Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи взаємодії модулів платформи віддаленого виклику процедур gRPC

(тема)

Виконав:

студент II курсу, групи СПм-23-2

Хорошилов В.Р.
(прізвище, ініціали)

Спеціальність

123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: проф. Волк М.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Хорошилову Валентину Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи взаємодії модулів платформи віддаленого виклику процедур gRPC

затверджена наказом по університету від " 22 " листопада 2024 р. № 1236 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20 січня 2025 р.

3. Вхідні дані до роботи 1) Мікросервісна архітектура
2) Основи та протоколи, що використовує фреймворк gRPC (HTTP/2, Protobuf).
3) Інструментальні засоби та мови програмування (Golang) для реалізації gRPC-сервісів.
4) Брокери повідомлень (Kafka) та підходи до асинхронної обробки даних.
5) Методи збору та аналізу метрик продуктивності.

4. Перелік питань, що потрібно опрацювати у роботі 1) Актуальність проблеми
2) Огляд і порівняння методів взаємодії у gRPC.
3) Проєктування та реалізація gRPC-сервісів на Golang.
4) Інтеграція з Kafka
5) Збір та аналіз метрик
6) Підсумкові висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 13 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд основних архітектурних підходів і	26.11.24 – 30.11.24	
	протоколів для організації взаємодії в gRPC		
2	Вибір та обґрунтування методики дослідження	02.12.24 – 05.12.24	
3	Вибір інструментальних засобів	06.12.24 – 10.12.24	
4	Розробка моделей методів взаємодії	11.12.24 – 21.12.24	
5	Проведення експериментів, вимірювання та	23.12.24 – 03.01.25	
	аналіз продуктивності		
6	Оформлення матеріалів кваліфікаційної роботи	04.01.25 – 07.01.25	
7	Подання кваліфікаційної роботи керівникові та	08.01.25 – 11.01.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензування	13.01.25 – 17.01.25	

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Волк М.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 63 с., 7 рис., 5 табл., 2 дод., 20 джерел.

КОМП’ЮТЕРНА МЕРЕЖА, ІНТЕРНЕТ, МАРШРУТИЗАТОР, ПРОТОКОЛ, СЕРВЕР, ШЛЮЗ, FIREWALL, WI-FI, WLAN.

Метою кваліфікаційної роботи є підвищення ефективності програмних систем шляхом оптимізації методів взаємодії модулів у платформі віддаленого виклику процедур gRPC. У сучасних розподілених системах, що базуються на мікросервісній архітектурі, швидка та надійна комунікація між сервісами є визначальним чинником продуктивності та масштабованості. В роботі проаналізовано переваги та обмеження синхронних і асинхронних викликів процедур, моделі запит-відповідь, а також інтеграції з брокерами повідомлень. Встановлено, що асинхронні виклики та стрімінгові RPC покращують масштабованість і скорочують затримки у високонавантажених системах. Водночас, метод публікація-підписка й інтеграція з брокерами повідомлень підвищують стійкість до збоїв і підтримують асинхронну взаємодію, хоча й потребують додаткових налаштувань.

У ході виконання кваліфікаційної роботи розроблено експериментальну платформу для оцінки різних методів взаємодії в gRPC. На основі зібраних метрик (час відповіді, пропускна здатність, використання ресурсів) сформульовано рекомендації щодо вибору відповідного методу залежно від рівня навантаження та складності бізнес-логіки. Запропоновані підходи можуть бути використані розробниками при проєктуванні мікросервісних систем, де важливо зменшити експлуатаційні витрати, підвищити швидкість розробки та розгортання, а також забезпечити ефективне використання комп’ютерних ресурсів.

ABSTRACT

Master’s thesis: 63 pages, 7 figures, 5 tables, 2 appendices, 20 sources.

FIREWALL, GATE, INTERNET, PROTOCOL, ROUTER, SERVER, WI-FI, WIRELESS NETWORK, WLAN.

The major goal of this thesis is to increase the efficiency of software systems by optimizing the methods of module interaction within the gRPC remote procedure call platform. In modern distributed systems that rely on microservice architectures, fast and reliable communication between services is a key factor for both performance and scalability. This study analyzes the advantages and limitations of synchronous and asynchronous procedure calls, the request-response model, and the integration with message brokers. It has been found that asynchronous calls and streaming RPCs improve scalability and reduce latency in high-load scenarios, while the publish-subscribe model and the use of message brokers enhance fault tolerance and support asynchronous interaction, albeit requiring additional configuration.

