоліків.

Компонент, який читає дані має періодично перевіряти наявність нових даних у спільному сховищі, що робить використання цього методу не придатним для ситуацій, коли потрібно обробляти дані з найменшою затримкою [8].

При використанні файлового сховища важливо пам’ятати, що у момент перевірки наявності нових даних для обробки, вони можуть бути ще не остаточно записані на у сховище, тому може виникнути ситуація, коли для обробки повідомлення буде використана файл, що не було повністю записано, що може призвести до виникнення помилок та неконсистентного стану. Для уникнення подібних ситуацій необхідно розробити якусь систему сигналів, яка дасть зрозуміти компоненту який має прочитати дані, що вони були записані повністю та готові для читання.

Використання у якості спільного сховища бази даних надає можливість використовувати вбудовані механізми забезпечення цілісності даних, але у такому підході є свої недоліки. У такому випадку різні компоненти будуть мати залежність від однієї бази даних та зміна схеми бази даних буде вимагати відповідних змін у всіх залежних компонентах, що створить високу зв'язність компонент [8].

Проте даний спосіб організації комунікації між компонентами розподіленої системи має суттєву перевагу, якої позбавлені всі інші, а саме у випадку використання файлового сховища компоненти можуть обмінюватись великими обсягами даних, наприклад великими файлами, що може складно реалізувати при використанні інших методів комунікації, або взагалі неможливо зробити.

1.3.3 Обмін повідомленнями

Наступним способом асинхронної комунікації у розподілених системах, який буде розглянуто, буде обмін повідомленнями. При використанні цього методу компоненти системи обмінюються повідомленнями. Обмін повідомленнями здійснюється не безпосередньо між компонентами, а з використанням додаткової системи — брокера повідомлень. Схематично взаємодія відображена на рисунку 1.7

Рисунок 1.5 — Схема комунікації за допомогою обміну повідомленнями

Використання повідомлень для комунікації має ряд переваг, які роблять цей спосіб комунікації популярним та поширеним у використанні. Серед основних

переваг можна виділити наступні.

Завдяки використанню брокера повідомлень система може забезпечити надійність та гарантувати доставку повідомлення навіть якщо компонент-споживач повідомлення недоступний в даний момент за рахунок збереження повідомлення. Крім того, система побудована на обміні повідомленнями здатна обробляти повідомлення з меншою затримкою, порівняно з системами, побудованими на спільному сховищі даних, які було розглянуто в розділі **Error! Reference source not found.**.

Використання брокерів повідомлень дозволяє зменшити зв'язаність системи, оскільки компонент, що публікує повідомлення не знає нічого про компонент, який буде отримуватиме повідомлення. Ба більше — отримувачів повідомлення може бути декілька, тоді кожен з отримувачем є незалежним компонентом системи, що не знає про інші частини.

Вбудовані механізми брокерів повідомлень дозволяють реалізовувати додаткову логіку обробки повідомлень, таке як повторна доставка повідомлення, якщо під час обробки повідомлення сталась помилка, доставка повідомлення з затримкою, балансування між споживачами.

Безумовно, даний спосіб не позбавлено недоліків. Першим недоліком є те, що в розподіленій системі з’являється додатковий компонент, який потребує уваги — керування, моніторинг, оновлення, контроль за забезпеченням безпеки компоненту.

Крім того, при використанні цього методу потрібно враховувати, що зміни схеми повідомлення (зміна атрибутів повідомлення, зміна формату, кодування та інше) вимагає внесення відповідних змін до коду додатків, які обробляють повідомлення. У великих розподілених системах такі зміни не можуть відбуватись одномоментно:

— для серверних додатків можуть використовуватись поступове розгортання (rolling update), коли різні компоненти розподіленої системи розгортаються незалежно один від одного, а черги можуть містити повідомлення

старого формату.

— у випадку, якщо є клієнтські додатки, які споживають повідомлення з черги, тоді потрібно враховувати що вони можуть бути не оновлені вчасно, адже оновлення таких компонент знаходиться під контролем користувачів.

Це означає, що стара та нова версія схеми повідомлення можуть існувати в системі одночасно, та це призводить до потреби підтримки як зворотної сумісності, коли нова версія компоненту має коректно обробляти повідомлення старого формату, так і прямої сумісності, коли стара версія компоненту має коректно працювати з повідомленнями нового формату. [9]

1.3.4 Комбінація різних способів комунікації

Комбінація різних способів асинхронної комунікації дозволяє використати їх переваги та зменшити вплив недоліків.

Розглянемо варіант комбінації асинхронної комунікації побудованої на основі веб-хуків та з використанням черг повідомлень. Розглянемо схему зображену на рисунку 1.8.

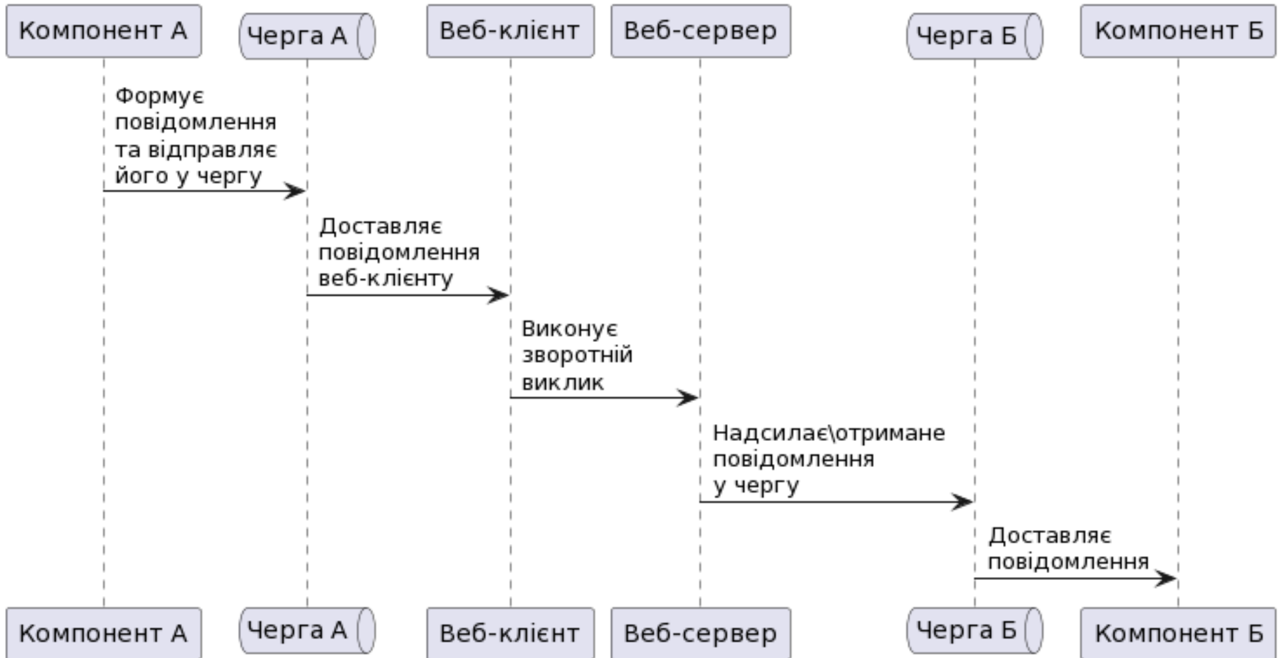

Рисунок 1.6 — Схема використання комбінації веб-хуків та черг повідомлень

Відповідно до цієї схеми компонент А замість того, щоб надіслати

повідомлення безпосередньо компоненту Б, надсилає його у чергу. Повідомлення з черги отримує веб клієнт, відповідальність якого полягає тільки у виклику веб-хуку.

Веб-сервер, який налаштовано очікувати веб-хук, надсилає повідомлення у іншу чергу, з якої повідомлення отримує компонент Б.

Веб-клієнт та веб-сервер можуть бути більш надійними за рахунок того, що вони виконують досить прості задачі та не будуть часто оновлюватись що робить систему загалом більш стійкою.

Використання черг у даній схемі дозволяє зберігати повідомлення на період коли компонент, який має його отримати (компонент Б відповідно до наведеного на схемі прикладу) може бути недоступний через розгортання нової версії або через помилку. Таке повідомлення буде доставлено коли компонент буде у працездатному стані.

Іншим прикладом комбінації різних способів комунікації, який відображає як поєднання різних способів комунікації дозволяє отримати більше переваг є поєднання комунікації через спільне сховище та черг. Схематично це відображено на рисунку 1.9.

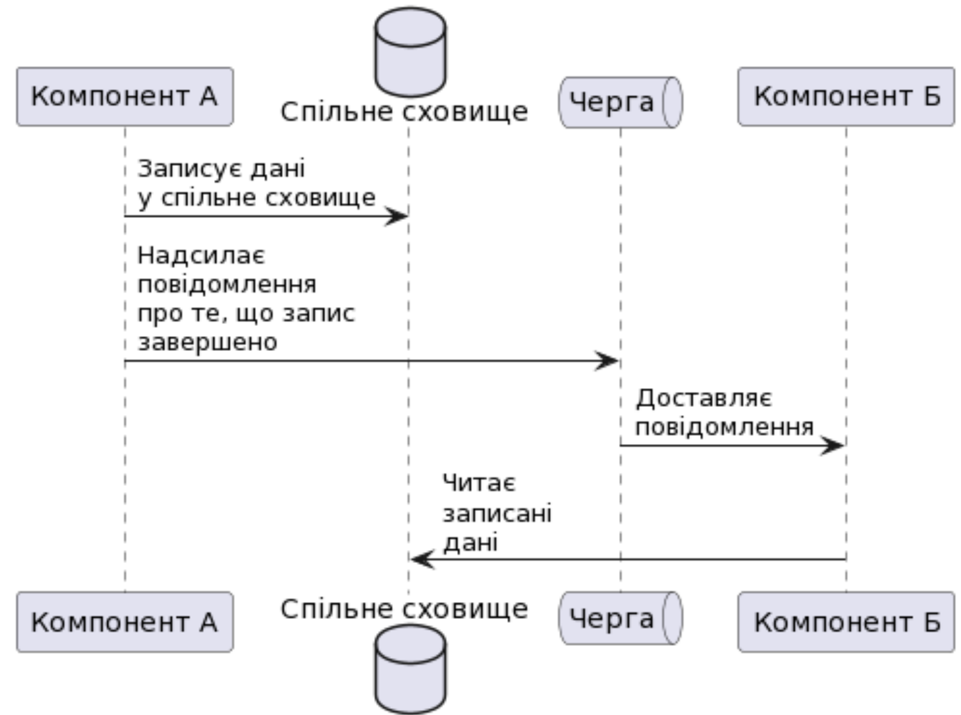

Рисунок 1.7 — Комунікації за допомогою спільного сховища та черг

При використанні поєднання комунікації за допомогою спільного сховища та повідомлень компонент А, який має надіслати дані, спочатку записує дані у сховище, а після успішного запису — надсилає повідомлення у чергу. Компонент Б отримує повідомлення з черги та має змогу отримати дані зі спільного сховища. При цьому, в момент отримання повідомлення, компонент Б прочитає цілісні дані, оскільки повідомлення було надіслано після успішного запису.

Така комбінація дозволяє з одного боку використати переваги способу комунікації з використанням спільного сховища, а саме — можливість обміну великими обсягами даних при використанні файлового сховища. З іншого боку використання черг повідомлень дозволяє нівелювати недоліки комунікації з використанням спільного сховища та по-перше при читанні бути впевненим, що дані записані повністю, а по-друге зменшити затримки при отриманні повідомлення, адже компонент, який читає данні більше не має періодично перевіряти сховище на наявність нових даних, а в замін цього отримає повідомлення.

Повідомлення про запис даних до спільного сховища (видалення, зміну або інші події) може бути відправлено механізмами самого сховища. Хмарні сервіси баз даних, такі як AWS DynamoDB, Google Bigtable та Azure Cosmos DB також надають можливість отримувати повідомлення про зміни даних, тим сами реагуючи на них з мінімальною затримкою [10, 11, 12].

Якщо йде мова про збереження даних великого обсягу (наприклад файлів), то хмарні сервіси для зберігання файлів у хмарі AWS S3, Google Cloud Storage та Azure Storage надають можливість отримати повідомлення про ряд подій які відбуваються з об’єктами які зберігаються [13, 14, 15].

1.4 Гарантії доставки повідомлень у асинхронних методах комунікації

Гарантії доставки повідомлень грають важливу роль у сучасних системах асинхронної комунікації, де надійність і коректність обміну даними мають велике

значення. Ці гарантії визначають рівень надійності, з якими повідомлення будуть доставлені від відправника до отримувача. Гарантії доставки ще називають «якісь обслуговування» (quality of service, QoS)

Зазвичай виділяють наступні рівні гарантій доставки повідомлень [16, 17].

Не більше одного разу (at most once). Якщо спосіб асинхронної комунікації надає такий рівень гарантій доставки повідомлення, то це означає, що воно буде доставлено один раз.

З точки зору відправника повідомлення, це найпростіша реалізація, адже вона передбачає, що повідомлення буде надіслано тільки один раз, без необхідності зберігати його та контролювати доставку повідомлення до отримувача. Це значно спрощує реалізацію та підвищує пропускну здатність.

Для отримувача це означає, що якщо після отримання повідомлення, але до завершення його обробки відбулась помилка та роботу додатку було завершено з помилкою тоді повідомлення не буде доставлено повторно.

Крім того, якщо під час надсилання повідомлення отримувач не був готовий його отримати, (наприклад компонент у цей момент не працює) або під час відправки повідомлення відбулась помилка на транспортному рівні (наприклад мережевий збій) тоді таке повідомлення буде втрачено.

Такий рівень гарантій доставки досить різко використовується через ризики втрати повідомлення. Його можна використовувати у ситуаціях, коли втрата повідомлення припустима — наприклад система збору метрик, коли втрата одного зразка не несе суттєвих ризиків та значення може бути інтерпольовано з використанням відомих значень.

Не менше одного разу (at least once). Такий рівень гарантій означає, що повідомлення може бути надіслано отримувачу декілька раз, до поки відправник не отримає підтвердження що повідомлення отримане.

Цей механізм є найбільш поширеним, адже за допомогою нього є гарантія того, що повідомлення буде опрацьовано, адже якщо отримувач не підтвердив обробку повідомлення через власну помилку або через помилку мережі при

доставці повідомлення його буде доставлено повторно.

Зворотній бік це ризик того, що повідомлення буде надіслано більш ніж один раз — якщо під час обробки повідомлення сталась помилка в додатку, яка унеможливлює подальшу роботу та процес завершується, тоді відправник може надіслати це ж саме повідомлення ще раз. Цей ризик вимагає, щоб система, яка отримує повідомлення була здатна опрацювати дублікати повідомлень.

Це може бути досягнуто, наприклад, шляхом визначення унікального ідентифікатора для кожного повідомлення. У такому випадку компонент, який отримує повідомлення, має зберігати ідентифікатори всіх отриманих та опрацьованих раніше повідомлень та для кожного нового повідомлення перевіряти, чи не було раніше опрацьовано повідомлення з таким самим ідентифікатором. Альтернативою штучному ідентифікатору може бути, наприклад хеш-сума від вмісту повідомлення.

Іншим способом може бути проектування системи таким чином, коли повторна обробка повідомлення не змінить стану системи — у такому випадку множинна обробка одного повідомлення не несе ризиків приведення системи та її даних неконсистентний стан.

Точно один раз (exactly once). Коли систем гарантує доставку повідомлення тільки один раз, це означає що повідомлення гарантовано буде доставлено тільки один раз. Цей рівень гарантій є найскладнішим, для реалізації та вимагає додаткових блокувань та розподілених транзакцій, що по-перше не дає стовідсоткових гарантій, а по-друге вимагає координації транзакції між різними компонентами що, в свою чергу, призводить до зменшення пропускної здатності [18].

Ще однією характеристикою способу асинхронної комунікації є порядок доставки повідомлень. При використанні способів доставки повідомлень розрізняють такі, що зберігають порядок повідомлень та можуть доставляти повідомлення у довільному порядку.

Якщо спосіб комунікації діє гарантує черговість доставки, то такий спосіб

діє як класична FIFO (first in, first out) черга де повідомлення доставляються споживачу у порядку їх публікації. У іншому випадку послідовність доставки повідомлень може бути довільною з метою забезпечення вищої пропускної здатності.

1.5 Формулювання проблемних питань та завдань дослідження

Впровадження мікросервісної архітектури в розподілених системах суттєво змінює підходи до розробки та управління програмним забезпеченням. Основним викликом є забезпечення ефективної асинхронної взаємодії між мікросервісами, яка відіграє ключову роль у забезпеченні високої доступності та масштабованості системи.

На сьогодні існує широкий спектр технологічних рішень для організації асинхронної комунікації в розподілених системах. Від вибору конкретних інструментів та платформ значною мірою залежить загальна архітектура системи, її продуктивність, масштабованість та надійність. Необхідно ретельно вивчити та порівняти різні підходи, враховуючи їх переваги та обмеження.

Кожне інфраструктурне рішення має свої унікальні властивості, що впливають на загальну роботу системи. Важливо визначити, як ці властивості впливають на ефективність взаємодії між мікросервісами, зокрема, на швидкість обробки запитів, витривалість системи до помилок та її масштабованість. У підрозділі 1.3 було розглянуто різні способи організації асинхронної взаємодії та переконались що кожен з перелічених способів має власні переваги та недоліки та може бути використаний у певних ситуаціях. Також було розглянуто варіанти комбінування різних способів з метою максимізації переваг та нівелювання недоліків.

У підрозділі 1.4 наведено перелік варіантів гарантій доставки повідомлень. Вибір конкретного механізму залежить від специфіки застосування та вимог до надійності системи.

При виборі конкретного рішення для асинхронної комунікації треба також

вважати на ризики так званого «вендор-локу» - це залежність від конкретного постачальника технологічних рішень, яка може обмежити гнучкість та розвиток системи. Важливо розробити підходи, які дозволять мінімізувати ризики вендор-локу, забезпечуючи системі можливість легкої адаптації до змін у технологічному ландшафті.

У промислових системах критично важливо мінімізувати ризики, пов'язані з надійністю, продуктивністю та безпекою. Вивчення та застосування найкращих практик та стандартів є ключовим для створення стійких та ефективних рішень у сфері асинхронної взаємодії.

В рамках даної роботи буде проведено дослідження, спрямоване на розробку способу асинхронної взаємодії, який дозволяє абстрагувати бізнес-логіку додатку від конкретних інфраструктурних рішень. Це дасть можливість розробляти програмне забезпечення без прив'язки до певних технологій та забезпечить гнучкість у виборі та комбінуванні різних інфраструктурних рішень для досягнення найкращих характеристик системи.

2 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ІНФОРМАЦІЙНОЇ СИСТЕМИ ДЛЯ АСИНХРОННОЇ ВЗАЄМОДІЇ У РОЗПОДІЛЕНІЙ КОМП'ЮТЕРНІЙ СИСТЕМІ

2.1 Виклики асинхронних методів взаємодії між компонентами розподіленої комп'ютерної системи

Асинхронні методи комунікації відіграють вирішальну роль у розвитку та оптимізації розподілених систем, що базуються на мікросервісній архітектурі. Основна перевага цих методів полягає в можливості масштабування систем, підвищення їхньої відмовостійкості та ефективного розподілу навантаження. Втім, вони також вносять певні виклики та складнощі в процес розробки та експлуатації системи.

Серед основних недоліків асинхронної комунікації є збільшення складності архітектури системи. Розробники змушені адаптувати свої підходи до розробки, приймаючи подійно-орієнтовану програмну модель. Це вимагає від них глибокого розуміння асинхронних процесів та здатності ефективно управляти потоками даних. Складність полягає в непередбачуваності часу та порядку виконання різних частин коду, особливо коли ці частини взаємодіють між собою. Для моделювання синхронної поведінки потрібно створювати окремі комунікаційні канали для передачі запитів та отримання відповідей.

Асинхронність ускладнює також процес налагодження та діагностики системи. Відсутність фіксованої послідовності обробки повідомлень може призвести до труднощів у визначенні точних причин помилок та збоїв. Повідомлення, які обробляються не у послідовності відправлення чи з затримками, ускладнюють відстеження потоків даних та виявлення місць збоїв в системі.

Управління помилками у асинхронному коді вимагає особливої уваги. Традиційні блоки обробки помилок часто виявляються неефективними, а винятки, що виникають під час асинхронних операцій, можуть не бути вчасно помічені або оброблені. Це може призвести до накопичення невиявлених

помилок, які ускладнюють подальшу роботу системи.

Тестування асинхронного коду також вимагає додаткових зусиль. Необхідно точно відтворювати асинхронні поведінки та умови гонки, що збільшує складність та час, необхідний для проведення тестувань. Симуляція асинхронних сценаріїв вимагає розробки спеціалізованих інструментів та методик.