In order to evaluate the impact of various gRPC interaction methods, an experimental platform was developed during this qualification project. By collecting metrics such as response time, throughput, and resource usage, the study formulated recommendations on choosing the most suitable method, depending on the load level and business logic complexity. These proposed approaches can be employed by developers to lower operational costs, speed up development and deployment, and ensure more efficient utilization of computing resources when designing microservice systems.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 10
1 ТЕОРЕТИЧНІ ОСНОВИ 12
1.1 Огляд мікросервісної архітектури 12
1.2 Основи технології gRPC 13
1.3 Порівняння gRPC з іншими технологіями 15
1.4 Методи взаємодії в gRPC 17
1.4.1 Синхронні та асинхронні виклики процедур 17
1.4.2 Модель запит-відповідь 18
1.4.3 Модель публікація-підписка 19
1.4.4 Використання шини та брокерів повідомлень 20
1.5 Golang 21
1.6 Kafka 22
2 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ 24
2.1 Архітектурний підхід 24
2.2 Структура проекту 24
2.3 Реалізація сервісів за допомогою gRPC 25
2.4 Серверна частина 28
2.5 Клієнтська частина 32
2.5.1 Збір метрик для аналізу 36
2.6 Автоматизація експериментів 38
3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ 41
3.1 Середа проведення дослідження 41
3.2 Результати дослідження 42
ВИСНОВКИ 51
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 53

ДОДАТОК А Графічний матеріал кваліфікаційної роботи 55
ДОДАТОК Б Довідка про публікацію 63

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Брокери повідомлень – системи для маршрутизації, зберігання та передачі повідомлень між компонентами.

Двонаправлений стрімінг – спосіб передачі даних, коли обмін відбувається одночасно в обох напрямках у форматі потоку.

Канал – логічне з’єднання між клієнтом і сервером для обміну повідомленнями.

Мікросервісна архітектура – стиль розробки програмного забезпечення, що базується на незалежних мікросервісах.

Стрім – потік даних у режимі реального часу.

Шини – інфраструктура для обміну повідомленнями між компонентами системи.

API – інтерфейс програмування додатків (англ., Application Programming Interface).

CI/CD – безперервна інтеграція та безперервне розгортання (англ., Continuous Integration/Continuous Deployment).

gRPC – фреймворк для віддалених викликів процедур (англ., Remote Procedure Call framework).

HTTP – протокол передачі гіпертексту (англ., HyperText Transfer Protocol).

JSON – текстовий формат обміну даними (англ., JavaScript Object Notation).

Kafka – розподілена платформа для роботи з потоками даних і повідомленнями (брокер повідомлень).

Protobuf – мова серіалізації структурованих даних (англ., Protocol Buffers).

REST – архітектурний стиль побудови розподілених систем (англ.,

Representational State Transfer).

RESTful API – реалізація API, що відповідає принципам REST.

RPC – механізм викликів віддалених процедур (англ., Remote Procedure Call).

XML – мова розмітки, що використовується для структурування, зберігання та передачі даних (англ., Extensible Markup Language).

ВСТУП

У сучасному світі інформаційних технологій відбувається стрімке зростання складності програмних систем, що обумовлює підвищені вимоги до їх продуктивності, масштабованості та надійності. У розподіленому середовищі, де більшість застосунків ґрунтуються на мікросервісній архітектурі, особливої уваги набуває проблема ефективної комунікації між численними компонентами системи. Хоча такі технології, як RESTful API[1], широко застосовуються на практиці, вони не завжди дають змогу досягти необхідних показників швидкодії та мають певні обмеження при обробці високих навантажень або забезпеченні двонаправленої передачі даних у реальному часі. За кордоном та в Україні це питання залишається актуальним, адже розробники й дослідники постійно шукають способи зменшити затримки та оптимізувати використання комп’ютерних ресурсів, не ускладнюючи при цьому архітектуру системи.

Світові тенденції свідчать про активне впровадження сучасних протоколів і технологій, здатних забезпечувати високу пропускну здатність та низькі затримки. Одним із провідних рішень вважають платформу gRPC[2], створену компанією Google, яка базується на протоколі HTTP/2 та використовує бінарний формат серіалізації Protocol Buffers. Завдяки цьому, gRPC забезпечує можливість двонаправленого стрімінгу, асинхронної обробки та ефективного управління ресурсами в умовах стрімкого зростання навантаження[3]. Утім, незважаючи на розв’язання низки завдань, пов’язаних із продуктивністю та масштабованістю, у цій сфері все ще існують прогалини щодо того, як саме застосовувати різні методи взаємодії модулів у конкретних сценаріях, щоб досягти оптимальних результатів[4].

Актуальність даної роботи зумовлена необхідністю детального аналізу та практичної перевірки різних методів взаємодії в gRPC, включаючи синхронні та асинхронні виклики, модель запит-відповідь, публікацію-

підписку, а також інтеграцію з брокерами повідомлень. Розуміння того, які методи найкраще підходять за певних умов, дозволить зменшити експлуатаційні витрати, прискорити обробку запитів і зробити програмні системи більш гнучкими та надійними[5]. Підстави для проведення цього дослідження лежать у потребі розробників та компаній, що працюють із мікросервісними системами, у виборі оптимальних технологій комунікації, здатних задовольнити вимоги високої пропускної здатності та мінімальних затримок.

Метою кваліфікаційної роботи є виявлення та дослідження кращих практик застосування методів взаємодії модулів у платформі віддаленого виклику процедур gRPC, а також розробка рекомендацій, які дозволять підвищити ефективність розподілених систем і знайти збалансований підхід до реалізації високонавантажених мікросервісних проєктів.

Сфера застосування результатів охоплює як корпоративні рішення з великим обсягом даних, так і стартапи, що прагнуть створювати конкурентоспроможні та масштабовані продукти[6].

Таким чином, дослідження робить свій внесок у розвиток практики та теорії побудови розподілених систем, пропонуючи методи й інструменти для вдосконалення архітектури програмного забезпечення на базі технології gRPC.

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1 Огляд мікросервісної архітектури

Мікросервісна архітектура є сучасним підходом до розробки програмного забезпечення, який базується на принципі розподілу системи на набір незалежних, дрібних сервісів. Кожен мікросервіс відповідає за виконання конкретної бізнес-логіки і може розгортатися, масштабуватися та оновлюватися незалежно від інших. Це дозволяє розробникам створювати більш гнучкі та стійкі до змін системи, які легше адаптуються до вимог ринку та технологічних інновацій[7].

Однією з головних переваг мікросервісної архітектури є можливість розподіленої розробки. Різні команди можуть працювати над окремими сервісами, використовуючи різні технології та мови програмування, що підвищує продуктивність та сприяє інноваціям. Крім того, мікросервіси полегшують процес безперервної інтеграції та доставки (CI/CD), що дозволяє швидше виводити нові функціональності на ринок.

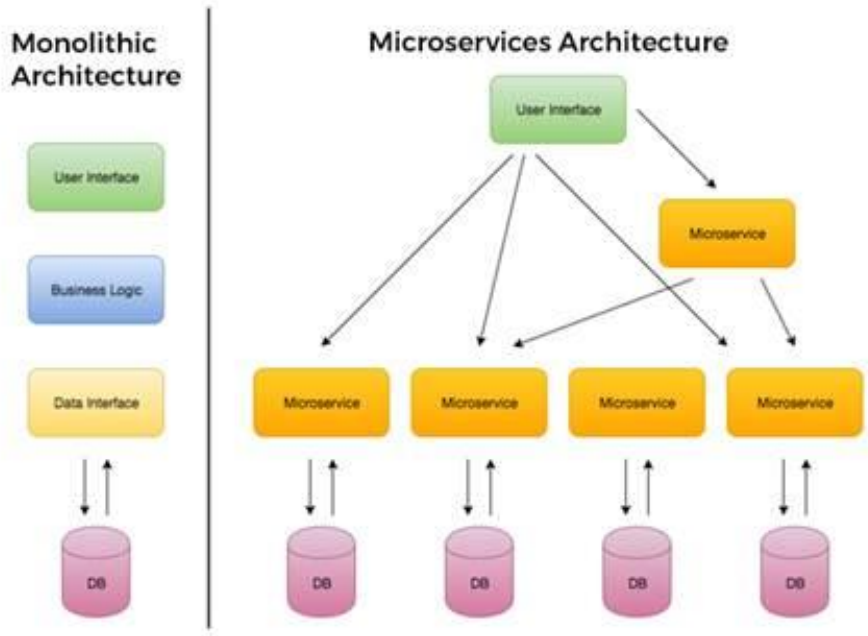

Рисунок 1.1 – Порівняння монолітної та мікросервісної архітектур

Масштабованість є ще однією важливою перевагою. У традиційних монолітних системах масштабування вимагало збільшення ресурсів для всієї системи, навіть якщо навантаження зростало лише на певному її компоненті. У мікросервісній архітектурі можна масштабувати лише ті сервіси, які відчувають підвищене навантаження, що оптимізує використання ресурсів та знижує витрати. На рисунку 1.1 показано порівняння монолітної та мікросервісної архітектур, що допомагає краще зрозуміти різницю між цими підходами.

Проте мікросервісна архітектура має і свої виклики. Розподіленість системи ускладнює її управління, моніторинг та забезпечення безпеки. Зростає складність взаємодії між сервісами, оскільки необхідно забезпечити надійну та ефективну комунікацію в умовах можливих мережевих збоїв. Також виникають питання щодо обробки транзакцій та узгодженості даних, особливо в системах, де необхідно підтримувати високу цілісність інформації[8].

Для успішного впровадження мікросервісної архітектури необхідно використовувати відповідні технології та інструменти. Серед них – це системи управління контейнерами, такі як Kubernetes, які полегшують розгортання та управління сервісами. Також важливу роль відіграють засоби моніторингу та логування, які забезпечують прозорість роботи системи та швидке виявлення проблем.