На закінчення, необхідно зазначити, що асинхронна комунікація, хоча й відкриває широкі можливості для масштабування та оптимізації розподілених систем, вимагає ретельного проектування, продуманого підходу до розробки, налагодження та тестування. Розробники повинні бути готові до викликів, які ставить перед ними асинхронна взаємодія, і знаходити ефективні шляхи їх подолання.

2.2 Вимоги до компоненту асинхронної комунікації у розподілених комп’ютерних системах

При розробці компоненту для асинхронної комунікації у розподіленій комп’ютерній системі важливо визначити вимоги, які така система має задовільнити.

Компонент має реалізовувати єдиний інтерфейс взаємодії, який дозволить програмному коду використовувати цей компонент незважаючи на інфраструктурний компонент, який використовується для комунікації.

Ключовою характеристикою компонента, що буде змодельовано та розроблено в рамках цього дослідження, є можливість використання різних інфраструктурних рішень у якості транспортного шару для передачі повідомлень. При чому різні інфраструктури компоненти можуть використовуватись одночасно в рамках системи для реалізації різних каналів комунікації. Це дозволить компоненту використовувати переваги різних систем обміну повідомленнями та застосовувати систему, що найбільш підходить у кожному конкретному випадку. Крім того це дозволить змінювати транспортний

шар без втручання у код додатку, що використовує компонент. На рисунку 2.1 схематично зображено розділення інформаційної системи на різні шари, де кожен шар має визначену зону відповідальності та чітко окреслену задачу.

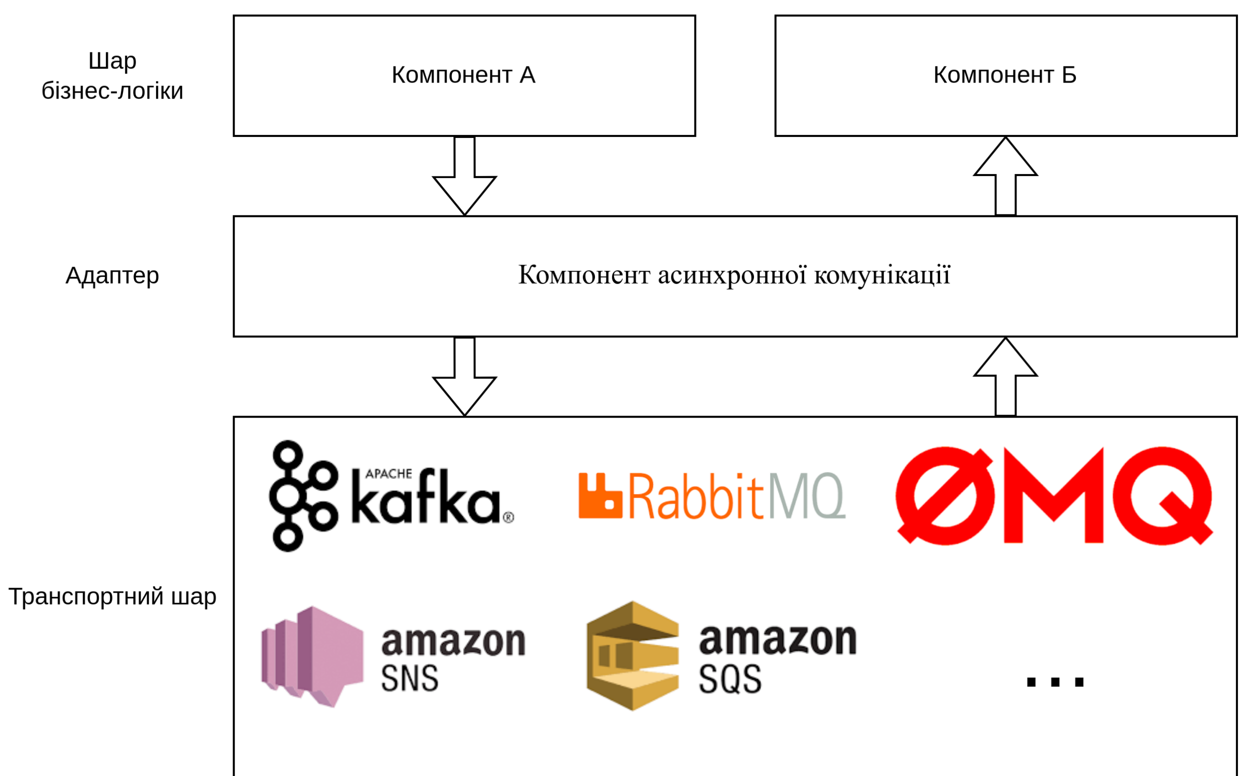

Рисунок 2.1 — Схематичне зображення інформаційної системи та місце компоненту асинхронної комунікації

Оскільки компонент має надавати можливість використовувати різні інфраструктурні рішення у якості транспортного шару, тоді він має надавати можливість легкої модифікації через конфігурацію, що дозволить динамічно змінювати налаштування. Крім того, налаштування компоненту мають надавати можливість легко визначати для яких каналів обміну даними які інфраструктурні компоненти мають використовуватись.

Забезпечення різних форматів серіалізації та десеріалізації даних є важливим, оскільки компонент має буди розроблено з врахуванням різних можливих сценаріїв використання ми не можемо обмежуватись якимось одним способом серіалізації повідомлення. Крім того, різні інфраструктурні

компоненти можуть накладати обмеження на те, яким чином повідомлення може бути представлено на транспортному рівні. Це вимагає від компоненту надавати можливість користувачу через конфігурацію визначити який метод серіалізації та десеріалізації застосовувати у кожному з конкретних випадків.

Важливим елементом будь-якої системи є можливість спостереження (моніторинг) за системою щоб у оператора була можливість відреагувати на нештатну поведінку системи та вжити заходів для усунення можливих недоліків. Ведення журналу (логування) також є важливим елементом системи що дозволяє проаналізувати як система поводилась при певних обставинах, виявити можливі недоліки та усунути їх. Тому надання можливості моніторингу стану системи та логування є однією з ключових вимог до компоненту асинхронної взаємодії.

Важливою складовою є швидкодія системи. В системах асинхронної комунікації, як і у будь-якій іншій, суттєвим аспектом є загальна швидкодія. У сучасному світі швидкість реакції на якусь подію є ключовим фактором та не може ігноруватись. Проте, в рамках дослідження, ми будемо розробляти компонент, який буде використовувати вже існуючи інфраструктурні рішення у якості транспортного шару, тому швидкодія буде прямо залежати саме від цієї частини, яка знаходиться поза межами контролю нашої системи за на неї будуть впливати як сам інфраструктурний компонент, що використовується, так і його налаштування.

Для компонента асинхронної комунікації важливою вимогою є гарантія доставки повідомлення. У розділі **Error! Reference source not found.** було р озглянуто різні рівні гарантії доставки повідомлення. При виборі конкретної реалізації важливо зважати на вимоги та обрати рішення, яке забезпечить необхідний рівень якості сервісу.

Важливою вимогою до компонента асинхронної комунікації може бути також можливість маршрутизації повідомлень, яка надає користувачу системи змогу гнучко налаштовувати систему та дозволяє надсилати повідомлення до різних отримувачів що важливо для розподілення навантаження системи, ізоляції

між різними кластерами даних, та A/B тестування. Існують різні варіанти маршрутизації повідомлень:

Найпростішим способом маршрутизації повідомлень є пряма доставка повідомлення одному споживачу. Схематично зображено на рисунку 2.2.

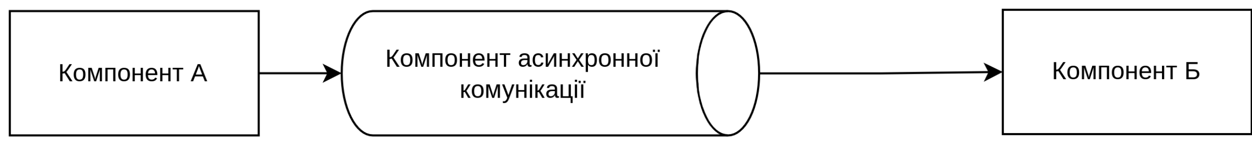

Рисунок 2.2 — Пряма доставка повідомлення

Іншим способом маршрутизації повідомлень є розповсюдження повідомлення всім отримувачам, які були підписані на нього. Такий спосіб маршрутизації може бути корисний коли кожен з отримувачів виконує різні задачі на підставі одного і того ж самого повідомлення. Схематично такий спосіб маршрутизації зображено на рисунку 2.3.

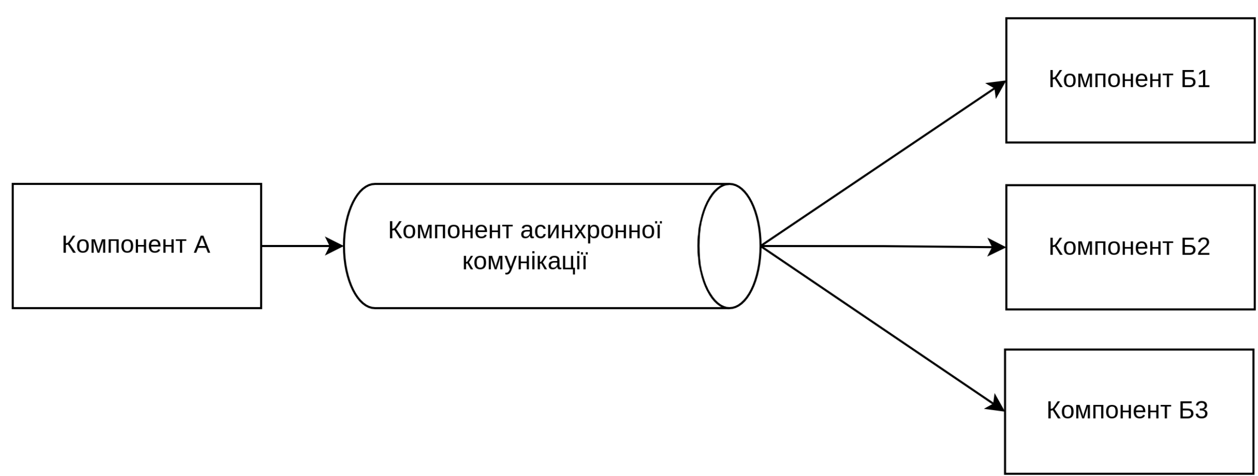

Рисунок 2.3 — Доставка повідомлення всім підписникам

Більш складним прикладом маршрутизації повідомлення є розподілення повідомлень відповідно до певних вагових коефіцієнтів. Такі коефіцієнти можуть залежати, наприклад, від потужності серверів які використовуються для запуску компонент, що обробляють повідомлення. Іншим прикладом застосування вагових коефіцієнтів є динамічне керування в залежності від поточної завантаженості серверних потужностей. Схематично це відображено на

рисунку 2.4.

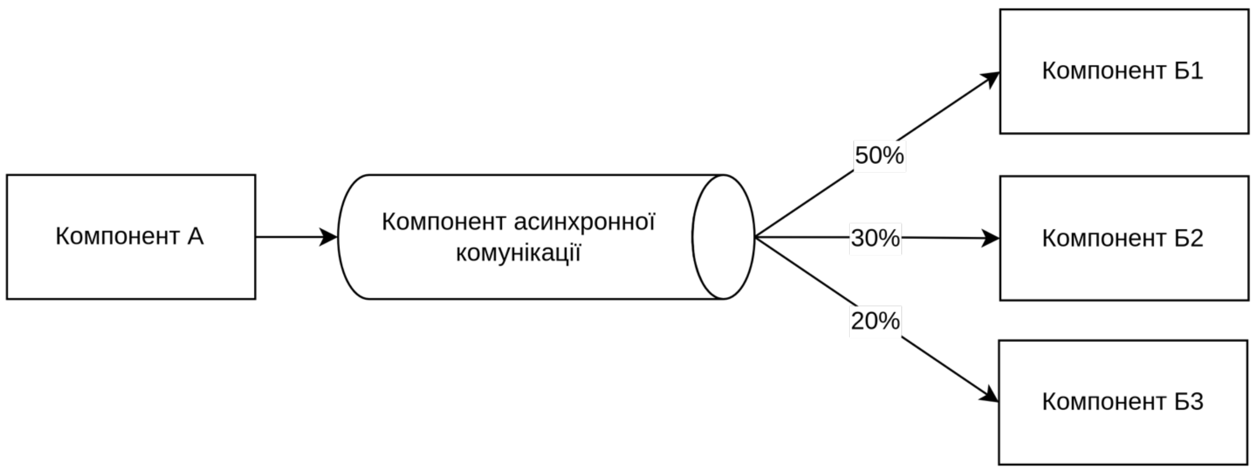

Рисунок 2.4 — Маршрутизація повідомлень зі застосуванням вагових коефіцієнтів

Іншим прикладом вибіркової маршрутизації може слугувати доставка повідомлень на підставі параметрів підписки та певних атрибутів повідомлення. Наприклад, з метою ізоляції обробки повідомлень від різних клієнтів, між різними обчислювальними кластерами може застосовуватись маршрутизація на основі назви клієнта. Така ізоляція надає можливість обробляти повідомлення кожного з клієнтів окремо один від одного, обраховувати вартість обчислювальних ресурсів для кожного клієнта окремо. Приклад застосування наведено на рисунку 2.5

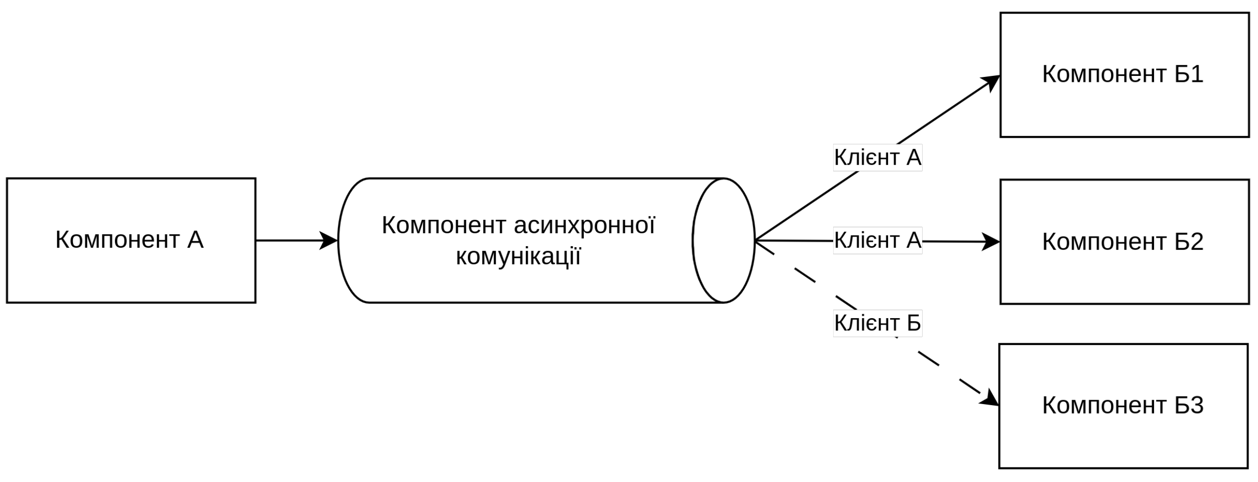

Рисунок 2.5 — Маршрутизація повідомлень на атрибутів повідомлення

2.3 Обрані критерії ефективності

В рамках цього дослідження буде розроблено передовий прототип інформаційної технології, який має на меті оптимізувати та спростити процес асинхронної комунікації в складних розподілених системах, які базуються на мікросервісній архітектурі. Цей прототип буде служити як новаторське рішення для управління асинхронними взаємодіями між різними компонентами системи, використовуючи різноманітні наявні інфраструктурні компоненти для асинхронної комунікації.

Прототип компоненту, який ми розробимо, буде інтегрувати такі потужні та відомі інструменти, як Kafka, RabbitMQ, AWS SNS/SQS та інші, використовуючи їх в якості транспортного шару. Цей підхід дозволяє не тільки використовувати переваги кожної з цих систем, але й забезпечує гнучкість у комбінуванні різних типів систем обміну повідомленнями для створення надійної та ефективної розподіленої системи.

У цій роботі ми також зосередимося на визначенні та аналізі ключових критеріїв ефективності запропонованої інформаційної технології. Основними критеріями універсальність використання різних наявних систем обміну повідомленнями та впровадження можливостей маршрутизації повідомлень.

Універсальність використання різних наявних систем обміну повідомленнями як транспортного шару. Це не тільки підвищує гнучкість системи, але й дозволяє максимально використовувати специфічні переваги кожної системи. Такий підхід дозволяє адаптувати систему до різних оперативних потреб та сценаріїв використання;

Впровадження можливостей маршрутизації повідомлень для ефективного розподілу навантаження, ізоляції кластерів даних та оптимізації процесів тестування. Це забезпечить більш стабільну роботу системи і підвищить її загальну продуктивність.

Хоча питання швидкодії та гарантій доставки повідомлень є важливими аспектами будь-якої системи обміну повідомленнями, в контексті цієї роботи ці параметри не розглядатимуться як основні критерії ефективності. Основна увага буде приділена інтеграції та взаємодії з існуючими системами. Однак, варто відзначити, що однією з ключових переваг розроблюваного компоненту буде його здатність до легкої заміни однієї системи обміну повідомленнями на іншу без значного перепроектування всієї системи. Це забезпечить значну гнучкість та адаптивність системи до змінних вимог та умов експлуатації.

3 ПРОГРАМНІ ЗАСОБИ ДЛЯ РЕАЛІЗАЦІЇ КОМПОНЕНТУ АСИНХРОННОЇ КОМУНІКАЦІЇ У РОЗПОДІЛЕНИХ КОМП’ЮТЕРНИХ СИСТЕМАХ

3.1 Вибір технологічного стеку

З метою верифікації ідеї, запропонованої в рамках даної роботи, буде розроблено прототип компоненту. В першу чергу потрібно обрати мову програмування, тому наведемо опис різних мов програмування, що можуть бути застосовані для розробки прототипу.

Для того, щоб зробити зважений вибір мови розробки спочатку визначимо характеристики мов програмування що важливі для розробки прототипу компоненту асинхронної комунікації.

Рівень абстракції у мові програмування визначає ступінь, до якого мова віддаляється від деталей машинного коду та операційної системи, наближаючись до більш високорівневих концепцій та мовних конструкцій. Вищий рівень абстракції означає, що мова приховує більше низькорівневих деталей від програміста, дозволяючи зосередитися на логіці та функціональності програми, а не на специфікації апаратного забезпечення чи управлінні ресурсами. Наприклад, мови високого рівня, такі як Python або Java, надають більш абстрактні засоби для роботи з даними та виконанням програми, у той час як мови низького рівня, такі як C++, C чи асемблер, вимагають більш детального управління ресурсами та пам'яттю.

Модель керування пам'яттю у програмуванні - це механізм, який визначає, як мова програмування або середовище виконання взаємодіє з пам'яттю комп'ютера. Це включає в себе процеси виділення, використання та звільнення пам'яті для зберігання даних та програмних структур. Модель керування пам'яттю може бути автоматизованою, як у мовах зі збирачем сміття (наприклад, Java чи Python), де система сама відслідковує та звільняє не використовувану пам'ять, або ручною, як у C чи C++, де програмісти самі відповідають за виділення та звільнення пам'яті. Керування пам'яттю є ключовим аспектом у

проектуванні та оптимізації програмного забезпечення, адже неефективне керування пам'яттю може призвести до помилок, таких як витоки пам'яті або порушення доступу до пам'яті.

Підтримка багатопоточності в програмуванні означає наявність в мові програмування механізмів, які дозволяють керувати декількома потоками виконання одночасно. У такому випадку програма може виконувати кілька завдань паралельно, підвищуючи ефективність та продуктивність. Багатопоточність є ключовою властивістю для створення високопродуктивних та ефективних додатків, особливо в мережевому програмуванні, великих обчислювальних завданнях та інтерактивних програмах.

Переносимість відноситься до здатності програмного забезпечення або коду працювати на різних обчислювальних платформах без змін або з мінімальними модифікаціями. Це означає, що програма, написана для однієї системи, наприклад, Windows, може легко працювати на іншій, такій як macOS або Linux. Переносимість є важливою характеристикою мов програмування та програмних рішень, оскільки це знижує витрати на розробку та підтримку, дозволяючи розробникам створювати більш універсальне програмне забезпечення, яке може задовольнити більш широку аудиторію користувачів.