1.2 Основи технології gRPC

gRPC (Remote Procedure Call) — це сучасна платформа віддаленого виклику процедур, розроблена компанією Google. Вона дозволяє ефективно будувати розподілені системи та мікросервіси, забезпечуючи швидку та надійну комунікацію між ними. gRPC базується на протоколі HTTP/2, що надає переваги у вигляді двонаправленого стрімінгу, мультиплексування запитів та покращеної компресії даних[9].

Однією з ключових особливостей gRPC є використання Protocol Buffers (protobuf) — мови опису інтерфейсів та формату серіалізації даних, який є більш ефективним та компактним порівняно з традиційними форматами, такими як JSON або XML. Це дозволяє зменшити розмір переданих повідомлень та підвищити швидкодію системи.

У gRPC сервіси та їх методи визначаються у файлах з розширенням «.proto», де описуються повідомлення та сервіси.

Лістинг 1.1 – Базова структура .proto-файлу

```
syntax = "proto3";
service MyService {
  rpc MyMethod (MyRequest) returns (MyResponse);
}
message MyRequest {
  string data = 1;
}
message MyResponse {
  string result = 1;
}
```

Після визначення інтерфейсу його можна скомпілювати у код для різних мов програмування, що забезпечує сумісність та спрощує розробку багатомовних систем.

gRPC підтримує кілька типів взаємодії між клієнтом та сервером:

- одноразовий виклик (Unary RPC) – клієнт надсилає один запит та отримує один відповідь;
- серверний стрімінг (Server Streaming RPC) – клієнт надсилає один запит, а сервер повертає потік відповідей;
- клієнтський стрімінг (Client Streaming RPC) – клієнт надсилає потік запитів, а сервер повертає один відповідь;
- двонаправлений стрімінг (Bidirectional Streaming RPC) – клієнт і сервер обмінюються потоками повідомлень одночасно.

Використання HTTP/2 дозволяє ефективно керувати цими потоками, забезпечуючи низькі затримки та високу пропускну здатність.

Безпека у gRPC забезпечується за допомогою TLS/SSL, що дозволяє шифрувати передані дані та аутентифікувати сторони з'єднання. Також можлива інтеграція з іншими механізмами аутентифікації, такими як OAuth2[10].

1.3 Порівняння gRPC з іншими технологіями

У процесі вибору технології для взаємодії між модулями важливо розуміти переваги та недоліки різних підходів. Найпоширенішою альтернативою gRPC є RESTful API, яка використовує протокол HTTP/1.1 та формат даних JSON. На рисунку 1.2 показані відмінності у архітектурі запитів для цих архітектур.

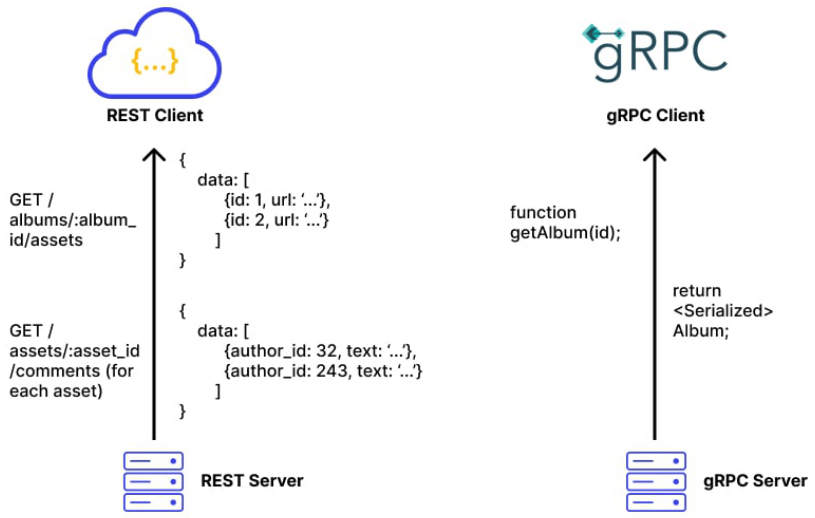

Рисунок 1.2 – Порівняння запитів між клієнт-серверним додатком у REST та gRPC відповідно

Переваги gRPC над RESTful API:

- продуктивність: завдяки використанню Protocol Buffers та HTTP/2, gRPC забезпечує менші затримки та вищу пропускну здатність, що особливо

важливо в системах з високими вимогами до швидкодії;

- бінарний формат даних: Protocol Buffers використовують бінарний формат, який є більш компактним та швидшим у серіалізації/десеріалізації порівняно з текстовими форматами, такими як JSON;

- підтримка стрімінгу: gRPC нативно підтримує різні типи стрімінгу, що спрощує реалізацію реального часу взаємодії та обробки потокових даних;

- автоматична генерація коду: визначення сервісів у «.proto» файлах дозволяє автоматично генерувати клієнтський та серверний код для різних мов програмування, забезпечуючи сумісність та знижуючи ризик помилок.