Важливою характеристикою успішності мови програмування є наявність відомих продукти, що використовують мову. Це дає розробникам впевненість у тому, що мова має комерційний успіх, має усталену спільноту, яка підтримує мову та розвиває екосистему розробляючи та публікуючи бібліотеки для вирішення типових задач. Крім того це означає, що на ринку присутні інженери розробники з відповідними навичками використання мови програмування, яких можна найняти для розробки та впровадження продукту.

3.1.1 Java

Мова програмування Java, розроблена компанією Sun Microsystems (тепер частина Oracle), є однією з найбільш широко використовуваних мов

програмування у світі. Її ключова особливість — це можливість написання коду, який може виконуватися на будь-якому пристрої з Java Virtual Machine (JVM), що забезпечує високу переносимість. Java є мовою зі строгою статичною типізацією, що забезпечує додаткову безпеку та надійність коду, зменшуючи ймовірність помилок, пов'язаних з типами даних.

Java також відома своїм багатим набором бібліотек та фреймворків, що робить її популярною в розробці корпоративних застосунків, веб-сервісів та мобільних додатків. Її об'єктно-орієнтований підхід сприяє створенню модульного та масштабованого коду. Крім того, Java підтримує багатопоточність, що є важливим для розробки застосунків з високою продуктивністю.

Однак Java має і свої недоліки. Одним з основних є її продуктивність, оскільки виконання коду в JVM може бути повільнішим у порівнянні з мовами, що компілюються безпосередньо в машинний код. Також сучасні розробники можуть вважати Java важкою для швидкої розробки через її складність та вимогливість до деталей у коді. Велика кількість шаблонного коду та вимога до об'явлення змінних може ускладнювати роботу з Java, особливо порівняно з більш гнучкими мовами, такими як Python чи JavaScript.

У таблиці 3.1 зазначимо ключові характеристики мови програмування Java що були визначені як важливі в контексті даного дослідження.

Таблиця 3.1 — Ключові характеристики мови програмування Java

Рівень абстракції	Високий рівень абстракції
Модель керування пам’яттю	Автоматичне керування пам’яттю
Підтримка багатопоточності	Так, на рівні потоків операційної системи
Переносимість	Підтримується велика кількість платформ на рівні середовища виконання.
Відомі продукти	Apache Kafka, Apache Cassandra

3.1.2 C#

Мова програмування C#, розроблена компанією Microsoft, є важливою частиною екосистеми .NET. Вона була створена як відповідь на Java, і, подібно

до Java, C# є об'єктно-орієнтованою, статично типізованою мовою. Однак, вона включає кілька важливих покращень і вдосконалень у порівнянні з Java, зокрема, більш гнучку систему типів та підтримку функціонального програмування.

Основна сила C# полягає в її тісній інтеграції з Windows та .NET Framework. Це робить C# ідеальною для розробки Windows-орієнтованих застосунків, включаючи настільні програми, веб-додатки та ігри (особливо з використанням Unity). C# також відома своєю чистотою синтаксису, що робить код зрозумілим та легким для читання.

C# включає в себе багато сучасних можливостей програмування, таких як LINQ (Language Integrated Query), асинхронне програмування та підтримка лямбда-виразів. Ці функції дозволяють писати високофункціональний та ефективний код. Іншою важливою особливістю є її підтримка багатопоточності та асинхронних операцій, що є критично важливим для сучасних застосунків.

Однак, незважаючи на ці переваги, C# має кілька недоліків. Оскільки вона є ключовою частиною екосистеми .NET, її використання найкраще підходить для платформ, що підтримуються Microsoft, хоча .NET Core розширює цю можливість. Це може обмежити її використання в деяких середовищах, особливо на серверах, що не використовують Windows. Також, подібно до Java, C# може виявитися менш продуктивною в порівнянні з мовами, що компілюються безпосередньо в машинний код.

У таблиці 3.2 зазначимо ключові характеристики мови програмування C# що були визначені як важливі в контексті даного дослідження.

Таблиця 3.2 — Ключові характеристики мови програмування C#

Рівень абстракції	Високий рівень абстракції
Модель керування пам’яттю	Автоматичне керування пам’яттю
Підтримка багатопоточності	Так, на рівні потоків операційної системи
Переносимість	Підтримується велика кількість платформ на рівні середовища виконання.
Відомі продукти	Unity Engine, Visual Studio, Orleans

3.1.3 Node.js (JavaScript)

Платформа Node.js представляє унікальний підхід у світі програмування, переносячи мову JavaScript, яка традиційно асоціювалася з веб-браузерами, у середовище серверного програмування. Node.js - це не сама мова, а середовище виконання, яке дозволяє використовувати JavaScript для розробки серверних застосунків, інструментів командного рядка та навіть для розробки на стороні клієнта у вигляді універсальних JavaScript-додатків.

Однією з ключових особливостей Node.js є його неблокуюча модель вводу/виводу, яка оптимізована для асинхронних операцій та здатна обробляти велику кількість паралельних з'єднань. Це робить Node.js ідеальним для розробки легких, швидких мережевих застосунків, таких як API сервери, реальний обмін даними (наприклад, чати) та потокові застосунки.

JavaScript на Node.js відрізняється від інших серверних мов тим, що він має однопоточну модель з подієво-орієнтованим циклом, яка дозволяє ефективно обробляти асинхронні запити. Це різко контрастує з традиційними багатопоточними серверними мовами, що може бути як перевагою, так і недоліком залежно від конкретного випадку використання.

Перевагами використання JavaScript на Node.js є його швидкість та ефективність завдяки асинхронному неблокуючому вводу/виводу. Node.js має велику спільноту та багатий набір модулів, доступних через менеджер пакетів npm, що робить його дуже гнучким для розробників. Також, єдиність мови на клієнті та сервері спрощує розробку та підтримку веб-додатків.

Проте, Node.js може стикатися з проблемами при обробці важких обчислювальних задач, які можуть блокувати його однопотоковий цикл подій. Також, управління пам'яттю та відлагодження асинхронного коду можуть бути складнішими, ніж у традиційних багатопотокових середовищах.

Загалом, JavaScript на платформі Node.js є потужним інструментом для створення швидких, ефективних мережевих застосунків, зокрема у випадках, коли потрібно обробляти велику кількість асинхронних запитів. Однак, для

вирішення завдань, які потребують великих обчислювальних ресурсів, можуть бути більш підходящі інші технології.

У таблиці 3.3 зазначимо ключові характеристики мови програмування JavaScript та платформи Node.js що були визначені як важливі в контексті даного дослідження.

Таблиця 3.3 — Ключові характеристики мови програмування JavaScript (Node.js)

Рівень абстракції	Високий рівень абстракції
Модель керування пам'яттю	Автоматичне керування пам'яттю
Підтримка багатопоточності	Обмежена підтримка
Переносимість	Підтримується велика кількість платформ на рівні середовища виконання.
Відомі продукти	PayPal, LinkedIn Socket.IO

3.1.4 Golang

Мова програмування Golang, розроблена в Google, швидко завоювала популярність завдяки своїй простоті, ефективності та сучасному підходу до розробки програмного забезпечення. Golang була створена як мова, що усуває деякі складнощі, які виникають при роботі з іншими мовами, зокрема C++ і Java, пропонуючи при цьому високу продуктивність та ефективність.

Основною особливістю Golang є її підтримка конкурентності на мовному рівні. Golang використовує горутини (goroutines), які є легшими за традиційні потоки, дозволяючи легко створювати багатопоточні додатки. Горутини масштабуються більш ефективно, що робить Golang ідеальною для розробки високопродуктивних та асинхронних систем, особливо в контексті мікросервісів та розподілених систем.

Крім того, Golang пропонує простоту та чистоту синтаксису, що робить її легкою для вивчення та читання. Мова уникає надмірної складності, не включаючи такі речі, як класи та спадкування, які є характерними для багатьох об'єктно-орієнтованих мов. Це спрощує розробку та підтримку коду, роблячи

Golang відмінним вибором для команд розробників.

Golang також забезпечує високу продуктивність, завдяки компіляції в машинний код. Це означає, що програми, написані на Golang, виконуються швидко та ефективно, що є важливим для систем, що вимагають високої продуктивності та низької затримки, особливо у випадках асинхронної взаємодії.

Однак, Golang має деякі обмеження, такі як відсутність деяких особливостей, які можуть бути знайдені в інших мовах. Однак, ці недоліки часто переважуються перевагами, особливо коли йдеться про створення високопродуктивних, легко масштабованих та надійних розподілених систем.

Загалом, Golang відмінно підходить для розробки компонентів асинхронної комунікації у розподілених системах, завдяки її підтримці легких горутин, високій продуктивності, простоті та ефективності. Ці якості роблять її відмінним вибором для сучасних мікросервісних архітектур та додатків, які потребують швидкої та ефективної обробки паралельних запитів.

Також слід зазначити, що мова Golang активно використовується для розробки систем інфраструктурного рівня – прикладами таких рішень є Docker та Kubernetes.

У таблиці 3.4 зазначимо ключові характеристики мови програмування Golang що були визначені як важливі в контексті даного дослідження.

Таблиця 3.4 — Ключові характеристики мови програмування Golang

Рівень абстракції	Високий рівень абстракції
Модель керування пам’яттю	Автоматичне керування пам’яттю
Підтримка багатопоточності	Так, через механізм горутин.
Переносимість	Код компілюється під цільову платформу
Відомі продукти	Kubernetes, Docker

3.1.5 C++

Мова програмування C++, яка виникла як розширення мови C, відома своєю високою продуктивністю та гнучкістю. Вона широко використовується

для системного програмування, розробки відеоігор, додатків з високою продуктивністю та для ситуацій, де потрібен тісний контроль над апаратними ресурсами. Однією з ключових особливостей C++ є її здатність поєднувати низькорівневе програмування з високорівневими абстракціями, що робить її могутнім інструментом у руках досвідчених програмістів.

C++ є статично типізованою, компільованою мовою, що забезпечує високу продуктивність та оптимізацію використання пам'яті. Це важливо для додатків, які вимагають максимальної ефективності, таких як ігри, програми для обробки відео та аудіо, та системне програмування. C++ дозволяє безпосередньо взаємодіяти з апаратним забезпеченням та операційною системою, що надає розробникам повний контроль над ресурсами системи.

Мова також підтримує об'єктно-орієнтоване програмування, включаючи концепції, такі як класи, спадкування, поліморфізм та інкапсуляція. Ці особливості роблять C++ потужною у створенні складних програмних систем, які легко масштабуються та модифікуються. Крім того, C++ підтримує шаблони, що дозволяє програмістам писати узагальнений код, який може працювати з різними типами даних.

Проте, разом з цими перевагами йдуть і певні складнощі. C++ вимагає від розробників глибокого розуміння таких понять, як управління пам'яттю та вказівники, що може призводити до складностей, особливо для новачків. Невірне управління пам'яттю може призвести до серйозних помилок, таких як витоки пам'яті та порушення доступу до пам'яті.

Крім того, складність синтаксису C++ та його багатий набір можливостей можуть зробити розробку більш трудомісткою порівняно з більш високорівневими мовами, такими як Python або Java. Незважаючи на це, C++ залишається важливою мовою у світі програмування, особливо коли мова йде про розробку високопродуктивних додатків, де потрібен тісний контроль над ресурсами системи.

У таблиці 3.5 зазначимо ключові характеристики мови програмування C++

що були визначені як важливі в контексті даного дослідження.

Таблиця 3.5 — Ключові характеристики мови програмування C++

Рівень абстракції	Низький рівень абстракції
Модель керування пам’яттю	Ручне керування пам’яттю
Підтримка багатопоточності	Так, через потоки операційної системи
Переносимість	Код компілюється під цільову платформу
Відомі продукти	Adobe Photoshop, Microsoft Office

3.1.6 Rust

Rust, молода мова програмування, розроблена Graydon Hoare у Mozilla Research, швидко набула популярності завдяки своєму акценту на безпеці, продуктивності та паралельності. Основною особливістю Rust є її унікальна система управління пам'яттю, яка використовує правила володіння для запобігання помилкам, пов'язаним з пам'яттю, не потребуючи при цьому автоматизованого управління пам'яттю, як, наприклад, збирання сміття.

Ця мова поєднує елементи низькорівневого програмування з високорівневими абстракціями, роблячи її ідеальною для системного програмування та розробки високопродуктивних застосунків. Rust підтримує конкурентність на мовному рівні, що робить її зручною для створення багатопоточних додатків. Ця мова також акцентує на безпеці типів та підтримує відсутність гонок даних, що робить її стійкою до певних типів помилок, які можуть виникати в інших мовах.

Синтаксис Rust нагадує C та C++, але він також вплинутий мовами функціонального програмування, що вносить у нього елементи, такі як незмінність, функції вищого порядку та алгебраїчні типи даних. Це робить Rust привабливим для розробників, які шукають більш безпечну альтернативу C++.

Однак, навчання Rust може бути складнішим через його строгі правила володіння та позичання. Незважаючи на це, Rust стає все більш популярним серед розробників, які цінують його безпеку, продуктивність та сучасні

можливості. Rust використовується в широкому спектрі застосунків, від системного програмування до веб-розробки, завдяки своїй гнучкості та надійності.

У таблиці 3.6 зазначимо ключові характеристики мови програмування Rust що були визначені як важливі в контексті даного дослідження.

Таблиця 3.6 — Ключові характеристики мови програмування Rust

Рівень абстракції	Середній рівень абстракції
Модель керування пам’яттю	Ручне керування пам’яттю
Підтримка багатопоточності	Так, через потоки операційної системи
Переносимість	Код компілюється під цільову платформу
Відомі продукти	Firefox, Dropbox

3.1.7 Python

Python, широко відомий своїм читабельним та легким для розуміння синтаксисом, є однією з найбільш популярних мов програмування для швидкої розробки та прототипування. Його гнучкість та велика кількість бібліотек роблять його зручним вибором для широкого спектру проектів, включаючи веб-розробку, наукові дослідження та автоматизацію.

Однак, попри свої переваги, Python може не бути найкращим вибором для деяких сценаріїв, особливо коли йдеться про високопродуктивні або ресурсно-вимогливі застосунки. Його динамічна природа та інтерпретованість часто призводять до нижчої продуктивності порівняно з компільованими мовами, такими як C++ або Java. Це може бути критичним недоліком у випадках, де потрібна висока продуктивність, наприклад, в системах з великою кількістю асинхронних запитів або високим навантаженням на обробку даних.

Додатково, управління пам'яттю в Python виконується через збирач сміття, що може призводити до непередбачуваних затримок у виконанні програм. Це може бути проблематичним для додатків, що вимагають високої надійності та стабільності.

Крім того, хоча Python підтримує асинхронне програмування через такі бібліотеки, як asyncio, його однопоточний характер може ускладнювати ефективне використання асинхронності, особливо у порівнянні з мовами, які мають вбудовану підтримку багатопотоковості, такими як Java або C#.

Загалом, Python є відмінним інструментом для багатьох задач, але його обмеження з точки зору продуктивності, управління пам'яттю та підтримки асинхронності роблять його не найкращим вибором для вимогливих асинхронних комунікаційних систем або високопродуктивних додатків.

У таблиці 3.7 зазначимо ключові характеристики мови програмування Rust що були визначені як важливі в контексті даного дослідження.

Таблиця 3.7 — Ключові характеристики мови програмування Rust

Рівень абстракції	Високий рівень абстракції
Модель керування пам’яттю	Автоматичне керування пам’яттю
Підтримка багатопоточності	Обмежена підтримка
Переносимість	Підтримується велика кількість платформ на рівні середовища виконання.
Відомі продукти	Instagram, Spotify.

3.1.8 Узагальнення та вибір мови програмування

Узагальнимо характеристики згаданих мов програмування та оберемо найбільш оптимальну мову для реалізації прототипу інформаційної системи для асинхроного обміну повідомленнями. У додатку Б наведено зведену порівняльну таблицю мов програмування за згаданими ознаками.

З метою обґрунтованого вибору технології проаналізуємо означені характеристики та порівняємо обрані мови програмування. Для кожної з характеристик, окрім переліку відомих продуктів, визначимо кількість балів для можливих значень що ми будемо використовувати для порівняння мов програмування.

Бали та їх значення вказано у таблиці 3.

Таблиця 3 — Характеристики мови програмування та бали для можливих значень

Характеристика	Кількість балів	Значення
Рівень абстракції	1	Низький рівень абстракції
	2	Середній рівень абстракції
	3	Високий рівень абстракції
Модель керування пам'яттю	1	Ручне керування пам'яттю
	2	Автоматичне керування пам'яттю
Підтримка багатопоточності	1	Багатопоточність відсутня
	2	Обмежена підтримка
	3	Багатопоточність підтримується
Переносимість	1	Переносимість обмежена, відсутня підтримка хоча б однієї з основних платформ.
	2	Підтримується велика кількість платформ на рівні середовища виконання.
	3	Код компілюється під цільову платформу

Визначимо бали по кожній з характеристик для кожної з мов програмування у таблиці 4.

Таблиця 4 — Характеристики обраних мов програмування та їх бали

	Java	C#	Node.js	Golang	C++	Rust	Python
Рівень абстракції	3	3	3	3	1	2	3
Модель керування пам'яттю	2	2	2	2	1	1	2
Підтримка багатопоточності	3	3	2	3	3	3	2
Переносимість	2	2	2	3	3	3	2
Всього	**10**	**10**	**9**	**11**	**8**	**9**	**9**

На підставі підрахунку балів мова Golang набрала найбільше балів — 11.

Мова Golang є вдалим вибором для розробки компонентів асинхронної

комунікації через свою унікальну комбінацію простоти, продуктивності та вбудованої підтримки конкурентності. Його легковагові горутини дозволяють створювати асинхронні операції, водночас забезпечуючи високу продуктивність з мінімальним споживанням ресурсів. Мінімалістичний синтаксис Golang сприяє швидкому розвитку та легкості підтримки коду. Крім того, багатий набір стандартних мережевих бібліотек та фреймворків в Golang робить його зручним для створення мережевих застосунків, зокрема у сферах, де потрібна ефективна асинхронна комунікація.

Крім того, на користь мови Golang говорить той факт, що цю мову програмування обрали для розробки таких інфраструктурних компонент як Docker та Kubernetes.

3.2 Розробка архітектури програмного компонента

3.2.1 Проектування предметної області програмного компонента

При розробці програмного компонента буде використано ряд понять для опису окремих складових. Визначимо ці поняття.

Визначимо «Повідомлення» («Message») як основну одиницю даних, яка передається між компонентами системи. Повідомлення може містити дані, команди, запити або відповіді. Є ключовим елементом у комунікації та координації між різними мікросервісами. Повідомлення складається з метаданих та корисного навантаження. Схематично структуру повідомлення зображено на рисунку 3.1.

«Метадані» («Metadata») повідомлення це частина повідомлення яка є службовими даними, які додаються до повідомлення. Такими даними може бути ідентифікатор повідомлення, час надсилання, тип повідомлення і так далі. На підставі цих метаданих буде здійснюватись маршрутизація повідомлення. Метадані являють собою перелік ключів та відповідних значень. Для прототипа, що буде розроблятись в рамках цієї роботи, типом даних для ключа та значення буде використовуватись рядковий тип даних.

Рисунок 3.1 — Схематичне зображення структури повідомлення

«Корисне навантаження» («Payload») це власне дані, які передає повідомлення які різні компоненти розподіленої системи використовують для своєї роботи. У нашій системі ми будемо використовувати масив байт у якості корисного навантаження. Це найбільш універсальний формат даних, що може бути використаний для передачі будь-яких даних.

Складові частини компоненту асинхронної комунікації та його взаємодія з зовнішніми системами зображено на рисунку 3.2.