Недоліки gRPC порівняно з RESTful API:

- сумісність з веб-браузерами: оскільки gRPC використовує HTTP/2 та бінарний формат даних, безпосереднє використання з веб-браузерами може бути ускладненим. Для вирішення цієї проблеми існує gRPC-Web, але це додає додатковий рівень складності;

- читабельність даних: бінарний формат Protocol Buffers менш читабельний для людини порівняно з JSON, що може ускладнити налагодження без відповідних інструментів;

- навчання та впровадження: командам, незнайомим з gRPC, може знадобитися час на вивчення нової технології та інструментів.

Порівняння з іншими технологіями:

- SOAP: це протокол обміну повідомленнями, який використовує XML для форматування даних[11]. Порівняно з SOAP, gRPC є більш легковажним та продуктивним рішенням, хоча SOAP може бути необхідним у середовищах, де потрібна розширена функціональність або сумісність зі старими системами;

- Thrift: розроблений компанією Facebook, Thrift також використовує бінарний формат та підтримує мультиплексування мов програмування. Проте

gRPC має перевагу у вигляді використання HTTP/2 та ширшої підтримки[4].

У підсумку, вибір між gRPC та іншими технологіями залежить від конкретних вимог проекту. Якщо важлива висока продуктивність, ефективність передачі даних та підтримка стрімінгу, gRPC є відмінним вибором. Якщо ж необхідна широка сумісність з веб-клієнтами або простота у використанні, може бути доцільним залишитися на RESTful API.

1.4 Методи взаємодії в gRPC

У платформі віддаленого виклику процедур gRPC існує кілька методів взаємодії між клієнтом та сервером, які дозволяють розробникам гнучко будувати розподілені системи з урахуванням специфіки їхніх вимог.

Ці методи включають синхронні та асинхронні виклики процедур, модель запит-відповідь, модель публікація-підписка, а також використання шини та брокерів повідомлень. Кожен з них має свої особливості, переваги та обмеження, що впливають на вибір оптимального підходу в конкретному сценарії[12].

1.4.1 Синхронні та асинхронні виклики процедур

Синхронні виклики процедур у gRPC передбачають, що клієнт надсилає запит до сервера і блокується до отримання відповіді. Це означає, що потік виконання на стороні клієнта зупиняється, очікуючи результату від сервера. Такий підхід є простим у реалізації та використанні, оскільки послідовність взаємодії є лінійною та передбачуваною. Він підходить для операцій, які виконуються швидко і не потребують тривалого часу обробки.

Однак, у випадках, коли серверна операція може зайняти значний час або коли необхідно обробляти велику кількість запитів одночасно, синхронні виклики можуть стати вузьким місцем системи. Блокування клієнтського

потоку призводить до неефективного використання ресурсів та може знижувати загальну продуктивність.

Асинхронні виклики процедур дозволяють клієнту надсилати запит і продовжувати виконання без очікування негайної відповіді від сервера. Відповідь обробляється пізніше, коли вона стає доступною, за допомогою зворотних викликів, обіцянок (promises) або майбутніх результатів (futures). Цей підхід підвищує ефективність використання ресурсів, оскільки клієнт може обробляти інші завдання під час очікування відповіді.

У gRPC асинхронні виклики реалізуються через неблокуючі операції вводу-виводу та підтримку багатопоточності. Це особливо корисно в системах з високими вимогами до масштабованості та продуктивності, де необхідно обробляти велику кількість одночасних з'єднань та запитів[5].

Вибір між синхронними та асинхронними викликами залежить від конкретних вимог до системи. Синхронні виклики можуть бути прийнятними для простих додатків з низьким навантаженням, тоді як асинхронні підходять для більш складних та високонавантажених систем.

1.4.2 Модель запит-відповідь

Модель запит-відповідь є однією з найбільш поширених парадигм взаємодії в розподілених системах. У цій моделі клієнт надсилає запит до сервера, який обробляє його та повертає відповідь. Це забезпечує чітку та передбачувану комунікацію, де кожен запит має відповідний йому відповідь.

У gRPC модель запит-відповідь реалізується через одноразові виклики (Unary RPC). Клієнт та сервер обмінюються одним повідомленням кожен, що робить цей підхід простим та ефективним для більшості операцій. Ця модель підходить для випадків, коли необхідно виконати окрему операцію, наприклад, отримати інформацію про користувача або оновити запис у базі даних[5].