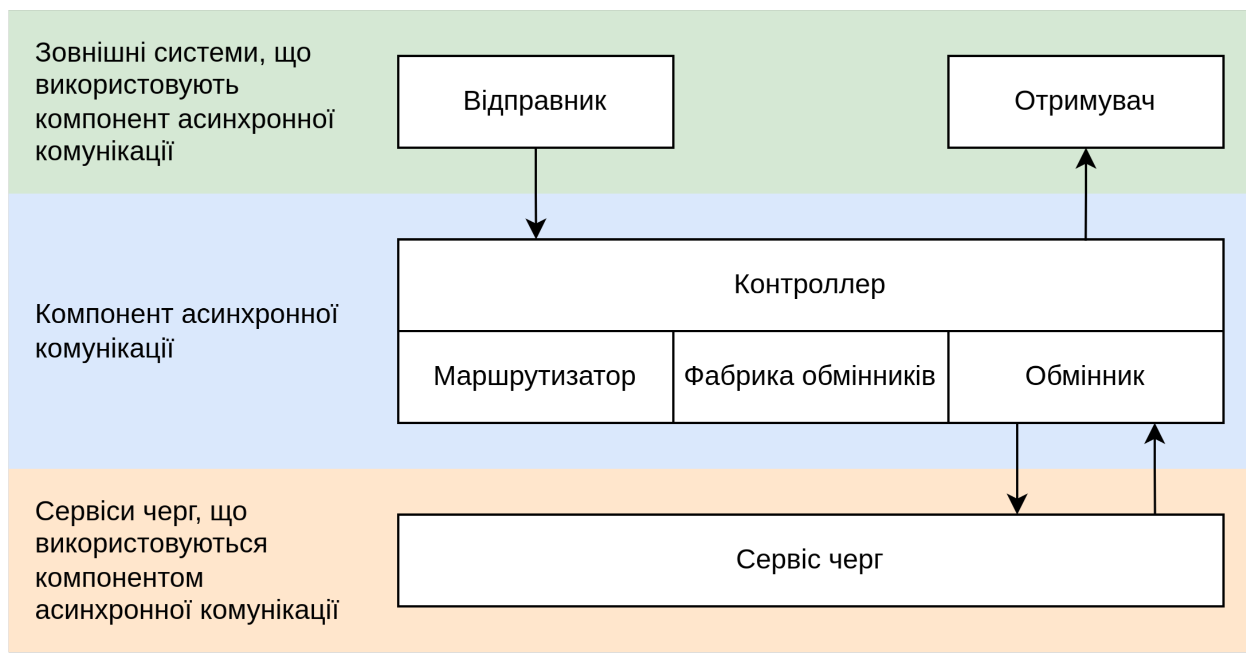

Рисунок 3.2 — Схема компоненту асинхронної комунікації та взаємодія з зовнішніми системами

«Відправником» вважатиметься компонент розподіленої системи що надсилає формує та повідомлення з використанням компоненту асинхронної комунікації.

«Отримувач», в свою чергу, це компонент розподіленої системи що отримує повідомлення, яке було надіслано відправником.

На рисунку 3.2 «контролер» — це центральний компонент системи, що відповідає за координацію роботи маршрутизатора, фабрики обмінників та обмінника. Крім того він надає публічний інтерфейс для використання компоненту асинхронної комунікації.

«Фабрика обмінників» («ExchangeFactory») це складова частина компонента асинхронної взаємодії що відповідає за ініціалізацію обмінників. На підставі назви обмінника та конфігурації компоненту асинхронної комунікації фабрика обмінників створює обмінник та ініціалізує з'єднання з сервісом черг.
«Обмінник» («Exchange») це частина компоненту асинхронної взаємодії що відповідає за надсилання повідомлення до сервісу черг. Кожне інфраструктурне рішення, що буде інтегроване до компоненту асинхронної взаємодії, буде мати окрему реалізацію обмінника.

«Маршрутизатор» («Router») це частина компоненту асинхронної взаємодії, яка відповідальна за спрямування повідомлення відповідно до налаштувань. На підставі правил маршрутизації у конфігурації, наданій при ініціалізації компоненту асинхронної комунікації, маршрутизатор здійснює вибір обмінника, куди потрібно надіслати відповідне повідомлення.

Сервіс черг це система (системи) що являють собою систему обміну повідомленнями яка використовується у якості транспортного шару.

3.2.2 Сценарії використання

Розглянемо основні сценарії використання компоненту асинхронної комунікації.

Узагальнено схеми використання компоненту асинхронної комунікації

наведено на рисунку 3.3.

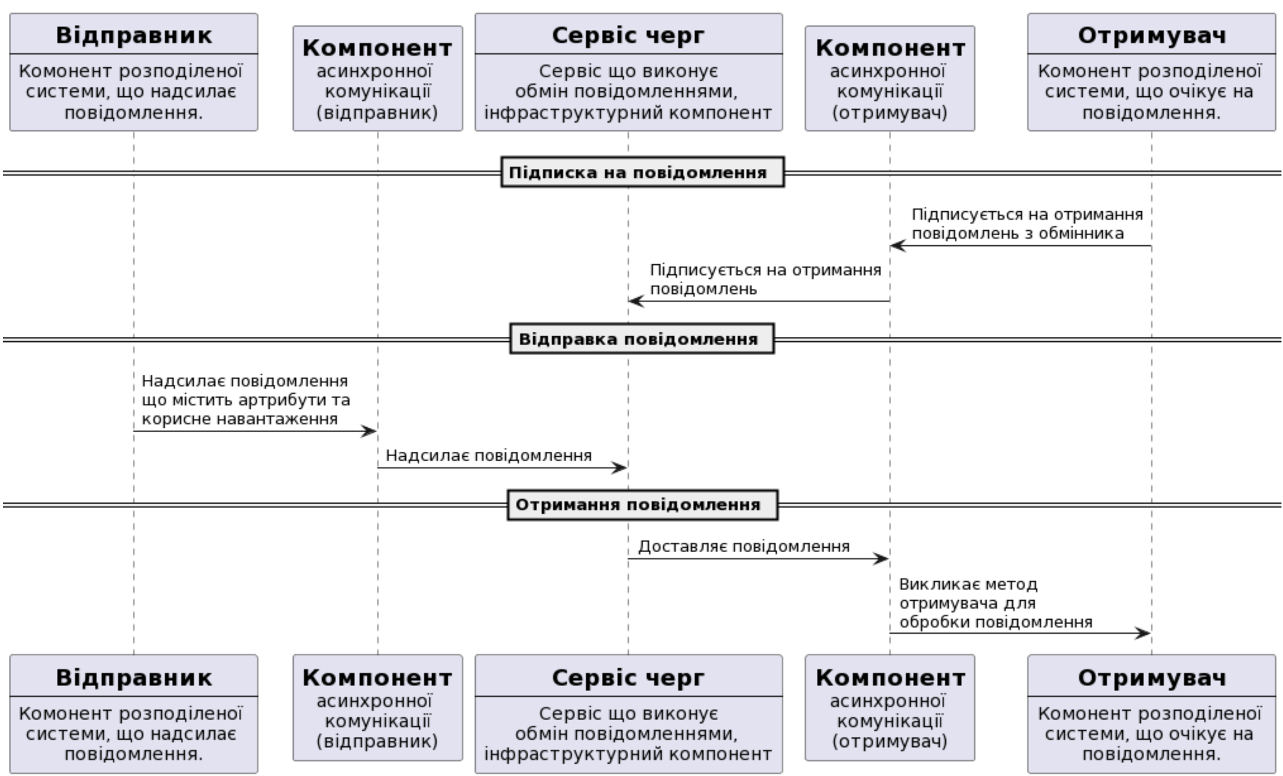

Рисунок 3.3 — Узагальнена схема використання компоненту асинхронної комунікації

На рисунку 3.3 відображена діаграма взаємодії компонентів в асинхронній комунікації розподіленої системи.

Перед початком процесу обміну повідомленнями отримувач має зареєструвати підписку - почати очікувати та читати повідомлення з черги.

Потім відправник генерує повідомлення, яке включає в себе важливі дані та метадані, необхідні для обробки і передачі далі. Далі повідомлення передається до компонента, що забезпечує асинхронну комунікацію, де воно може бути відформатоване або змінене за необхідності перед подальшою відправкою.

Далі, сервіс черг приймає повідомлення від цього компонента та займається його керуванням, забезпечуючи, що повідомлення буде доставлене до кінцевого отримувача навіть у випадку, коли система зайнята або отримувач не може відразу обробити повідомлення. Після того, як повідомлення надійде в чергу,

воно зберігається там до того моменту, коли отримувач буде готовий його обробити.

На завершальному етапі, компонент асинхронної комунікації, який виступає в ролі отримувача, витягує повідомлення з черги. Отримувач обробляє повідомлення, що може включати виклик специфічних методів для реакції на отримані дані.

Першим сценарієм, який ми розглянемо детально буде надсилання повідомлення до сервісу черг.

У додатку В наведено діаграму процесу взаємодії між компонентами розподіленої системи, яка займається асинхронним обміном повідомлень під час надсилання повідомлення. Діаграма деталізує процес від моменту створення повідомлення відправником до його обробки через ряд компонентів, які спільно забезпечують маршрутизацію, обробку та доставку повідомлення у сервіс черг.

Процес починається з відправника, який створює повідомлення із необхідними атрибутами та інформацією. Відправник служить початковою точкою для повідомлення, яке повинно бути оброблено та передано через систему.

Контролер виконує роль координатора процесу відправлення. Спочатку контролер звертається до маршрутизатора.

Маршрутизатор займається визначенням маршруту для повідомлення в системі. При виборі маршруту враховуються атрибути повідомлення на підставі яких і здійснюється вибір обмінника, куди повідомлення має бути надіслано.

Якщо маршрутизатор повернув назву обмінника, куди необхідно надіслати повідомлення, координатор звертається до фабрики обмінників.

Фабрика обмінників ініціює створенні обмінника, який буде обробляти повідомлення. Цей компонент динамічно генерує обмінники на основі потреб, що виникають під час процесу маршрутизації. Після створення екземпляру обмінника, фабрика зберігає його та при наступному звернені до того ж обмінника повертає вже створений раніше екземпляр.

Після отримання екземпляру обмінника координатор надсилає повідомлення з його використанням. Обмінник, у свою чергу, надсилає повідомлення до сервісу черг, який виконує роль інфраструктурного шару та забезпечує подальшу доставку повідомлення отримувачу.

У випадку, якщо на якомусь з етапів відбувається помилка, то координатор повертає помилку відправнику.

Другим сценарієм, що ми розглянемо буде отримання повідомлення.

На діаграмі послідовності що наведено у додатку Г описано детальний процес підписки на повідомлення та його отримання. Діаграма демонструє взаємодію між кількома компонентами системи, кожен з яких виконує певну роль у процесі підписки на повідомлення від початкового відправника до кінцевого отримувача.

Процес розпочинається з Отримувача, який активно очікує на повідомлення в системі. Отримувач не лише чекає на нові повідомлення, але й реєструє свою підписку на конкретний обмінник, щоб отримувати повідомлення, які стосуються його інтересів або обов'язків. Цей крок є фундаментальним для встановлення зв'язку між різними частинами системи та забезпеченням їх здатності спілкуватися між собою.

Контролер, наступний у ланцюжку, відіграє роль координатора для процесу відправлення повідомлень. Він відповідає за перевірку назви обмінника, що гарантує, що повідомлення буде направлено до правильного компонента для обробки. Після цього, контролер ініціює запуск обмінника, вказуючи на необхідність обробити нове повідомлення. У випадку виникнення помилок контролер повертає відповідну помилку, що дозволяє системі уникнути некоректної обробки даних або надсилання повідомлень не туди.

Фабрика здатна створювати нові обмінники або використовувати вже наявні екземпляри, залежно від поточної потреби системи. Це забезпечує ефективне використання ресурсів та знижує зайві навантаження на систему шляхом перевикористання компонентів, де це можливо.

Коли обмінник готовий до роботи, він бере на себе відповідальність за відправку повідомлення. Обмінник може виконувати цю дію безпосередньо, за допомогою виклику методу в бібліотеці, що відправляє повідомлення далі в систему, або може взаємодіяти з Сервісом черги, якщо це необхідно. Важливо зазначити, що обмінник також обладнаний механізмами для обробки помилок, що дозволяє системі гарантувати достовірність обробки повідомлень і уникати переривань у комунікації.

Сервіс черги відповідає за кінцеву обробку повідомлень, які проходять через систему. Він може приймати нові повідомлення, обробляти їх відповідно до визначених правил та передавати їх далі або виконувати потрібні дії. Цей компонент також може стикатися з ситуаціями, коли потрібно призупинити обробку повідомлень, наприклад, під час обслуговування системи, що також відображено на діаграмі як альтернативний шлях виконання.

3.3 Реалізація компонента асинхронної взаємодії з розподіленою системою

3.3.1 Конфігурація компонента асинхронної взаємодії.

Конфігурація є ключовим елементом компонента асинхронної комунікації. Приклад конфігурації у форматі JSON наведено у лістингу 3.1.

Лістинг 3.1 — Приклад конфігурації у форматі JSON

```
{
 "routes": [
   {
    "condition": {
     "@equals": {
      "key": "type",
      "value": "event"
     }
    },
   },
   "target": {
     "kind": "@direct",
     "config": {
      "exchange": "kafka_exchange"
```

```
        }
      }
    },
  ],
  "exchanges": {
    "kafka_exchange": {
      "kind": "kafka",
      "config": {
        "topic": "topic"
      }
    },
  }
}
```

Наведемо опис властивостей конфігурації:

— routes — властивість верхнього рівня, яка є масивом що визначає перелік маршрутів; очікується, що кожен з елементів масиву має 2 властивості - «condition» та «target»;

— routes[].condition — властивість маршруту яка є об’єктом та визначає умови, якім має відповідати повідомлення, щоб бути спрямованим по цьому маршруту;

— routes[].target — визначає куди потрібно спрямувати повідомлення, якщо воно відповідає умовам, що визначаються «condition»;

— exchanges — властивість верхнього рівня, що є об’єктом та визначає існуючи обмінники та параметри їх конфігурації; властивість об’єкту є назвою обмінника;

— exchanges{}.kind — визначає тип інфраструктурного компонента що використовується для обмінника;

— exchanges{}.config — об’єкт, що містить конфігурації для інфраструктурного компонента.

Під час розробки прототипу в рамках даного дослідження було розроблено впроваджено підтримку наступних умов:

— «@equals» — визначає, що атрибути повідомлення мають містити

певний ключ, що має певне значення, використовується для порівняння значення конкретного ключа в метаданих повідомлення з вказаним значенням, якщо вони співпадають, умова вважається виконаною;

— «@not» — дозволяє інвертувати результат іншої умови, це означає, що якщо вкладена умова повертає «true», @not поверне false, і навпаки;

— «@all» — дозволяє вказати список умов, всі з яких мають повернути true, щоб умова вважалася виконаною, це корисно, коли ви хочете надіслати повідомлення по маршруту, якщо вони відповідають кільком критеріям одночасно.

Розроблений прототип підтримує наступні типи цілей, куди потрібно спрямувати повідомлення що відповідає визначеній умові для маршруту:

— «@direct» — вказує один обмінник куди має бути відправлено повідомлення;

— «@weight» — вказує множину обмінників з їх вагами; якщо повідомлення відповідає умовам цього маршруту, тоді під час надсилання буде обрано один з відомих обмінників, при чому буде врахована вага, що вказана в конфігурації.

3.3.2 Ключові типи, що використовуються у компоненту асинхронної комунікації

Розглянемо ключові типи та інтерфейси що створено та використовуються у рамках компоненту асинхронної комунікації.

Повідомлення, що обробляються компонентом асинхронної комунікації мають реалізовувати інтерфейс Message, який наведено у лістингу 3.2.

Лістинг 3.2 — Інтерфейс Message та тип Metadata

```
type (
	Message interface {
		Metadata() Metadata
		Payload() []byte
	}
```

```
	Metadata map[string]string
)
```

Інтерфейс «Message» містить наступні методи:

- Metadata() — повертає метадані повідомлення які, у тому числі, використовуються для маршрутизації;
- Payload() — повертає динамічний масив байт який містить корисне навантаження повідомлення.

Тип «routeResolver» реалізує функціонал пошук маршруту, що відповідає повідомленню. У лістингу 3.3 наведено опис структури «routeResolver» та метод «FindMatching».

Лістинг 3.3 — Структура routeResolver

```
type routeResolver struct {
	routes []RouteConfig
}
```

Інтерфейс «Condition» описує метод, що має бути реалізовано різними умовами. Код інтерфейсу наведено у лістингу 3.4.

Лістинг 3.4 — Інтерфейс Condition

```
type Condition interface {
	Satisfy(msg Message) (bool, error)
}
```

У інтерфейсі «Condition» визначено тільки метод «Satisfy» що у якості аргументу приймає повідомлення та а у результаті повертає значення типу bool та помилку. Якщо повідомлення відповідає умові, тоді метод повертає true, інакше false. Якщо під час перевірки умови виникла помилка тоді метод має її повернути.

Приклад реалізації інтерфейсу «Condition» що реалізує оператор «@equal» наведено у лістингу 3.5.

Лістинг 3.5 — Реалізація інтерфейсу «Condition» для оператору $equal

```
type MetadataValueCondition struct {
```

```
	key   string
	value any
}

func (m *MetadataValueCondition) Satisfy(msg Message) (bool, error) {
	value, exists := msg.Metadata()[m.key]
	if !exists {
		return false, nil
	}

	return reflect.DeepEqual(value, m.value), nil
}
```

Тип «exchangeFactory» реалізує створення обмінників. Код декларації структури наведено у лістингу 3.6

Лістинг 3.6 — Структура exchangeFactory

```
type exchangeFactory struct {
	cache           exchangeStorage
	initiators      map[ExchangeKind]Connector
	exchangeConfigs map[ExchangeName]ExchangeDestination
}
```

Інтерфейс «Exchange» описує методи, що мають надавати інфраструктурно-залежні реалізації для відправки та отримання повідомлення. Інтерфейс містить методи «Publish» та «Subscribe». Код декларації інтерфейсу наведено у лістингу 3.7.

Лістинг 3.7 — Інтерфейс Exchange

```
type Exchange interface {
	Publish(ctx context.Context, message Message) error
	Subscribe(ctx context.Context, subscription Subscription) error
}
```

Метод «Publush» слугує для відправки повідомлень та приймає наступні аргументи:

- «ctx» типу «context.Context» — параметр, що дозволяє керувати життєвим циклом застосунку, це є типовим підходом для мови Go;
- «message» типу «Message» — повідомлення що потрібно надіслати.

Приклад реалізації методу «Publish» що використовує Apache Kafka у якості інфраструктурного компоненту, наведено у лістингу 3.8.

Лістинг 3.8 — Реалізація методу Publish з використанням Apache Kafka

```
func (k kafkaQueueHandler) Publish(ctx context.Context, message Message) error {
	pm := &sarama.ProducerMessage{
		Topic:   k.topic,
		Key:     nil,
		Value:   encode(message),
		Headers: k.buildHeaders(message),
	}

	_, _, err := k.producer.SendMessage(pm)
	return err
}
```

Метод «Subscribe» реалізує підписку на отримання нових повідомлень та приймає наступні методи:

— «ctx» типу «context.Context» — параметр, що дозволяє керувати життєвим циклом застосунку, це є типовим підходом для мови Go;

— «subscription» типу «Subscription» — обробник повідомлення що має бути надано зовнішнім кодом, що використовує компонент асинхронної комунікації.

Код декларації інтерфейсу Subscription наведено у лістингу 3.9.

Лістинг 3.9 — Інтерфейс Subscription

```
type Subscription interface {
	OnMessage(context.Context, Message) error
}
```

Приклад реалізації методу «Subscribe» що використовує Apache Kafka у якості інфраструктурного компоненту, наведено у лістингу 3.10.

Лістинг 3.10 — Реалізація методу Subscribe з використанням Apache Kafka

```
func (k kafkaQueueHandler) Subscribe(ctx context.Context, subscription Subscription) error {
	pc, err := k.consumer.ConsumePartition(k.topic, 0, sarama.OffsetNewest)
	if err != nil {
```

```
		return err
	}

	for {
		select {
		case <-ctx.Done():
			return ctx.Err()
		case msg := <-pc.Messages():
			md := k.getMetadata(msg.Headers)
			rm := NewReceivedMessage(md, msg.Value)

			err = subscription.OnMessage(ctx, rm)
			if err != nil {
				return err
			}
		}
	}
}
```

3.3.3 Ключові методи компоненту асинхронної комунікації

Розглянемо методи, що надає компонент асинхронної комунікації для відправки та отримання повідомлення.

Для відправки повідомлення використовується метод Publish. Код цього методу наведено у лістингу 3.11.

Лістинг 3.11 — Метод Publish

```
func (b *Bridge) Publish(ctx context.Context, message Message) (err error) {
	var dest Destination
	if dest, err = b.router.FindMatching(ctx, message); err != nil {
		return err
	}

	exchangeName := dest.Exchange()

	var e CreateExchange
	if e, err = b.factory.CreateExchange(ctx, exchangeName); err != nil {
		return err
	}
	return p.Publish(ctx, message)
}
```

Метод Publish приймає 2 параметри:

— «ctx» типу «context.Context» — параметр, що дозволяє керувати життєвим циклом застосунку, це є типовим підходом для мови Go;

— «message» типу «Message» — повідомлення що потрібно надіслати.

Використовуючи метод «FindMatching» структури «routeResolver» здійснюється пошук маршруту, що відповідає даному повідомленню. Код методу «FindMatching» наведено у лістингу 3.12.

Лістинг 3.12 — Метод routeResolver.FindMatching

```
func (r *routeResolver) FindMatching(ctx context.Context, message Message) (target Destination, err error) {
	var match bool

	for _, route := range r.routes {
		if match, err = route.Condition.Satisfy(message); err != nil {
			return target, err
		} else if !match {
			continue
		}
		target = route.Target
		return target, nil
	}

	return target, ErrNoMatchingTarget
}
```

Метод «FindMatching» приймає наступні аргументи:

— «ctx» типу «context.Context» — параметр що дозволяє достроково перервати виконання операції;

— «message» типу «Message» — повідомлення, для якого треба знайти маршрут.

Якщо такий маршрут було знайдено, тоді відбувається виклик методу «CreateExchange» що створює обмінник та повертає його. Код методу «CreateExchange» наведено у лістингу 3.13.

Лістинг 3.13 — Метод CreateExchange

```
func (q *exchangeFactory) CreateExchange(ctx context.Context, name
```

```
ExchangeName) (qh Exchange, err error) {
	dest, exists := q.exchangeConfigs[name]
	if !exists {
		return nil, fmt.Errorf("%w, exchange name %s",
ErrUnknownExchangeConfiguration, name)
	}

	conn, exists := q.initiators[dest.Kind]
	if !exists {
		return nil, fmt.Errorf("%w, exchange kind %s",
ErrUnknownExchangeKind, dest.Kind)
	}

	qh, err = conn.Connect(ctx, &dest)
	if err != nil {
		return nil, fmt.Errorf("%w: %s", ErrExchangeCreation, err.Error())
	}

	return qh, nil
}
```

Метод «CreateExchange» приймає наступні параметри:

— «ctx» типу «context.Context» — параметр що дозволяє достроково перервати виконання операції;

— «name» типу «ExchangeName» — назва обмінника що має бути використано для відправки повідомлення.

Метод здійснює ініціалізацію обмінника використовуючи раніше зазначену конфігурацію.

Після успішного створення обмінника метод «Publish» викликає метод обмінника «Publish» що здійснює відправку повідомлення з використанням інфраструктурного компонента.

У випадку, якщо на будь-якому етапі виникає помилка, така помилка буде повернена з методу «Publish» та може бути оброблена кодом, що викликав метод «Publish».

Для отримання повідомлення використовується метод «Subscribe», код методу наведено у лістингу 3.14.

Лістинг 3.14 — Метод Subscribe

```
func (b *Bridge) Subscribe(ctx context.Context, name ExchangeName, subscription Subscription) (err error) {
	var e Exchange
	e, err = b.factory.CreateExchange(ctx, name)
	if err != nil {
		return err
	}

	return e.Subscribe(ctx, consumer)
}
```

Метод приймає наступні параметри:

— «ctx» типу «context.Context» — параметр що дозволяє достроково перервати виконання операції;

— «name» типу «ExchangeName» — назва обмінника що має бути використано для отримання повідомлення;

— «subscription» типу «Subscription» — реалізація обробника повідомлення, код декларації інтерфейсу «Subscription» наведено у лістингу 3.9.

Метод «Subscribe» викликає метод «CreateExchange», код якого наведено у лістингу 3.13 для отримання реалізації обмінника, що відповідає конфігурації. Після отримання сконфігурованої реалізації обмінника, викликається метод «Subscribe» для ініціалізації підписки на повідомлення з обраного обмінника.

4 ВЕРИФІКАЦІЯ ТА ДОСЛІДЖЕННЯ КОМПОНЕНТУ АСИНХРОННОЇ КОМУНІКАЦІЇ

4.1 Методики тестування та верифікації асинхронної взаємодії

Важливість тестування в сучасній розробці програмного забезпечення є визначальною, оскільки воно відіграє ключову роль у забезпеченні якості, надійності та безпеки розроблюваних продуктів. В умовах невпинного технологічного розвитку та зростаючих вимог до програмного забезпечення, тестування стає не просто етапом в циклі розробки, але й стратегічно важливою частиною процесу розробки.

Перше і, мабуть, найважливіше значення тестування полягає у забезпеченні якості. Тестування дозволяє виявити та усунути помилки та баги до того, як продукт потрапить до кінцевого користувача, що знижує ризик проблем, пов'язаних з функціональністю, сумісністю та користувацьким досвідом. Якісне тестування гарантує, що програмний продукт відповідає всім вимогам та специфікаціям, що є критично важливим в умовах постійного зростання очікувань користувачів.

Крім того, тестування важливе для забезпечення безпеки програмного забезпечення. У сучасному світі, де кібербезпека є одним із головних пріоритетів, тестування допомагає ідентифікувати та виправити вразливості, які можуть бути використані зловмисниками. Це особливо актуально для розробки веб-додатків, мобільних додатків та інших продуктів, що вимагають забезпечення конфіденційності та цілісності даних користувачів.

Тестування також сприяє підвищенню ефективності та скороченню витрат у процесі розробки. Виявлення та усунення помилок на ранніх етапах розробки є значно дешевшим та менш трудомістким, ніж їх виправлення після випуску продукту. Автоматизоване тестування, що є частиною методологій неперервної інтеграції та неперервного розгортання (CI/CD), дозволяє швидко та ефективно перевіряти продукт після кожного оновлення, забезпечуючи стабільність та постійну готовність продукту до випуску.

З метою верифікації прототипу інформаційної системи, розробленого у рамках даного дослідження було реалізовано тестування системи з використанням вбудованих можливостей інструментарію мови розробки Golang.

Під час тестування, у випадку якщо система, що тестується, взаємодіє з зовнішніми залежностями, зазвичай такі залежності емулюються з використанням методів мокування та стабінгу. Мокування дозволяє створити легкі версії зовнішніх систем або сервісів, які імітують реальну взаємодію без необхідності підключення до реальних систем. Це допомагає тестувати поведінку системи в ізольованому середовищі, зосереджуючись лише на функціоналі, який безпосередньо стосується об'єкта тестування. Такий підхід сприяє ефективності та точності тестування, забезпечуючи контрольоване середовище, у якому можна точно відтворити та аналізувати конкретні сценарії та випадки використання.

Проте, у випадку тестування прототипу, що розроблено в рамках даного дослідження важливо використовувати реальні інфраструктурні компоненти, що дозволить гарантувати що система коректно працює з різними типами залежностей.

Саме тому, буде використано систему Docker для запуску інфраструктурних компонент. Це дозволить створити ізольоване та уніфіковане середовище для кожного компонента, забезпечуючи консистентність та відтворюваність умов розгортання незалежно від локального середовища розробника або сервера. Використання Docker також спрощує процес масштабування, оскільки дозволяє легко розгортати та управляти копіями контейнерів на різних середовищах. Крім того, це полегшує інтеграцію з системами безперервної інтеграції та безперервного розгортання (CI/CD), оскільки Docker-контейнери можуть бути швидко запущені та зупинені на різних етапах процесу розробки та розгортання.

Розглянемо код тестування прототипу асинхронної комунікації. Спочатку створюється структура для групування набору тестів та запускається тест. Код наведено у лістингу 4.1.

Лістинг 4.1 — Створення та запуск тесту

```
type testE2ESuite struct {
	suite.Suite
	pool      *dockertest.Pool
	ctx       context.Context
	resources []*dockertest.Resource
}

func TestE2E(t *testing.T) {
	suite.Run(t, new(testE2ESuite))
}
```

Передбачено, що є певні етапи роботи тесту, коли розробник може здійснити певні налаштування, створити необхідні ресурси або очистити використані ресурси. Існують такі методи:

— SetupSuite — налаштування та ресурси що створюються перед запуском набору тестів;

— SetupTest — виконується перед кожним тестом;

— TearDownTest — виконується після кожного тесту;

— TearDownSuite — виконується один раз, після завершення всіх тестів.

Буде використано метод SetupSuite для що створює з'єднання з сервісом Docker. Код наведено у лістингу 4.2.

Лістинг 4.2 — Підготовка для запису тестів

```
func (s *testE2ESuite) SetupSuite() {
	var err error
	s.pool, err = dockertest.NewPool("")
	s.Require().NoError(err)
	s.Require().NoError(s.pool.Client.Ping())

	s.ctx = context.Background()
}
```

Метод «SetupSuite» викликається один раз для групи тестів. Він створює з’єднання з Docker та перевіряє, що воно пройшло без помилок.

Далі запускається власне тест, що перевіряє що прототип працює. Код тесту наведено у лістингу 4.3.

Лістинг 4.3 — Тест TestKafka

```
func (s *testE2ESuite) TestKafka() {
	s.withKafka()

	b := bridge.New(bridge.Config{
		Routes: []bridge.RouteConfig{
			{
				Condition: &bridge.EverythingCondition{},
				Target:    bridge.DirectDestination("kafka_exchange"),
			},
		},
		ExchangeConfigs:
map[bridge.ExchangeName]bridge.ExchangeDestination{
			"kafka_exchange": {
				Kind:   "kafka",
				Config:                              toJsonRawMessage(s.T(),
bridge.KafkaConfig{Topic: "test_exchange"}),
			},
		},
	})

	kafka, err := bridge.NewKafka([]string{"localhost:9092"})
	s.Require().NoError(err)
	b.RegisterConnector("kafka", kafka)

	msgCh := s.expectMessageOnce(b, "kafka_exchange")
	msg,      err      :=      bridge.NewJSONMessage("Hello,      World!",
map[string]string{"foo": "bar"})
	s.Require().NoError(err)

	err = b.Publish(s.ctx, msg)
	s.Require().NoError(err)

	receivedMessage := <-msgCh
	s.Assert().Equal(msg.Metadata(), receivedMessage.Metadata())
	s.Assert().Equal(msg.Payload(), receivedMessage.Payload())
}
```

У тесті викликається метод приватний «withKafka» що створює Docker-контейнер з Apache Kafka.

Після цього створюється екземпляр прототипа компоненту та передається конфігурація. Конфігурація в тесті визначає один маршрут, якому відповідають

всі повідомлення та один обмінник, куди повідомлення мають потрапити. Така конфігурація дозволить переконатись що прототип працює та доставляє повідомлення.

Наступним кроком створюється та реєструється конектор до Apache Kafka. Після цього викликається метод «expectMessageOnce» що повертає канал (chan), що, в свою чергу, поверне повідомлення яке буде доставлено.

Далі створюється екземпляр структури повідомлення та відправляється за допомогою методу Publish. Після того, як повідомлення було надіслано, воно має потрапити до каналу, що було повернуто з функції «expectMessageOnce». Коли повідомлення надійшло, у тесті виконується перевірка що корисне навантаження відповідає тому, що було надіслано.

Код допоміжного методу «withKafka» наведено у лістингу 4.4.

Лістинг 4.4 — Допоміжний метод withKafka

```
func (s *testE2ESuite) withKafka() {
	resource, err := s.pool.RunWithOptions(
		&dockertest.RunOptions{
			Repository: "bitnami/kafka",
			Tag:        "3.5.1-debian-11-r7",
			Env: []string{
// ...
			},
			PortBindings: map[docker.Port][]docker.PortBinding{
				"9092/tcp": {
					{HostIP: "", HostPort: "9092"},
				},
			},
		},
		func(config *docker.HostConfig) {
			config.AutoRemove = true
			config.RestartPolicy = docker.RestartPolicy{Name: "no"}
		},
	)
	s.Require().NoError(err)
	s.resources = append(s.resources, resource)
	time.Sleep(5 * time.Second)
```

```
}
```

Метод викликає функцію «RunWithOptions» що запускає контейнер «bitnami/kafka:3.5.1-debian-11-r7» які́й містить Apache Kafka.

З метод перевірки роботи компоненту потрібно виконати тести. Щоб запустити тести потрібно виконати команду «go test ./bridge/...» що запустить всі наявні тести. Результат роботи вказано у лістингу 4.5.

Лістинг 4.5 — Виконання тестів

```
$ go test ./bridge/...
ok      github.com/RidgeA/masters/bridge        (6.315s)
```

4.2 Верифікація сумісності з різними інфраструктурними компонентами.

З метою проведення верифікації сумісності розробленого прототипу з іншими інфраструктурними рішеннями перевіримо, чи можливо створити інший тип обмінника, що використовує інший сервіс у якості інфраструктурного шару. У якості такого інфраструктурного рішення було обрано RabbitMQ що реалізує протокол AMQP 0.9.1.

Спроможність розширити функціональність системи шляхом інтеграції нового інфраструктурного компоненту підтвердить універсальність рішення та можливість подальшої адаптації системи до нових інфраструктурних компонент.

Реалізуємо новий тим обмінника. Код декларації наведено у лістингу 4.6.

Лістинг 4.6 — Код amqpExchange

```
type amqpExchange struct {
        channel *amqp091.Channel
}

type AMQPConfig struct {
        Queue string `json:"queue"`
}

func NewAMQP(url string) (Connector, error) {
        conn, err := amqp091.Dial(url)
```

```
	if err != nil {
		return nil, err
	}

	channel, err := conn.Channel()
	if err != nil {
		return nil, err
	}
	return &amqpExchange{
		channel: channel,
	}, nil
}

func (a *amqpExchange) Connect(ctx context.Context, cp ConfigProvider) (Exchange, error) {
	cfg := AMQPConfig{}
	if err := cp.Configure(&cfg); err != nil {
		return nil, err
	}

	return &amqpHandler{
		channel: a.channel,
		cfg:     cfg,
	}, nil
}

type amqpHandler struct {
	channel *amqp091.Channel
	cfg     AMQPConfig
}
```

Для того, що цей обмінник можна було використати потрібно імплементувати інтерфейс «Exchange», який наведено у лістингу 3.7. Код реалізації методу «Publish» для обмінника, що використовує RabbitMQ наведено у лістингу 4.7.

Лістинг 4.7 — Метод Publish AMQP обмінника

```
func (a *amqpHandler) Publish(ctx context.Context, message Message) error {

	headers := map[string]any{}
	for key, value := range message.Metadata() {
		headers[key] = value
```

```
	}

	return a.channel.PublishWithContext(
		ctx,
		"",
		a.cfg.Queue,
		true,
		false,
		amqp091.Publishing{
			DeliveryMode: amqp091.Persistent,
			Headers:     headers,
			Body:        message.Payload(),
		},
	)
}
```

У цьому методі ми спочатку маємо перетворити метадані повідомлення до типу, який потрібно використати для виклику методу бібліотеки взаємодії з RabbitMQ.

Після цього виконується виклик функції «PublishWithContext» куди передаються наступні параметри:

— контекст що використовується для виклику методу, «ctx»;

— назва AMQP обмінника куди потрібно надіслати повідомлення, у нашому випадку ми будемо використовувати обмінник по замовченню, тому вказуємо пустий рядок — «""»;

— назва черги, куди потрібно надіслати повідомлення, «a.cfg.Queue»;

— обов'язковість доставки — повідомлення не буде доставлено, якщо немає черги, яке це повідомлення має отримати, вказуємо «true»;

— миттєва доставка — якщо вказано «false» то повідомлення буде збережено у сервері черг до поки не з'явиться читач повідомлення;

— власне повідомлення що потрібно відправити з заголовками.

Для отримання повідомлень використовується метод «Subscribe». Код цього методу для AMQP обмінника наведено у лістингу 4.8.

Лістинг 4.8 — Метод Subscribe AMQP обмінника

```
func (a *amqpHandler) Subscribe(ctx context.Context, subscription Subscription)
error {

	if err := a.channel.Qos(1, 0, false); err != nil {
		return err
	}

	consumeCh, err := a.channel.ConsumeWithContext(ctx, a.cfg.Queue, "", false,
false, false, false, nil)
	if err != nil {
		return err
	}

	for delivery := range consumeCh {
		md := Metadata{}
		for key, value := range delivery.Headers {
			md[key] = value.(string)
		}
		rm := NewReceivedMessage(md, delivery.Body)

		err = subscription.OnMessage(ctx, rm)
		if err != nil {
			return err
		}
		err = delivery.Ack(false)
		if err != nil {
			return err
		}
	}

	return nil
}
```

У цьому методі спочатку викликається метод «Qos» яким визначається кількість повідомлень які будуть читатись з черги бібліотекою для взаємодії з RabbitMQ.

Після цього викликається метод «ConsumeWithContext» у який ми передаємо контекст, назву черги та інші параметри. Метод повертає Golang-канал з якого відбувається читання повідомлень.

Для верифікації що цей код працює реалізуємо тест. Цей тест буде схожий

на вже існуючий, що було наведено у лістингу 4.3, але замість Kafka він буде використовувати RabbitMQ.

Спочатку реалізуємо допоміжний метод «withAMQP» який створить Docker-контейнер з RabbitMQ, код методу наведено у лістингу 4.9.

Лістинг 4.9 — Метод withAMQP

```
func (s *testE2ESuite) withAMQP() {
	resource, err := s.pool.RunWithOptions(
		&dockertest.RunOptions{
			Name:       "rabbitmq",
			Repository: "rabbitmq",
			Tag:        "3.12.9-management",
			PortBindings: map[docker.Port][]docker.PortBinding{
				"5672/tcp": {
					{HostIP: "", HostPort: "5672"},
				},
			},
		},
		func(config *docker.HostConfig) {
			config.AutoRemove = true
		},
	)
	s.Require().NoError(err)
	s.resources = append(s.resources, resource)
	time.Sleep(5 * time.Second)

	conn, err := amqp091.Dial("amqp://guest:guest@localhost:5672/")
	s.Require().NoError(err)
	defer conn.Close()

	channel, err := conn.Channel()
	s.Require().NoError(err)

	_, err = channel.QueueDeclare("queue", true, false, false, false, nil)
	s.Require().NoError(err)
}
```

Код тесту, що перевіряє взаємодію з RabbitMQ наведено у лістингу 4.10.

Лістинг 4.10 — Метод TestRabbitMQ

```
func (s *testE2ESuite) TestRabbitMQ() {
```

```
	s.withAMQP()

	b := bridge.New(bridge.Config{
		Routes: []bridge.RouteConfig{
			{
				Condition: &bridge.EverythingCondition{},
				Target:    bridge.DirectDestination("amqp_exchange"),
			},
		},
		ExchangeConfigs: 
map[bridge.ExchangeName]bridge.ExchangeDestination{
			"amqp_exchange": {
				Kind:   "amqp",
				Config: toJsonRawMessage(s.T(), 
bridge.AMQPConfig{Queue: "queue"}),
			},
		},
	})

	amqp, err := bridge.NewAMQP("amqp://guest:guest@localhost:5672/")
	s.Require().NoError(err)
	b.RegisterConnector("amqp", amqp)

	msgCh := s.expectMessageOnce(b, "amqp_exchange")

	msg, err := bridge.NewJSONMessage("Hello, World!", 
map[string]string{"foo": "bar"})
	s.Require().NoError(err)

	err = b.Publish(s.ctx, msg)
	s.Require().NoError(err)

	receivedMessage := <-msgCh
	s.Assert().Equal(msg.Metadata(), receivedMessage.Metadata())
	s.Assert().Equal(msg.Payload(), receivedMessage.Payload())
}
```

4.3 Верифікація можливості маршрутизації повідомлень

Можливість маршрутизації повідомлень є важливим аспектом роботи компоненту для ефективного розподілу навантаження, ізоляції кластерів даних та оптимізації процесів тестування. Це забезпечить більш стабільну роботу

системи і підвищить її загальну продуктивність.

В рамках прототипу було реалізовано декілька умов, які дозволяють застосовувати правила маршрутизації на підставі значень метаданих повідомлення.

Для тесту маршрутизації було створено екземпляр компонента асинхронної комунікації, для якого створюється 2 обмінника з правилами маршрутизації. Код створення наведено у лістингу 4.11.

Лістинг 4.11 — Створення компоненту з правилами маршрутизації

```
b := bridge.New(bridge.Config{
	Routes: []bridge.RouteConfig{
		{
			Condition: &bridge.Negate{
				Condition: &bridge.MetadataValueCondition{
					Key:   "route",
					Value: "kafka",
				},
			},
			Target: bridge.DirectDestination("amqp_exchange"),
		},
		{
			Condition: &bridge.MetadataValueCondition{
				Key:   "route",
				Value: "kafka",
			},
			Target: bridge.DirectDestination("kafka_exchange"),
		},
	},
	ExchangeConfigs:
map[bridge.ExchangeName]bridge.ExchangeDestination{
		"amqp_exchange": {
			Kind:   "amqp",
			Config:                              toJsonRawMessage(s.T(),
bridge.AMQPConfig{Queue: "queue"}),
		},
		"kafka_exchange": {
			Kind:   "kafka",
			Config:                              toJsonRawMessage(s.T(),
bridge.KafkaConfig{Topic: "topic"}),
		},
```

```
        },
    })
```

Компонент налаштовано таким чином, щоб повідомлення, які містять ключ «route» зі значенням «kafka» у метаданих, було спрямовано до «kafka_exchange» а якщо ні — «amqp_exchange».

Далі створимо коннектори та зареєструємо їх, відповідний код наведено у лістингу 4.12.