Перевагами моделі запит-відповідь є її простота та низька накладна

вартість. Відсутність необхідності підтримувати тривалі з'єднання або обробляти потоки даних знижує складність реалізації та покращує продуктивність. Однак, ця модель може бути обмеженою в сценаріях, де необхідна передача великого обсягу даних або реактивна взаємодія в режимі реального часу.

1.4.3 Модель публікація-підписка

Модель публікація-підписка (Publish-Subscribe) є патерном взаємодії, який дозволяє відправникам (публікаторам) передавати повідомлення безпосередньо невідомим отримувачам (підписникам). Публікатори надсилають повідомлення на певні теми або канали, а підписники отримують повідомлення, на які вони підписані. Це розв'язує відправників та отримувачів, дозволяючи їм працювати незалежно один від одного[6].

У gRPC модель публікація-підписка може бути реалізована за допомогою стрімінгових RPC. Зокрема, серверний стрімінг (Server Streaming RPC) дозволяє серверу надсилати потік повідомлень у відповідь на одиничний запит клієнта. Двонаправлений стрімінг (Bidirectional Streaming RPC) дозволяє обом сторонам обмінюватися потоками повідомлень одночасно.

Цей підхід підходить для додатків, де необхідна передача даних у режимі реального часу або обробка подій. Наприклад, у чатах, системах моніторингу, оновленнях статусу або потокових даних від датчиків. Використання стрімінгу зменшує накладні витрати на встановлення та закриття з'єднань для кожного повідомлення, підвищуючи ефективність системи.

Проте, модель публікація-підписка вимагає більш складної реалізації та обробки. Необхідно враховувати управління підписками, маршрутизацію повідомлень та обробку можливих збоїв у з'єднаннях. Також важливо забезпечити безпеку та автентифікацію при обміні повідомленнями.

1.4.4 Використання шини та брокерів повідомлень

Шина повідомлень та брокери повідомлень є посередницькими компонентами, які забезпечують асинхронну та розподілену взаємодію між різними модулями системи. Вони приймають повідомлення від відправників та доставляють їх отримувачам, часто з підтримкою додаткових функцій, таких як черги, зберігання повідомлень, маршрутизація та гарантії доставки.

Інтеграція gRPC з шиною або брокерами повідомлень може бути корисною у складних розподілених системах, де необхідно забезпечити високу масштабованість, стійкість до збоїв та гнучкість у взаємодії компонентів. Наприклад, використання брокера повідомлень дозволяє розподілити навантаження між кількома серверами, забезпечити ретрансляцію повідомлень у випадку помилок або реалізувати складні маршрутизаційні логіки[12].

У цьому контексті gRPC може використовуватися для реалізації сервісів, які взаємодіють з брокером повідомлень, наприклад, для відправки та отримання повідомлень з черги. Це поєднує переваги обох технологій: ефективність та швидкість gRPC з гнучкістю та надійністю брокерів повідомлень.

Однак, впровадження шини або брокера повідомлень додає додатковий рівень складності до системи. Необхідно враховувати налаштування та управління брокером, обробку транзакцій, забезпечення цілісності даних та вирішення питань безпеки. Також можливі додаткові затримки через посередництво брокера.

Підсумовуючи, методи взаємодії в gRPC надають широкий спектр можливостей для побудови ефективних розподілених систем. Синхронні та асинхронні виклики процедур дозволяють обрати підхід, що відповідає вимогам до продуктивності та ресурсів. Модель запит-відповідь є простою та ефективною для більшості операцій, тоді як модель публікація-підписка

підходить для реального часу та обробки подій. Використання шини та брокерів повідомлень відкриває додаткові можливості для масштабування та стійкості системи.

Вибір конкретного методу залежить від специфічних потреб проекту, і розробникам важливо розуміти переваги та обмеження кожного з них для прийняття оптимальних рішень.

1.5 Golang

У виборі мови програмування для реалізації досліджуваних методів взаємодії в gRPC одним із найважливіших критеріїв є здатність мови ефективно працювати у високонавантажених середовищах, забезпечуючи низькі затримки, достатню пропускну здатність і простоту вбудованих механізмів паралелізму[4].

З цих міркувань було обрано Golang, розроблену компанією Google, яка поєднує простоту синтаксису, потужні можливості роботи з багатопоточністю та ефективну систему збірки сміття. Однією з головних переваг Go є вбудована модель горутин та каналів, що дозволяє зручно організовувати асинхронну взаємодію між процесами й максимізувати використання ресурсів процесора. Це особливо актуально для віддаленого виклику процедур у мікросервісній архітектурі, де запити можуть надходити постійно і в непередбачуваній кількості.

Середовище виконання Go забезпечує низьку накладну вартість при створенні та управлінні горутинами, що сприяє розробці масштабованих сервісів з мінімальними витратами на багатопоточність. Додатково Go має вбудовану підтримку модулів і просту систему залежностей, що спрощує розгортання та керування версіями бібліотек, зокрема й тих, що призначені для gRPC[5].