Лістинг 4.12 — Створення та реєстрація коннекторів

```
amqp, err := bridge.NewAMQP("amqp://guest:guest@localhost:5672/")
s.Require().NoError(err)
b.RegisterConnector("amqp", amqp)

kafka, err := bridge.NewKafka([]string{"localhost:9092"})
s.Require().NoError(err)
b.RegisterConnector("kafka", kafka)

amqpMsgCh := s.expectMessageOnce(b, "amqp_exchange")
kafkaMsgCh := s.expectMessageOnce(b, "kafka_exchange")
```

Потім створимо канали що будуть отримувати повідомлення, створимо самі повідомлення та відправимо їх. Код наведено у лістингу 4.13.

Лістинг 4.13 — Створення повідомлень

```
amqpMsgCh := s.expectMessageOnce(b, "amqp_exchange")
kafkaMsgCh := s.expectMessageOnce(b, "kafka_exchange")

msgKafka, err := bridge.NewJSONMessage("Hello, World!", map[string]string{"route": "kafka"})
s.Require().NoError(err)

msgAMQP, err := bridge.NewJSONMessage("Hello, World!", map[string]string{"route": "amqp"})
s.Require().NoError(err)

err = b.Publish(s.ctx, msgKafka)
s.Require().NoError(err)

err = b.Publish(s.ctx, msgAMQP)
s.Require().NoError(err)
```

Врешті решт ми можемо перевірити, що повідомлення було доставлено та вони відповідають очікуванням, відповідний код наведено у лістингу 4.14.

Лістинг 4.14. — Перевірка відповідності повідомлень очікуваним даним

```
amqpMsg := <-amqpMsgCh
s.Assert().Equal(msgAMQP.Metadata(), amqpMsg.Metadata())
s.Assert().Equal(msgAMQP.Payload(), amqpMsg.Payload())

kafkaMsg := <-kafkaMsgCh
s.Assert().Equal(msgKafka.Metadata(), kafkaMsg.Metadata())
s.Assert().Equal(msgKafka.Payload(), kafkaMsg.Payload())
```

5 ЕКОНОМІЧНА ЧАСТИНА

Для успішного впровадження науково-технічної розробки вирішально важливим є її відповідність сучасним стандартам науково-технічного прогресу та врахування економічних аспектів. Ключовим етапом цього процесу є проведення оцінки економічної ефективності отриманих результатів науково-дослідної роботи.

Дослідження, яке представлене у магістерській роботі та присвячене розробці та аналізу "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури", віднесено до проектів, спрямованих на виведення на ринок. Виведення на ринок може бути визначено під час виконання самої роботи і розглядається як етап комерціалізації науково-технічної розробки. Цей напрямок розглядається як пріоритетний, оскільки отримані результати можуть бути корисними для різних зацікавлених сторін і приносити економічні вигоди.

Проте для успішної реалізації цього процесу вирішальним є здійснення пошуку зацікавленого інвестора, який виявить інтерес до втілення даного проекту, та переконання його у доцільності вкладання інвестицій у цю розробку. З цією метою були визначені наступні етапи виконання робіт:

— проведення комерційного аудиту науково-технічної розробки, включаючи визначення науково-технічного рівня та комерційного потенціалу;

— розрахунок витрат на реалізацію науково-технічної розробки;

— проведення розрахунку економічної ефективності впровадження та комерціалізації науково-технічної розробки для потенційного інвестора, а також обґрунтування економічної доцільності комерціалізації з точки зору інвестора.

5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки

Метою проведення комерційного і технологічного аудиту дослідження за

темою "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" .

Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням 5-ти бальної системи оцінювання за 12-ма критеріями, наведеними в таблиці в Додатку Д [19].

Для оцінки науково-технічного рівня і комерційного потенціалу розробки експертами було запрошено трьох незалежних експертів з Вінницького національного технічного університету, кафедри «Обчислювальної техніки»: Мартинюк Тетяна Борисівна, доктор технічних наук, професор; Крупельницький Леонід Віталійович кандидат технічних наук, доцент; Богомолов Сергій Віталійович кандидат технічних наук, старший викладач.

Таблиця 5.1 — Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт		
	Мартинюк Тетяна Борисівна	Крупельницький Леонід Віталійович	Богомолов Сергій Віталійович
	Бали:		
1. Технічна здійсненність концепції	4	3	3
2.Ринкові переваги (наявність аналогів)	2	2	2
3. Ринкові переваги (ціна продукту)	4	4	4
4. Ринкові переваги (технічні властивості)	4	3	2
5. Ринкові переваги (експлуатаційні витрати)	4	4	4
6. Ринкові перспективи (розмір ринку)	3	2	2
7. Ринкові перспективи (конкуренція)	3	3	3
8. Практична здійсненність (наявність фахівців)	4	4	4
9. Практична здійсненність (наявність фінансів)	4	4	4
10. Практична здійсненність (необхідність нових матеріалів)	4	4	4
11. Практична здійсненність (термін реалізації)	4	4	4
12. Практична здійсненність (розробка документів)	4	4	41
Сума балів	$СБ_1=44$	$СБ_2=41$	$СБ_3=40$
Середньоарифметична сума балів $СБ_с$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{44+41+40}{3} = 42$		

За результатами розрахунків, наведених в таблиці 5.1, зробимо висновок щодо науково-технічного рівня і рівня комерційного потенціалу розробки. При цьому використаємо рекомендації, наведені в таблиці 5.2.

Таблиця 5.2 — Науково-технічні рівні та комерційні потенціали розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Науково-технічний рівень та комерційний потенціал розробки
41…48	Високий
31…40	Вище середнього
21…30	Середній
11…20	Нижче середнього
0…10	Низький

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" становить 42 бали, що, відповідно до таблиці 5.2 рівень комерційного потенціалу розробки високий, що свідчить про комерційну важливість проведення даних досліджень.

5.2 Розрахунок витрат на проведення науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи на тему "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури", під час планування, обліку і калькулювання собівартості науково-дослідної роботи групуємо за відповідними статтями.

5.2.1 Витрати на оплату праці

До статті «Витрати на оплату праці» належать витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп, науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми,

обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці.

Витрати на основну заробітну плату дослідників ($З_о$) розраховуємо у відповідності до посадових окладів працівників, за формулою [19]:

$$З_o = \sum_{i=1}^{k} \frac{M_{ni} \cdot t_i}{T_p}, \quad (5.1)$$

де k – кількість посад дослідників залучених до процесу досліджень;

M_{ni} – місячний посадовий оклад конкретного дослідника, грн;

t_i – число днів роботи конкретного дослідника, дн.;

T_p – середнє число робочих днів в місяці, T_p=21 дні.

$$З_о = 18000 \cdot 5 / 21 = 4091 \text{ грн.}$$

Проведені розрахунки зведемо до таблиці 5.3

Таблиця 5.3 — Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Керівник проекту	18000	818,2	5	4091
Інженер-програміст	20000	909,1	35	31818
Всього				35909

Додаткову заробітну плату розраховуємо як 10 … 12% від суми основної заробітної плати дослідників та робітників за формулою:

$$З_{дод} = (З_o + З_p) \cdot \frac{H_{дод}}{100\%}, \quad (5.2)$$

де $H_{дод}$ – норма нарахування додаткової заробітної плати. Приймемо 11%.

$З_{дод}$ = (35909) · 11 / 100% = 3950 грн.

5.2.2 Відрахування на соціальні заходи

Нарахування на заробітну плату дослідників та робітників розраховуємо як 22% від суми основної та додаткової заробітної плати дослідників і робітників за формулою:

$$З_н = (З_о + З_р + З_{дод}) \cdot \frac{Н_{зп}}{100\%} \qquad (5.3)$$

де $Н_{зп}$ – норма нарахування на заробітну плату. Приймаємо 22%.

Зн = (35909+3950) · 22 / 100% = 8769 грн.

5.2.3 Сировина та матеріали

До статті «Сировина та матеріали» належать витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби і предмети праці, які придбані у сторонніх підприємств, установ і організацій та витрачені на проведення досліджень за темою "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури".

Витрати на матеріали (*М*), у вартісному вираженні розраховуються окремо по кожному виду матеріалів за формулою:

$$M = \sum_{j=1}^{n} Н_j \cdot Ц_j \cdot К_j - \sum_{j=1}^{n} В_j \cdot Ц_{в j}, \qquad (5.4)$$

де $Н_j$ – норма витрат матеріалу *j*-го найменування, кг;

n – кількість видів матеріалів;

$Ц_j$ – вартість матеріалу *j*-го найменування, грн/кг;

$К_j$ – коефіцієнт транспортних витрат, ($К_j = 1,1 \dots 1,15$);

B_j – маса відходів j-го найменування, кг;

$Ц_{вj}$ – вартість відходів j-го найменування, грн/кг.

Проведені розрахунки зведемо до таблиці 5.4.

Таблиця 5.4 — Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за 1 кг, грн	Норма витрат, кг	Вартість витраченого матеріалу, грн
Папір А4	170	1	170
Ручка	30	1	30
Диск оптичний CD	17	1	17
Flash-пам'ять 64	346	1	346
Всього			563
З врахуванням коефіцієнта транспортування			619,3

5.2.4 Розрахунок витрат на комплектуючі

Витрати на комплектуючі *($К_в$)*, які використовують при проведенні НДР на тему "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" відсутні.

5.2.5 Спецустаткування для наукових (експериментальних) робіт

До статті «Спецустаткування для наукових (експериментальних) робіт» належать витрати на виготовлення та придбання спецустаткування необхідного для проведення досліджень, також витрати на їх проектування, виготовлення, транспортування, монтаж та встановлення в роботі відсутні.

5.2.6 Програмне забезпечення для наукових (експериментальних) робіт

До статті «Програмне забезпечення для наукових (експериментальних) робіт» належать витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення, (програм, алгоритмів, баз даних) необхідних для проведення досліджень, також витрати на їх проектування, формування та встановлення. Програмні засоби, які були використанні при написанні

магістерської роботи є безкоштовними.

5.2.7 Амортизація обладнання, програмних засобів та приміщень

В спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню тощо, розраховуємо з використанням прямолінійного методу амортизації за формулою:

$$А_{обл} = \frac{Ц_б}{Т_в} \cdot \frac{t_{вик}}{12}, \quad (5.5)$$

де $Ц_б$ – балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ – термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$Т_в$ – строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

$$А_{обл} = (25000 \cdot 1) / (2 \cdot 12) = 1041{,}67 \text{ грн.}$$

Проведені розрахунки зведемо до таблиці 5.5.

Таблиця 5.5 — Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Комп'ютер	25000	2	1	1041,67
Приміщення лабораторії	250000	20	1	1041,67
Всього				2083,33

5.2.8 Паливо та енергія для науково-виробничих цілей

Витрати на силову електроенергію ($В_е$) розраховуємо за формулою:

$$В_е = \sum_{i=1}^{n} \frac{W_{уi} \cdot t_i \cdot Ц_е \cdot К_{впi}}{\eta_i}, \qquad (5.6)$$

де W_{yi} – встановлена потужність обладнання на визначеному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_е$ – вартість 1 кВт-години електроенергії, грн; (вартість електроенергії визначається за даними енергопостачальної компанії), приймемо $Ц_е$ = 7,5 грн;

$К_{впi}$ – коефіцієнт, що враховує використання потужності, $К_{впi} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

$$В_е = 0{,}25 \cdot 240 \cdot 7{,}5 \cdot 0{,}5 / 0{,}8 = 281{,}25 \text{ грн.}$$

5.2.9 Службові відрядження

До статті «Службові відрядження» дослідної роботи на тему "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" належать витрати на відрядження штатних працівників, працівників організацій, які працюють за договорами цивільно-правового характеру, аспірантів, зайнятих розробленням досліджень, відрядження, пов’язані з проведенням випробувань машин та приладів, а також витрати на відрядження на наукові з’їзди, конференції, наради, пов’язані з виконанням конкретних досліджень.

Витрати за статтею «Службові відрядження» розраховуємо як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$В_{св} = (З_о + З_р) \cdot \frac{Н_{св}}{100\%}, \qquad (5.7)$$

де $Н_{св}$ – норма нарахування за статтею «Службові відрядження», приймемо $Н_{св}$ = 20%.

$$В_{св} = (35909) \cdot 20 / 100\% = 7181{,}82 \text{ грн.}$$

5.2.10 Витрати на роботи, які виконують сторонні підприємства, установи і організації

Витрати за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації» відсутні.

5.2.11 Інші витрати

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуємо як 50…100% від суми основної заробітної плати дослідників та робітників за формулою:

$$I_{в} = (З_{о} + З_{р}) \cdot \frac{Н_{ів}}{100\%}, \qquad (5.8)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати», приймемо $Н_{ів} = 50\%$.

$$І_{в} = (35909) \cdot 50 / 100\% = 17954{,}55 \text{ грн.}$$

5.2.12 Накладні (загальновиробничі) витрати

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов’язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов’язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов’язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуємо

як 100…150% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \qquad (5.9)$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати», приймемо $Н_{нзв}$ = 100%.

$В_{нзв}$ = (35909) · 100 / 100% = 35909,09 грн.

Витрати на проведення науково-дослідної роботи на тему "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури". розраховуємо як суму всіх попередніх статей витрат за формулою:

$$B_{заг} = З_о + З_р + З_{дод} + З_н + М + К_в + B_{спец} + B_{прг} + А_{обл} + B_е + B_{св} + B_{сп} + І_в + B_{нзв}. \qquad (5.10)$$

$В_{заг}$=35909+3950+8769+619,3+2083,33+281,25+7181,82+17954,55+35909,09 =112657,43 грн.

Загальні витрати *ЗВ* на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховується за формулою:

$$ЗВ = \frac{B_{заг}}{\eta}, \qquad (5.11)$$

де η - коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи, приймемо η=0,7.

ЗВ = 112657,43 / 0,7 = 160939,18 грн.

5.3 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку.

Результати дослідження проведені за темою "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" передбачають комерціалізацію протягом 3-х років реалізації на ринку.

В цьому випадку основу майбутнього економічного ефекту будуть формувати:

ΔN – збільшення кількості споживачів яким надається відповідна інформаційна послуга у періоди часу, що аналізуються;

N– кількість споживачів яким надавалась відповідна інформаційна послуга у році до впровадження результатів нової науково-технічної розробки, приймемо 1 особа

$Ц_б$ – вартість послуги у році до впровадження інформаційної системи, приймемо 3000,00 грн;

$\pm \Delta Ц_о$ – зміна вартості послуги від впровадження результатів, приймемо зростання на 500,00 грн.

Можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ для кожного із 3-х років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховуємо за формулою [19]:

$$\Delta П_i = (\pm \Delta Ц_о \cdot N + Ц_о \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (5.12)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2021 році ставка податку на додану вартість складає 20%, а коефіцієнт λ*=0,8333*;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту). Приймемо ρ=40*%*;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2023 році ϑ*=18%*;

Збільшення чистого прибутку 1-го року:

$\Delta П_1 =$(1·500+3000·650)·0,83·0,4·(1-0,18/100%)=388715,7 грн.

Збільшення чистого прибутку 2-го року:

$\Delta П_2 =$(1·500+3000·(650+600))·0,83·0,4·(1-0,18/100%)=747865,94 грн.

Збільшення чистого прибутку 3-го року:

$\Delta П_3 =$(1·500+3000·(650+600+550))·0,83·0,4·(1-0,18/100%)=1076707 грн.

Приведена вартість збільшення всіх чистих прибутків *ПП*, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \quad (5.13)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному з років, протягом яких

виявляються результати впровадження науково-технічної розробки, грн;

T – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ=18%;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$ПП$ =388715,7 /(1+0,18)^1+747865,94 /(1+0,18)^2+1076707 /(1+0,18)^3=
= 1469273,51 грн.

Величина початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки:

$$PV = k_{інв} \cdot ЗВ, \tag{5.14}$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$=2;

$ЗВ$ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо грн. 160939,18

$PV = k_{інв} \cdot ЗВ$= 2 · 160939,18 = 321878,37 грн.

Абсолютний економічний ефект $E_{абс}$ для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV \tag{5.15}$$

де $ПП$ – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, 1469273,51 грн;

PV – теперішня вартість початкових інвестицій 321878,37 грн.

$$E_{абс} = ПП - PV = 1469273{,}51 - 321878{,}37 = 1147395{,}14 \text{ грн.}$$

Внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$E_в = \sqrt[Тж]{1 + \frac{E_{абс}}{PV}} - 1, \tag{5.16}$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, грн;

PV – теперішня вартість початкових інвестицій, грн;

$T_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, 3 роки.

$$E_в = \sqrt[Тж]{1 + \frac{E_{абс}}{PV}} - 1 = (1+1147395{,}14 /321878{,}37)^{1/3} - 1 = 1{,}01.$$

Мінімальна внутрішня економічна дохідність вкладених інвестицій $\tau_{мін}$:

$$\tau_{мін} = d + f, \tag{5.17}$$

де d – середньозважена ставка за депозитними операціями в комерційних

банках; в 2023 році в Україні d =0,1;

f – показник, що характеризує ризикованість вкладення інвестицій, приймемо 0,25.

$\tau_{мін}$= 0,1+0,25 = 0,35 < 1,01 свідчить про те, що внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки вища мінімальної внутрішньої дохідності. Тобто інвестувати в науково-дослідну роботу за темою "Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури" доцільно.

Період окупності інвестицій $T_{ок}$ які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$T_{ок} = \frac{1}{E_в}, \tag{5.18}$$

де $E_в$ – внутрішня економічна дохідність вкладених інвестицій.

$T_{ок}$= 1 / 1,01 = 1 р.

$T_{ок}$ < 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

ВИСНОВКИ

У рамках даної магістерської роботи було здійснено всебічне дослідження актуальності асинхронної комунікації у контексті розподілених систем. Актуальність теми підтверджується швидким розвитком інформаційних технологій та зростаючими вимогами до обробки великих обсягів даних, що вимагає більш ефективних та гнучких підходів до комунікації. В результаті дослідження було виявлено, що тема асинхронної комунікації у розподілених системах є вкрай важливою та перспективною для подальшого розвитку.

Теоретичний аспект роботи включав глибоке дослідження принципів асинхронної комунікації, охоплюючи аналіз таких ключових елементів, як вебхуки, спільні файлові сховища та черги повідомлень. Це дозволило не тільки глибше зрозуміти різні механізми та архітектурні підходи до асинхронної комунікації, але й виявити їх практичну застосовність у розподілених системах.

Основною частиною роботи стало розроблення інформаційної технології, заснованої на асинхронній взаємодії в рамках мікросервісної архітектури. Ця технологія дозволяє ефективно інтегрувати різноманітні інфраструктурні компоненти, забезпечуючи високу пропускну спроможність, надійність та гнучкість системи. Важливим аспектом є можливість налаштування маршрутизації повідомлень, що дозволяє адаптувати систему під специфічні потреби та умови використання.

Для підтвердження теоретичних розробок та технічної можливості реалізації інформаційної технології було створено прототип системи. Результати тестування прототипу продемонстрували його високу ефективність та практичну застосовність у різних сценаріях використання, що відкриває шлях для його подальшої адаптації та імплементації в реальних бізнес-процесах.

Загалом, результати цієї магістерської роботи вносять вклад у розвиток інформаційних технологій, особливо у сфері асинхронної комунікації у розподілених системах. Робота не тільки розкриває теоретичні аспекти теми, але й надає практичні рішення для її застосування, що робить її цінною як для

наукового співтовариства, так і для практичної діяльності в галузі інформаційних технологій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Press Release Gartner Forecasts Global Devices Installed Base to Reach 6.2 Billion Units in 2021 [Електронний ресурс] — Режим доступу: https://www.gartner.com/en/newsroom/press-releases/2021-04-01-gartner-forecasts-global-devices-installed-base-to-reach-6-2-billion-units-in-2021.

2. Maarten van Steen Distributed Systems Third edition / Maarten van Steen Andrew S. Tanenbaum — Pearson Education, Inc. 2020 — 598 с.

3. DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition / [George Coulouris, Jean Dollimore, Tim Kindberg, Golangrdon Blair] — Addison-Wesley 2012 — 1047 с.

4. About webhooks [Електронний ресурс] — Режим доступу: https://docs.github.com/en/webhooks/about-webhooks

5. Webhooks [Електронний ресурс] — Режим доступу: https://developer.atlassian.com/server/jira/platform/webhooks/

6. Sending messages using Incoming Webhooks [Електронний ресурс] — Режим доступу: https://api.slack.com/messaging/webhooks

7. The OAuth 2.0 Authorization Framework [Електронний ресурс] — Режим доступу: https://datatracker.ietf.org/doc/html/rfc6749

8. Sam Newman. Building microservices Designing Fine-Grained Systems Second Edition / Sam Newman; O`Reilly Media Inc. 2021 — 586 с.

9. Martin Kleppmann. Designing Data-Intensive Applications; O`Reilly Media Inc. 2017 — 590 с.

10. DynamoDB Streams and AWS Lambda triggers [Електронний ресурс] — Режим доступу: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.html

11. Change streams overview [Електронний ресурс] — Режим доступу: https://cloud.Golangogle.com/bigtable/docs/change-streams-overview

12. Change feed design patterns in Azure Cosmos DB [Електронний ресурс]

— Режим доступу: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/change-feed-design-patterns?tabs=latest-version

13. Amazon S3 Event Notifications [Електронний ресурс] — Режим доступу: https://docs.aws.amazon.com/AmazonS3/latest/userguide/EventNotifications.html

14. Pub/Sub notifications for Cloud Storage [Електронний ресурс] — Режим доступу: https://cloud.Golangogle.com/storage/docs/pubsub-notifications

15. Reacting to Blob storage events [Електронний ресурс] — Режим доступу: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-event-overview

16. Message Delivery Guarantees [Електронний ресурс] — Режим доступу: https://docs.confluent.io/kafka/design/delivery-semantics.html

17. The Big Little Guide to Message Queues [Електронний ресурс] — Режим доступу: https://sudhir.io/the-big-little-guide-to-message-queues

18. Kafka Transactions: Part 1: Exactly-Once Messaging [Електронний ресурс] — Режим доступу: https://www.linkedin.com/pulse/kafka-transactions-part-1-exactly-once-messaging-rob-Golanglder/

19. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
проф., д.т.н.. Азаров О.Д..
"29" вересня 2023 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи
“Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури”
08-54.МКР.003.00.000 ТЗ

Науковий керівник: к.т.н., доц. каф. ОТ
_________ Кадук О. В.
Студент групи 1КІ-22м
________ Грядченко А. О.

ВНТУ 2022

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

1.1 Важливим є актуальність дослідження у напрямку магістерської роботи, яка обумовлена тим, що в даний час розподілені системи застосовуються все більше, як відповідь на виклик зростаючої складності комп'ютерних систем. Асинхронна комунікація є важливою складовою побудови розподілених систем, адже такий спосіб комунікації надає можливість масштабувати розподілені системи та обробляти більший потік даних.

1.2 Наказ про затвердження теми МКР.

2 Мета МКР і призначення розробки

2.1 Мета роботи — вдосконалення процесу розробки розподілених систем що використовують асинхронні методи комунікації шляхом розробки інформаційної технології, яка спрощує інтеграцію компонент розподіленої системи.

2.2 Призначення розробки — запропонувати інформаційну систему асинхронної комунікації що покращує процес інтеграції інфраструктурних рішень для обміну повідомленнями. Система має дозволяє легко інтегрувати різні типи інфраструктурних рішень шляхом зміни конфігурації. Це дозволить замінювати інфраструктурне рішення тим самим оптимізуючи витрати та зменшуючи ризики залежності від одного постачальника послуг. Крім того, така система має підтримувати маршрутизацію повідомлень, що має дозволити гнучко спрямовувати повідомлення до різних отримувачів відповідно до налаштувань.

3 Вихідні дані для виконання МКР

Провести аналіз розповсюдження розподілених систем здійснити огляд наявних методів асинхронної комунікації, змоделювати сценарії використання.

4 Вимоги до виконання МКР

Розробити прототип інформаційної системи яка дозволить використовувати різні інфраструктурні рішення у якості транспортного шару, система має надавати можливість маршрутизації повідомлень.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз актуальності теми			Розділ 1
2	Аналіз наявних способів асинхронної взаємодії			Розділ 2
3	Моделювання сценаріїв використання			Частина розділу 3
4	Вибір технологічного стеку			Частина розділу 3
5	Створення коду прототипу інформаційної системи			Розділ 4
6	Розрахунок економічної частини			Розділ 5
7	Оформлення пояснювальної записки та ілюстративного матеріалу			ПЗ, графічний матеріал
8	Підготовка і підпис супроводжуючих документів, нормоконтроль та тест на плагіат			Оформлені документи

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — «Комп'ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21».

ДОДАТОК Б

Порівняльна таблиця мов програмування

Таблиця Б.1 — Характеристики мов програмування

	Рівень абстракції	Керування пам’яттю	Підтримка багатопотоковості	Сумісність з різними платформами	Відомі продукти, що використовують мову
Java	Високий рівень абстракції	Автоматичне керування пам’яттю	Так	Підтримується велика кількість платформ на рівні середовища виконання.	Apache Kafka, Apache Cassandra
C#	Високий рівень абстракції	Автоматичне керування пам’яттю	Так	Підтримується велика кількість платформ на рівні середовища виконання.	Unity Engine, Visual Studio, Orleans
Node.js	Високий рівень абстракції	Автоматичне керування пам’яттю	Обмежена підтримка	Підтримується велика кількість платформ на рівні середовища виконання.	PayPal, LinkedIn Socket.IO
Golang	Високий рівень абстракції	Автоматичне керування пам’яттю	Так, через механізм горутин.	Код компілюється під цільову платформу	Kubernetes, Docker
C++	Низький рівень абстракції	Ручне керування пам’яттю	Так	Код компілюється під цільову платформу	Adobe Photoshop, Microsoft Office
Rust	Середній рівень абстракції	Ручне керування пам’яттю.	Так	Код компілюється під цільову платформу	Firefox, Dropbox
Python	Високий рівень абстракції	Автоматичне керування пам’яттю	Обмежена підтримка	Підтримується велика кількість платформ на рівні середовища виконання.	Instagram, Spotify.

ДОДАТОК В

Схема взаємодії при надсиланні повідомлення

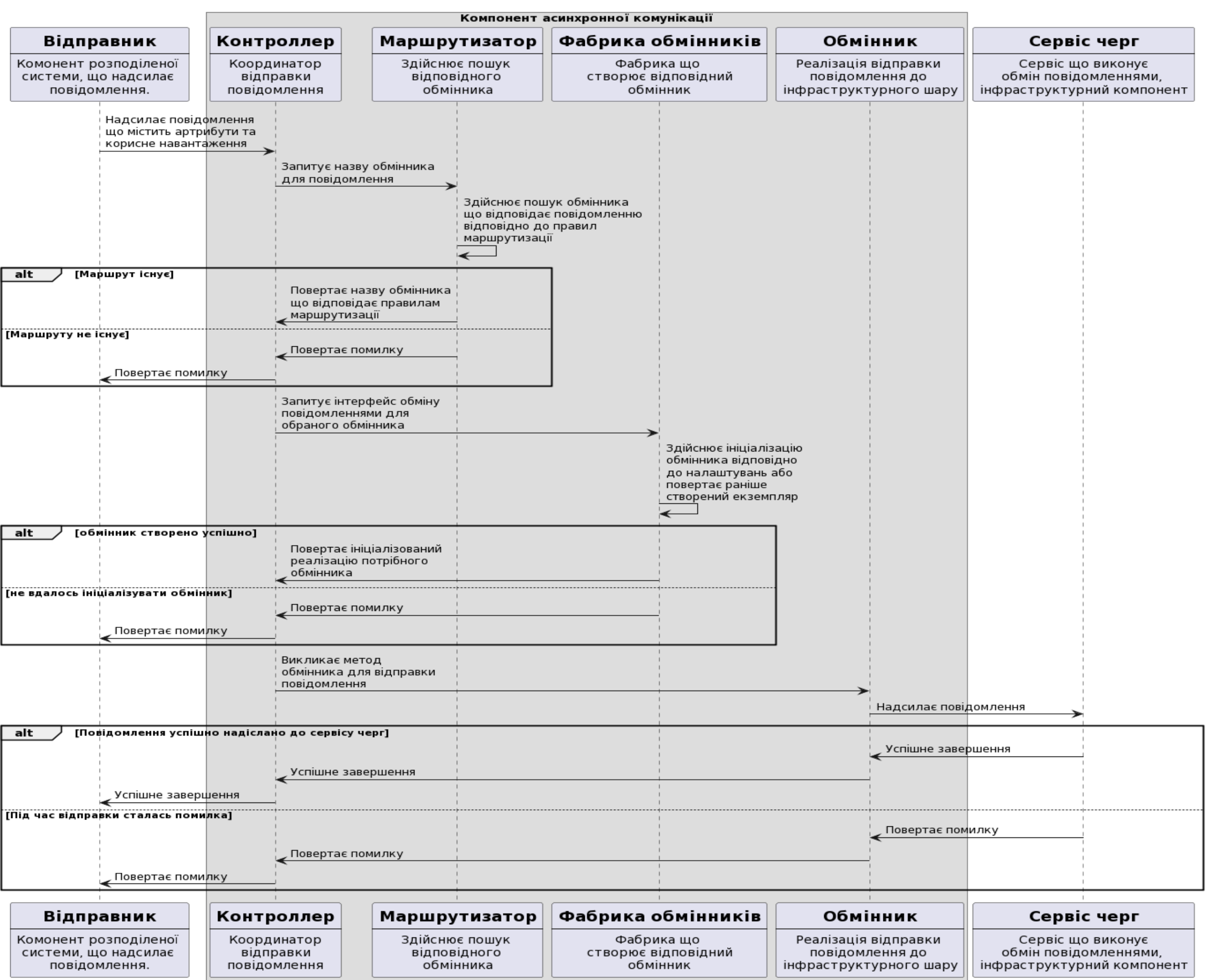

Рисунок В.1 — Схема взаємодії при надсилання повідомлення

ДОДАТОК Г

Схема взаємодії при підписці на повідомлення

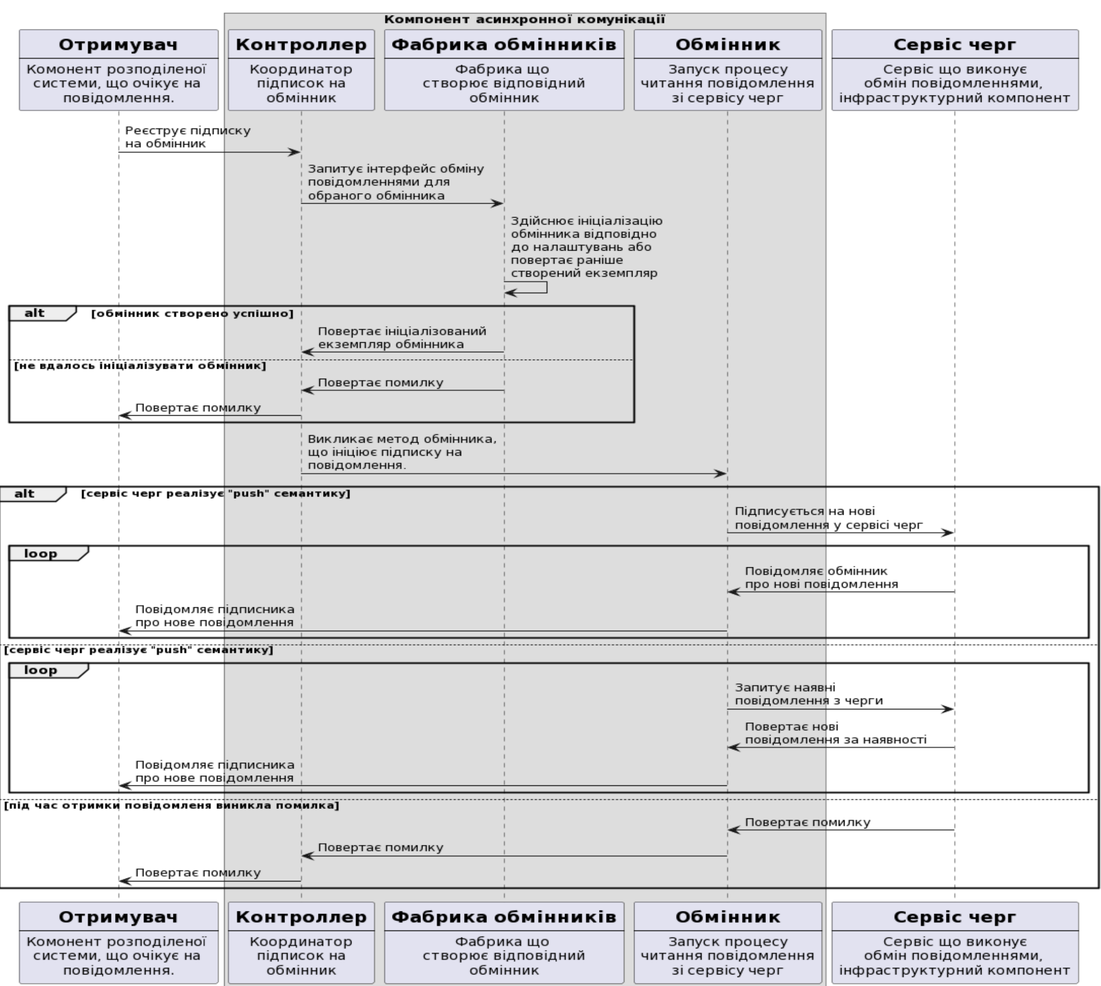

Рисунок В.1 — Схема взаємодії при підписці на повідомлення

ДОДАТОК Д

Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка

Таблиця Д.1 — Критерії оцінювання

	Бали (за 5-ти бальною шкалою)				
	0	1	2	3	4
	Технічна здійсненність концепції				
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено працездатність продукту в реальних умовах
	Ринкові переваги (недоліки)				
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
	Ринкові перспективи				
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
	Практична здійсненність				
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовують ся у військово промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь- які регламентні обмеження на виробництво та реалізацію продукту