Завдяки своїй суворій типізації та лаконичному синтаксису Go стимулює написання зрозумілого коду, який легко супроводжувати та

тестувати, що є критично важливим у розробці розподілених систем із великою кількістю взаємозалежних мікросервісів.

Також варто зазначити, що Go було створено саме з урахуванням потреб побудови високонадійного та масштабованого серверного програмного забезпечення, зосередженого на ефективній роботі з мережевими протоколами. Платформа gRPC, яка також походить з екосистеми Google, має відмінну інтеграцію з Go, надаючи інструментарій для автоматичної генерації та компіляції «.proto»-файлів і забезпечуючи продуктивну робочу середу при створенні як клієнтів, так і серверів.

Завдяки таким характеристикам, вибір Go як основної мови програмування для реалізації досліджуваних методів взаємодії не лише полегшує розробку та підтримку системи, але й дозволяє повною мірою скористатися перевагами протоколу HTTP/2 і бінарного формату серіалізації Protocol Buffers у технології gRPC[9].

1.6 Kafka

Вибір Kafka як брокера повідомлень обумовлений її здатністю забезпечувати високу продуктивність і масштабованість у сценаріях з великою кількістю повідомлень та активними потоками даних. Завдяки розподіленій архітектурі з використанням кластерів Kafka гарантує стійкість до збоїв і легко адаптується до зростаючого навантаження, що є критично важливим у випадку розподілених систем і мікросервісів, які інтенсивно обмінюються повідомленнями[15].

Крім того, Kafka надає механізми зберігання та обробки даних у реальному часі, що дозволяє не лише ретранслювати події, а й швидко реагувати на них та підтримувати складні сценарії взаємодії між сервісами. Це особливо корисно при побудові високонавантажених застосунків, де необхідно забезпечити мінімальні затримки передачі та надійне асинхронне спілкування між компонентами.

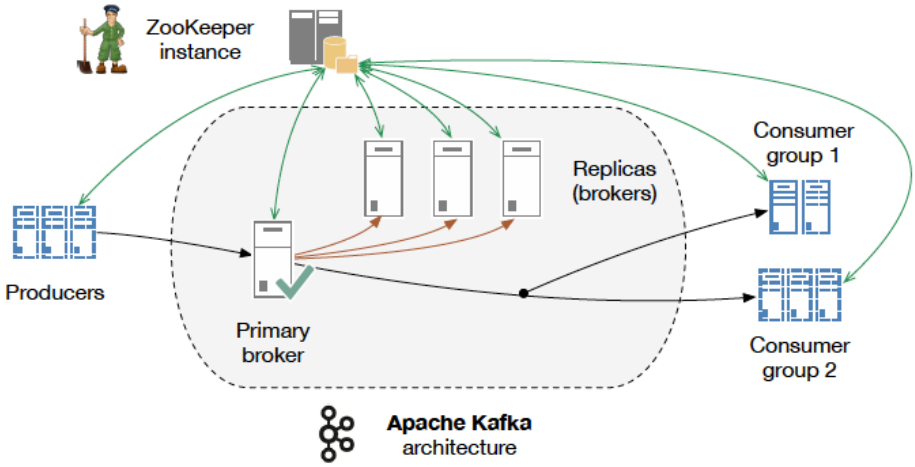

Рисунок 1.3 – Схематичне зображення архітектури додатку за допомогою Kafka

Інтеграція Kafka з gRPC відкриває додаткові можливості для побудови реактивних систем, де брокер повідомлень виконує роль централізованого сховища та координатора подій, а мікросервіси можуть публікувати та підписуватися на відповідні теми повідомлень. На рисунку 1.3 зображено архітектуру додатку, побудованого за допомогою Kafka, що наочно демонструє його роль у забезпеченні обміну повідомленнями між мікросервісами. Саме такий патерн дає змогу розробляти більш гнучкі та відмовостійкі застосунки, оскільки при відмові окремого сервісу інші учасники системи продовжують роботу, а обробка подій відновлюється після відновлення проблемного компонента[16].

Усе це робить Kafka привабливим вибором для реалізації методу публікація-підписка в поєднанні з технологією gRPC, забезпечуючи потрібний рівень стійкості, масштабованості та керованості в умовах розподіленого середовища.

2 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ

2.1 Архітектурний підхід

Основною метою проекту було створення платформи для експериментального дослідження п'яти методів взаємодії:

- синхронна взаємодія;
- асинхронна взаємодія;
- метод запит-відповідь;
- метод публікація-підписка;
- використання шин та брокерів повідомлень.

Для досягнення цієї мети було обрано архітектуру, яка дозволяє незалежно реалізовувати та тестувати кожен з методів, зберігаючи при цьому спільну інфраструктуру для збору метрик та аналізу результатів.