ДОДАТОК Е

Початковий код програми

Лістинг Е.1 — початковий код

```
package bridge

import (
"context"
)

type (
ExchangeName string

Subscription interface {
	OnMessage(context.Context, Message) error
}

Bridge struct {
	router  *routeResolver
	factory *exchangeFactory
}
)

func (b *Bridge) Publish(ctx context.Context, message Message) (err error) {

var dest Destination
if dest, err = b.router.FindMatching(ctx, message); err != nil {
	return err
}
```

```
exchangeName := dest.Exchange()

var e Exchange
if e, err = b.factory.CreateExchange(ctx, exchangeName); err != nil {
	return err
}

return e.Publish(ctx, message)
}

func (b *Bridge) Subscribe(ctx context.Context, name ExchangeName, consumer Subscription) (err error) {
var e Exchange
e, err = b.factory.CreateExchange(ctx, name)
if err != nil {
	return err
}

return e.Subscribe(ctx, consumer)
}

func (b *Bridge) RegisterConnector(destinationType ExchangeKind, connector Connector) {
b.factory.RegisterConnector(destinationType, connector)
}

func New(cfg Config) *Bridge {
rCfg := RouterConfig{
	Routes: cfg.Routes,
}
```

```
fCfg := FactoryConfig{
	ExchangeConfigs: cfg.ExchangeConfigs,
}

r := newRouter(rCfg)
f := newExchangeFactory(fCfg)

return &Bridge{
	router:  r,
	factory: f,
}
}

type SubscriptionFunc func(context.Context, Message) error

func (cf SubscriptionFunc) OnMessage(ctx context.Context, msg Message) error
{
return cf(ctx, msg)
}
package bridge

import (
"encoding/json"
"errors"
"reflect"
)

type Condition interface {
Satisfy(msg Message) (bool, error)
```

```
}

type EverythingCondition struct{}

func (m EverythingCondition) Satisfy(_ Message) (bool, error) {
return true, nil
}

type MetadataValueCondition struct {
Key   string
Value any
}

func (m *MetadataValueCondition) UnmarshalJSON(bytes []byte) error {
mm := make(map[string]string)
err := json.Unmarshal(bytes, &mm)
if err != nil {
	return err
}

m.Key = mm["Key"]

var isValue bool
if m.Value, isValue = mm["Value"]; !isValue {
	return errors.New("no Value")
}

return nil
}
```

```
func (m *MetadataValueCondition) Satisfy(msg Message) (bool, error) {
value, exists := msg.Metadata()[m.Key]
if !exists {
	return false, nil
}

return reflect.DeepEqual(value, m.Value), nil
}

type Negate struct {
Condition Condition
}

func (m Negate) Satisfy(msg Message) (bool, error) {
satisfy, err := m.Condition.Satisfy(msg)
if err != nil {
	return false, err
}
return !satisfy, nil
}

type All struct {
conditions []Condition
}

func (m All) Satisfy(msg Message) (bool, error) {
for _, condition := range m.conditions {
	satisfy, err := condition.Satisfy(msg)
	if err != nil {
		return false, err
```

```
		}
		if !satisfy {
			return false, nil
		}
	}
	return true, nil
	}
	package bridge

	import (
	"encoding/json"
	"fmt"
	)

	type (
	Config struct {
		Routes          []RouteConfig                        `json:"routes"`
		ExchangeConfigs                    map[ExchangeName]ExchangeDestination
`json:"exchanges"`
	}

	ExchangeDestination struct {
		Kind   ExchangeKind    `json:"kind"`
		Config json.RawMessage `json:"config"`
	}
	)

	func (td *ExchangeDestination) Configure(value any) error {
	return json.Unmarshal(td.Config, value)
	}
```

```go
func (td *ExchangeDestination) UnmarshalJSON(data []byte) error {
m := map[string]json.RawMessage{}
err := json.Unmarshal(data, &m)
if err != nil {
	return err
}

if err = unmarshall(m, "kind", &td.Kind, true); err != nil {
	return err
}

return unmarshall(m, "config", &td.Config, true)
}

func unmarshall(m map[string]json.RawMessage, field string, target any, mandatory bool) error {
v, exists := m[field]
if !exists && mandatory {
	return fmt.Errorf("no %q", field)
} else if !exists {
	return nil
}

err := json.Unmarshal(v, target)
if err != nil {
	return fmt.Errorf("failed to unmarshal %q %w", field, err)
}
return nil
}
```

```
package bridge_test

import (
"context"
"encoding/json"
"github.com/RidgeA/masters/bridge"
"github.com/ory/dockertest/v3"
"github.com/ory/dockertest/v3/docker"
"github.com/rabbitmq/amqp091-go"
"github.com/stretchr/testify/require"
"github.com/stretchr/testify/suite"
"sync"
"testing"
"time"
)

type testE2ESuite struct {
suite.Suite
pool      *dockertest.Pool
ctx       context.Context
resources []*dockertest.Resource
}

func TestE2E(t *testing.T) {
suite.Run(t, new(testE2ESuite))
}

func (s *testE2ESuite) SetupSuite() {
var err error
s.pool, err = dockertest.NewPool("")
```

```
s.Require().NoError(err)

s.Require().NoError(s.pool.Client.Ping())

s.ctx = context.Background()
}

func (s *testE2ESuite) TearDownSuite() {
for _, resource := range s.resources {
	s.NoError(s.pool.Purge(resource))
}
}

func (s *testE2ESuite) TestKafka() {
s.withKafka()

b := bridge.New(bridge.Config{
	Routes: []bridge.RouteConfig{
		{
			Condition: &bridge.EverythingCondition{},
			Target:    bridge.DirectDestination("kafka_exchange"),
		},
	},
	ExchangeConfigs:
map[bridge.ExchangeName]bridge.ExchangeDestination{
		"kafka_exchange": {
			Kind:   "kafka",
			Config:                          toJsonRawMessage(s.T(),
bridge.KafkaConfig{Topic: "test_exchange"}),
		},
```

```
        },
    })

    kafka, err := bridge.NewKafka([]string{"localhost:9092"})
    s.Require().NoError(err)
    b.RegisterConnector("kafka", kafka)

    msgCh := s.expectMessageOnce(b, "kafka_exchange")

    msg, err := bridge.NewJSONMessage("Hello, World!",
map[string]string{"foo": "bar"})
    s.Require().NoError(err)

    err = b.Publish(s.ctx, msg)
    s.Require().NoError(err)

    receivedMessage := <-msgCh
    s.Assert().Equal(msg.Metadata(), receivedMessage.Metadata())
    s.Assert().Equal(msg.Payload(), receivedMessage.Payload())
    }

    func (s *testE2ESuite) TestRabbitMQ() {
    s.withAMQP()

    b := bridge.New(bridge.Config{
        Routes: []bridge.RouteConfig{
            {
                Condition: &bridge.EverythingCondition{},
                Target:   bridge.DirectDestination("amqp_exchange"),
            },
```

```
		},
		ExchangeConfigs:
map[bridge.ExchangeName]bridge.ExchangeDestination{
			"amqp_exchange": {
				Kind:   "amqp",
				Config:                          toJsonRawMessage(s.T(),
bridge.AMQPConfig{Queue: "queue"}),
			},
		},
	})

	amqp, err := bridge.NewAMQP("amqp://guest:guest@localhost:5672/")
	s.Require().NoError(err)
	b.RegisterConnector("amqp", amqp)

	msgCh := s.expectMessageOnce(b, "amqp_exchange")

	msg,      err      :=      bridge.NewJSONMessage("Hello,      World!",
map[string]string{"foo": "bar"})
	s.Require().NoError(err)

	err = b.Publish(s.ctx, msg)
	s.Require().NoError(err)

	receivedMessage := <-msgCh
	s.Assert().Equal(msg.Metadata(), receivedMessage.Metadata())
	s.Assert().Equal(msg.Payload(), receivedMessage.Payload())
	}

	func (s *testE2ESuite) TestRouting() {
```

```
	s.withAMQP()
	s.withKafka()

	b := bridge.New(bridge.Config{
		Routes: []bridge.RouteConfig{
			{
				Condition: &bridge.Negate{
					Condition: &bridge.MetadataValueCondition{
						Key:   "route",
						Value: "kafka",
					},
				},
				Target: bridge.DirectDestination("amqp_exchange"),
			},
			{
				Condition: &bridge.MetadataValueCondition{
					Key:   "route",
					Value: "kafka",
				},
				Target: bridge.DirectDestination("kafka_exchange"),
			},
		},
		ExchangeConfigs:
map[bridge.ExchangeName]bridge.ExchangeDestination{
			"amqp_exchange": {
				Kind:   "amqp",
				Config:                     toJsonRawMessage(s.T(),
bridge.AMQPConfig{Queue: "queue"}),
			},
			"kafka_exchange": {
```

```
                    Kind:   "kafka",
                    Config:                        toJsonRawMessage(s.T(),
bridge.KafkaConfig{Topic: "topic"}),
                },
            },
        })

        amqp, err := bridge.NewAMQP("amqp://guest:guest@localhost:5672/")
        s.Require().NoError(err)
        b.RegisterConnector("amqp", amqp)

        kafka, err := bridge.NewKafka([]string{"localhost:9092"})
        s.Require().NoError(err)
        b.RegisterConnector("kafka", kafka)

        amqpMsgCh := s.expectMessageOnce(b, "amqp_exchange")
        kafkaMsgCh := s.expectMessageOnce(b, "kafka_exchange")

        msgKafka,     err     :=     bridge.NewJSONMessage("Hello,     World!",
map[string]string{"route": "kafka"})
        s.Require().NoError(err)

        msgAMQP,     err     :=     bridge.NewJSONMessage("Hello,     World!",
map[string]string{"route": "amqp"})
        s.Require().NoError(err)

        err = b.Publish(s.ctx, msgKafka)
        s.Require().NoError(err)

        err = b.Publish(s.ctx, msgAMQP)
```

```
s.Require().NoError(err)

amqpMsg := <-amqpMsgCh
s.Assert().Equal(msgAMQP.Metadata(), amqpMsg.Metadata())
s.Assert().Equal(msgAMQP.Payload(), amqpMsg.Payload())

kafkaMsg := <-kafkaMsgCh
s.Assert().Equal(msgKafka.Metadata(), kafkaMsg.Metadata())
s.Assert().Equal(msgKafka.Payload(), kafkaMsg.Payload())
}

func (s *testE2ESuite) withKafka() {
resource, err := s.pool.RunWithOptions(
	&dockertest.RunOptions{
		Repository: "bitnami/kafka",
		Tag:       "3.5.1-debian-11-r7",
		Env: []string{
			"ALLOW_PLAINTEXT_LISTENER=yes",

"KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE=true",
			// KRaft settings
			"KAFKA_CFG_NODE_ID=0",
			"KAFKA_CFG_PROCESS_ROLES=controller,broker",

"KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@:9093",
			// Listeners

"KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093",

"KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092
```

```
",

        "KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CONTROLLE
R:PLAINTEXT,PLAINTEXT:PLAINTEXT",

        "KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER",

        "KAFKA_CFG_INTER_BROKER_LISTENER_NAME=PLAINTEXT",
                },
                PortBindings: map[docker.Port][]docker.PortBinding{
                    "9092/tcp": {
                        {HostIP: "", HostPort: "9092"},
                    },
                },
            },
            func(config *docker.HostConfig) {
                config.AutoRemove = true
                config.RestartPolicy = docker.RestartPolicy{Name: "no"}
            },
        )
        s.Require().NoError(err)
        s.resources = append(s.resources, resource)
        time.Sleep(5 * time.Second)
        }

        func (s *testE2ESuite) withAMQP() {
        resource, err := s.pool.RunWithOptions(
            &dockertest.RunOptions{
                Name:       "rabbitmq",
                Repository: "rabbitmq",
```

```
				Tag:       "3.12.9-management",
				PortBindings: map[docker.Port][]docker.PortBinding{
					"5672/tcp": {
						{HostIP: "", HostPort: "5672"},
					},
				},
			},
			func(config *docker.HostConfig) {
				config.AutoRemove = true
				config.RestartPolicy = docker.RestartPolicy{Name: "no"}
			},
		)
		s.Require().NoError(err)
		s.resources = append(s.resources, resource)
		time.Sleep(5 * time.Second)

		conn, err := amqp091.Dial("amqp://guest:guest@localhost:5672/")
		s.Require().NoError(err)
		defer conn.Close()

		channel, err := conn.Channel()
		s.Require().NoError(err)

		_, err = channel.QueueDeclare("queue", true, false, false, false, nil)
		s.Require().NoError(err)
		}

		func (s *testE2ESuite) expectMessageOnce(b *bridge.Bridge, exchangeName
bridge.ExchangeName) chan bridge.Message {
		receivedMessageCh := make(chan bridge.Message, 1)
```

```
	subscribeErrorCh := make(chan error, 1)

	go func() {
		wg := sync.WaitGroup{}
		wg.Add(1)
		ctx, cancel := context.WithCancel(s.ctx)

		err := b.Subscribe(ctx, exchangeName, bridge.SubscriptionFunc(func(ctx
context.Context, msg bridge.Message) error {
			receivedMessageCh <- msg
			cancel()
			wg.Done()
			return nil
		}))
		subscribeErrorCh <- err
		wg.Wait()
	}()

	select {
	case err := <-subscribeErrorCh:
		s.Require().NoError(err)
	default:
	}

	return receivedMessageCh
	}

	func toJsonRawMessage(t *testing.T, v any) json.RawMessage {
	bytes, err := json.Marshal(v)
	require.NoError(t, err)
```

```
return bytes
}
package bridge

import (
"context"
"github.com/rabbitmq/amqp091-go"
)

type amqpExchange struct {
channel *amqp091.Channel
}

type AMQPConfig struct {
Queue string `json:"queue"`
}

func NewAMQP(url string) (Connector, error) {
conn, err := amqp091.Dial(url)
if err != nil {
	return nil, err
}

channel, err := conn.Channel()
if err != nil {
	return nil, err
}
return &amqpExchange{
	channel: channel,
}, nil
```

```go
}

func (a *amqpExchange) Connect(ctx context.Context, cp ConfigProvider) (Exchange, error) {
	cfg := AMQPConfig{}
	if err := cp.Configure(&cfg); err != nil {
		return nil, err
	}

	return &amqpHandler{
		channel: a.channel,
		cfg:     cfg,
	}, nil
}

type amqpHandler struct {
	channel *amqp091.Channel
	cfg     AMQPConfig
}

func (a *amqpHandler) Publish(ctx context.Context, message Message) error {

	headers := map[string]any{}
	for key, value := range message.Metadata() {
		headers[key] = value
	}

	return a.channel.PublishWithContext(
		ctx,
		"",
```

```
			a.cfg.Queue,
			true,
			false,
			amqp091.Publishing{
				DeliveryMode: amqp091.Persistent,
				Headers:      headers,
				Body:         message.Payload(),
			},
		)
	}

	func (a *amqpHandler) Subscribe(ctx context.Context, subscription Subscription) error {

		if err := a.channel.Qos(1, 0, false); err != nil {
			return err
		}

		consumeCh, err := a.channel.ConsumeWithContext(ctx, a.cfg.Queue, "", false, false, false, false, nil)
		if err != nil {
			return err
		}

		for delivery := range consumeCh {
			md := Metadata{}
			for key, value := range delivery.Headers {
				md[key] = value.(string)
			}
			rm := NewReceivedMessage(md, delivery.Body)
```

```go
		err = subscription.OnMessage(ctx, rm)
		if err != nil {
			return err
		}
		err = delivery.Ack(false)
		if err != nil {
			return err
		}
	}

	return nil
	}
	package bridge

	import (
	"context"
	"errors"
	"fmt"
	)

	var (
	ErrUnknownExchangeKind          = errors.New("unknown exchange kind")
	ErrExchangeCreation           = errors.New("failed to create exchange")
	ErrUnknownExchangeConfiguration = errors.New("unknown exchange
configuration")
	)

	type (
	ExchangeKind string
```

```
ConfigProvider interface {
	Configure(value any) error
}

Connector interface {
	Connect(context.Context, ConfigProvider) (Exchange, error)
}

FactoryConfig struct {
	ExchangeConfigs map[ExchangeName]ExchangeDestination
}

Exchange interface {
	Publish(ctx context.Context, message Message) error
	Subscribe(ctx context.Context, subscription Subscription) error
}
)

type exchangeFactory struct {
initiators     map[ExchangeKind]Connector
exchangeConfigs map[ExchangeName]ExchangeDestination
}

func (q *exchangeFactory) CreateExchange(ctx context.Context, name ExchangeName) (qh Exchange, err error) {
dest, exists := q.exchangeConfigs[name]
if !exists {
	return nil, fmt.Errorf("%w, exchange name %s", ErrUnknownExchangeConfiguration, name)
```

```
	}

	conn, exists := q.initiators[dest.Kind]
	if !exists {
		return nil, fmt.Errorf("%w, exchange kind %s", ErrUnknownExchangeKind, dest.Kind)
	}

	qh, err = conn.Connect(ctx, &dest)
	if err != nil {
		return nil, fmt.Errorf("%w: %s", ErrExchangeCreation, err.Error())
	}

	return qh, nil
	}

	func (q *exchangeFactory) RegisterConnector(destinationType ExchangeKind, connector Connector) {
	q.initiators[destinationType] = connector
	}

	func newExchangeFactory(cfg FactoryConfig) *exchangeFactory {
	return &exchangeFactory{
		initiators:      make(map[ExchangeKind]Connector),
		exchangeConfigs: cfg.ExchangeConfigs,
	}
	}
	package bridge

	import (
```

```
"context"
"github.com/IBM/sarama"
)

type kafka struct {
producer sarama.SyncProducer
consumer sarama.Consumer
}

type KafkaConfig struct {
Topic string `json:"topic"`
}

func NewKafka(brokers []string) (Connector, error) {
cfg := sarama.NewConfig()
cfg.Version = sarama.MaxVersion
cfg.Producer.Return.Successes = true
syncProducer, err := sarama.NewSyncProducer(brokers, cfg)
if err != nil {
	return nil, err
}
consumer, err := sarama.NewConsumer(brokers, cfg)
if err != nil {
	return nil, err
}
return kafka{producer: syncProducer, consumer: consumer}, nil
}

func (c kafka) Connect(ctx context.Context, cp ConfigProvider) (Exchange, error) {
```

```
	cfg := KafkaConfig{}
	if err := cp.Configure(&cfg); err != nil {
		return nil, err
	}

	return &kafkaQueueHandler{
		producer: c.producer,
		consumer: c.consumer,
		topic:    cfg.Topic,
	}, nil
	}

	type kafkaQueueHandler struct {
	topic    string
	consumer sarama.Consumer
	producer sarama.SyncProducer
	}

	func (k kafkaQueueHandler) Publish(ctx context.Context, message Message)
error {
	pm := &sarama.ProducerMessage{
		Topic:   k.topic,
		Key:     nil,
		Value:   encode(message),
		Headers: k.buildHeaders(message),
	}

	_, _, err := k.producer.SendMessage(pm)
	return err
	}
```

```go
func (k kafkaQueueHandler) Subscribe(ctx context.Context, subscription Subscription) error {
	pc, err := k.consumer.ConsumePartition(k.topic, 0, sarama.OffsetNewest)
	if err != nil {
		return err
	}

	for {
		select {
		case <-ctx.Done():
			return ctx.Err()
		case msg := <-pc.Messages():
			md := k.getMetadata(msg.Headers)
			rm := NewReceivedMessage(md, msg.Value)

			err = subscription.OnMessage(ctx, rm)
			if err != nil {
				return err
			}
		}
	}
	}

func (k kafkaQueueHandler) buildHeaders(message Message) []sarama.RecordHeader {
	records := make([]sarama.RecordHeader, 0, len(message.Metadata())+1)
	for key, value := range message.Metadata() {
		records = append(records, sarama.RecordHeader{
			Key:   []byte(key),
```

```
				Value: []byte(value),
			})
		}

		return records
	}

	func (k kafkaQueueHandler) getMetadata(headers []*sarama.RecordHeader) Metadata {
		md := map[string]string{}
		for _, rec := range headers {
			key := string(rec.Key)
			value := string(rec.Value)
			md[key] = value
		}

		return md
	}

	func encode(m Message) sarama.Encoder {
		return sarama.ByteEncoder(m.Payload())
	}
	package bridge

	import (
		"encoding/json"
	)

	type (
		Message interface {
```

```
		Metadata() Metadata
		Payload() []byte
	}
	Metadata map[string]string
)

func NewReceivedMessage(md Metadata, payload []byte) Message {
	return &receivedMessage{
		metadata: md,
		payload:  payload,
	}
}

type receivedMessage struct {
	payload  []byte
	metadata Metadata
}

func (m *receivedMessage) Metadata() Metadata {
	md := make(Metadata, len(m.metadata))
	for key, value := range m.metadata {
		md[key] = value
	}
	if len(md) == 0 {
		return nil
	}
	return md
}

func (m *receivedMessage) Payload() []byte {
```

```
return m.payload
}

type jsonMessage struct {
md      Metadata
payload []byte
}

func (n *jsonMessage) Metadata() Metadata {
return n.md
}

func (n *jsonMessage) Payload() []byte {
return n.payload
}

func NewJSONMessage(payload any, md Metadata) (Message, error) {
data, err := json.Marshal(payload)
if err != nil {
        return nil, err
}
return &jsonMessage{
        md:     md,
        payload: data,
}, nil
}
package bridge

import (
"encoding/json"
```

```
"fmt"
"math/rand"
"sort"
"strings"
)

type Destination interface {
Exchange() ExchangeName
String() string
}

type directDestination ExchangeName

func (d directDestination) Exchange() ExchangeName {
return ExchangeName(d)
}

func (d directDestination) String() string {
return string(d)
}

func (d *directDestination) UnmarshalJSON(data []byte) error {
str := string(data)
*d = directDestination(strings.Trim(str, `"`))
return nil
}

func DirectDestination(name ExchangeName) Destination {
return directDestination(name)
}
```

```
type weightedDestination struct {
targets []ExchangeWeight
}

func (w weightedDestination) Exchange() ExchangeName {
sort.Slice(w.targets, func(i, j int) bool {
	return w.targets[i].Weight > w.targets[j].Weight
})

random := rand.Float64()
acc := .0
for _, target := range w.targets {
	acc += target.Weight
	if random < acc {
		return target.Exchange
	}
}
return w.targets[len(w.targets)-1].Exchange
}

func (w weightedDestination) String() string {
b := strings.Builder{}
for _, target := range w.targets {
	b.WriteString(fmt.Sprintf("%s:%.3f;", target.Exchange, target.Weight))
}
return b.String()
}

func (w *weightedDestination) UnmarshalJSON(data []byte) error {
```

```
if err := json.Unmarshal(data, &w.targets); err != nil {
        return err
}
return nil
}

type ExchangeWeight struct {
Exchange ExchangeName
Weight   float64
}

func WeightedDestination(targets ...ExchangeWeight) Destination {
return &weightedDestination{
        targets: targets,
}
}
package bridge

import (
"encoding/json"
"fmt"
)

type RouteConfig struct {
Condition Condition   `json:"Condition"`
Target    Destination `json:"target"`
}

type RouterConfig struct {
Routes []RouteConfig
```

```
}

type destinationConfig struct {
Kind   string
Config json.RawMessage
}

func (r *RouteConfig) UnmarshalJSON(bytes []byte) error {
var err error

data := map[string]json.RawMessage{}
err = json.Unmarshal(bytes, &data)
if err != nil {
        return err
}

if r.Condition, err = buildCondition(data); err != nil {
        return err
}

if r.Target, err = buildDestination(data); err != nil {
        return err
}

return nil
}

func buildCondition(data map[string]json.RawMessage) (Condition, error) {
var exists bool
var val json.RawMessage
```

```
	val, exists = data["Condition"]
	if !exists {
		return EverythingCondition{}, nil
	}

	var attrs map[string]json.RawMessage
	err := json.Unmarshal(val, &attrs)
	if err != nil {
		return nil, fmt.Errorf("unknown attribute format: %w", err)
	}

	return buildAttributesConditionRec(attrs, nil)
	}

	func buildAttributesConditionRec(attrs map[string]json.RawMessage,
aggregation func(...Condition) Condition) (Condition, error) {
	ms := make([]Condition, 0, len(attrs))

	for key, jsonRaw := range attrs {
		var m Condition
		var err error

		switch key {
		case "@equals":
			m = &MetadataValueCondition{}
			err = json.Unmarshal(jsonRaw, m)
		case "@not":
			m, err = buildComplex(jsonRaw, func(conditions ...Condition)
Condition {
				return &Negate{Condition: conditions[0]}
```

```
			})
		case "@all":
			m, err = buildComplex(jsonRaw, func(conditions ...Condition)
Condition {
				return &All{conditions: conditions}
			})
		default:
			err = fmt.Errorf("unknown Condition %q", key)
		}

		if err != nil {
			return nil, err
		}

		ms = append(ms, m)
	}

	if aggregation != nil {
		return aggregation(ms...), nil
	} else if len(ms) == 1 {
		return ms[0], nil
	} else {
		return All{conditions: ms}, nil
	}
	}

	func buildComplex(data json.RawMessage, aggregator func(...Condition)
Condition) (Condition, error) {
	var attrsArray []json.RawMessage
	err := json.Unmarshal(data, &attrsArray)
```

```
	if err != nil {
		return nil, fmt.Errorf("unknown format, array expected, got: %T; error: %w", data, err)
	}
	ms := make([]Condition, 0, len(attrsArray))
	for _, attrJsonRaw := range attrsArray {

		var attrs map[string]json.RawMessage
		err := json.Unmarshal(attrJsonRaw, &attrs)
		if err != nil {
			return nil, fmt.Errorf("unknown attribute format: %w", err)
		}

		m, err := buildAttributesConditionRec(attrs, nil)
		if err != nil {
			return nil, fmt.Errorf("error building conditions")
		}

		ms = append(ms, m)
	}
	return aggregator(ms...), nil
	}

	func buildDestination(data map[string]json.RawMessage) (Destination, error) {
	var exists bool
	var destRaw json.RawMessage
	destRaw, exists = data["target"]
	if !exists {
		return nil, fmt.Errorf("no destination")
	}
```

```
var destConfig destinationConfig
err := json.Unmarshal(destRaw, &destConfig)
if err != nil {
	return nil, fmt.Errorf("failed to unmarshal destination: %w", err)
}

var target Destination
switch destConfig.Kind {
case "@direct":
	var t directDestination
	err = json.Unmarshal(destConfig.Config, &t)
	target = t
case "@weight":
	var t weightedDestination
	err = json.Unmarshal(destConfig.Config, &t)
	target = t
default:
	err = fmt.Errorf("unknown destination type")
}

if err != nil {
	return nil, err
}

return target, nil
}
package bridge

import (
```

```
"context"
"errors"
)

var ErrNoMatchingTarget = errors.New("no matching target")

type routeResolver struct {
routes []RouteConfig
}

func (r *routeResolver) FindMatching(ctx context.Context, message Message)
(target Destination, err error) {
var match bool

for _, route := range r.routes {
	if match, err = route.Condition.Satisfy(message); err != nil {
		return target, err
	} else if !match {
		continue
	}
	target = route.Target
	return target, nil
}

return target, ErrNoMatchingTarget
}

func newRouter(cfg RouterConfig) *routeResolver {
r := &routeResolver{
	routes: cfg.Routes,
```

```
}
for i := range r.routes {
	if r.routes[i].Condition == nil {
		r.routes[i].Condition = EverythingCondition{}
	}
}
return r
}
```

ДОДАТОК Ж

Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень.

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Інформаційна технологія асинхронної взаємодії у розподіленій системі на основі мікросервісної архітектури

Тип роботи: магістерська кваліфікаційна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 99% Схожість 1%

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку ________ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи ________ Грядченко А. О.
(підпис) (прізвище, ініціали)

Керівник роботи ________ Кадук О. В.
(підпис) (прізвище, ініціали)