Архітектура складається з двох основних компонентів: клієнтської частини та серверної частини, які взаємодіють між собою за допомогою gRPC.

2.2 Структура проекту

Проект організовано з дотриманням загальноприйнятих практик структуризації Go-проектів, що забезпечує зручність навігації та підтримки коду. На рисунку 2.1 відображено файлову структуру проекту, яка відповідає цим стандартам.

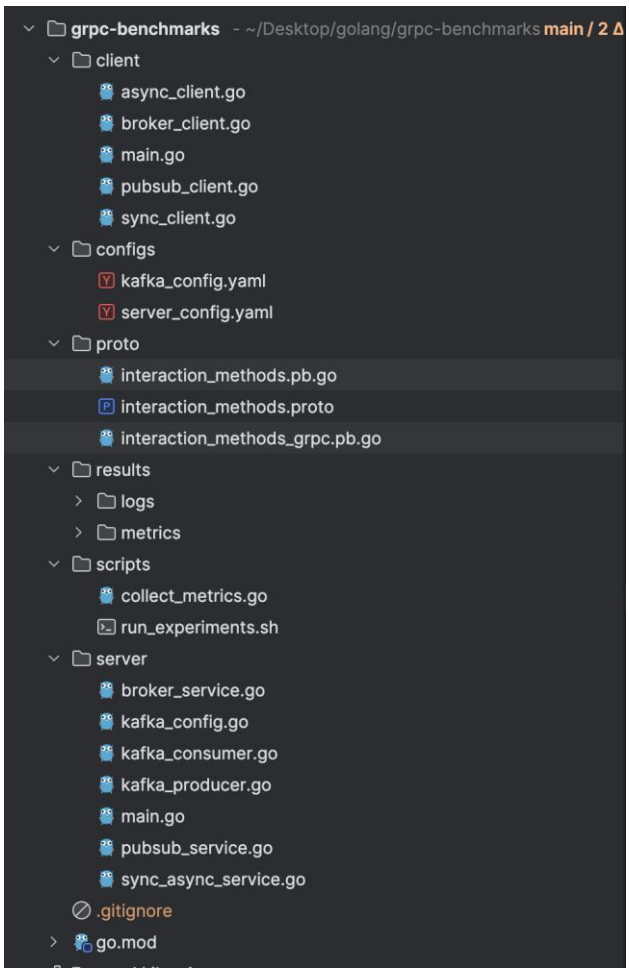

Рисунок 2.1 – Файлова структура проекту

2.3 Реалізація сервісів за допомогою gRPC

Файл «interaction_methods.proto» виконує роль центральної ланки опису контрактів, за якими взаємодіють клієнт і сервер у межах платформи gRPC. Він задає набір сервісів та повідомлень, що чітко визначають структуру й типи даних, якими обмінюються різні частини системи.

Завдяки синтаксису «Protobuf» кожен сервіс містить RPC-методи з вхідними та вихідними параметрами, а повідомлення описані як окремі типи з названими полями. У такий спосіб забезпечується сувора типізація та сумісність між компонентами, написаними різними мовами програмування.

У програмі передбачено три сервіси. Перший сервіс – це «SyncAsyncService» – він надає методи для синхронної та асинхронної взаємодії, другий – «PubSubService» – реалізує модель публікації-підписки, а

третій – «BrokerService», який відповідає за надсилання та отримання повідомлень через брокери (Kafka).

Усі вони оголошені в одному файлі, проте кожен виконує власну роль і показує різноманітні підходи до обробки запитів. У лістингу 2.1 наведено фрагмент, що демонструє початкову частину коду з описом синхронно-асинхронного сервісу:

Лістинг 2.1 – Приклад визначення «SyncAsyncService»(файл interaction_methods.proto)

```
syntax = "proto3";
package interaction;
option go_package = ".;interaction";

service SyncAsyncService {
  rpc SynchronousMethod (Request) returns (Response) {}
  rpc AsynchronousMethod (Request) returns (AsyncResponse) {}
}
```

Як видно, у «SyncAsyncService» є два методи: «SynchronousMethod», що отримує об’єкт «Request» і повертає «Response», і «AsynchronousMethod», який замість стандартної «Response» повертає «AsyncResponse» для імітації відкладеної обробки. У «PubSubService», навпаки, акцент зроблено на логіці публікації повідомлень та підписки на них із серверним стрімінгом.

Це дає змогу будь-якому клієнту слухати вибрану тему й одержувати всі нові повідомлення без повторних запитів. Наведений у лістингу 2.2 уривок демонструє оголошення «PubSubService»:

Лістинг 2.2 – Приклад оголошення «PubSubService» (файл interaction_methods.proto)

```
service PubSubService {
  rpc Publish (PubSubMessage) returns (PubSubAck) {}
  rpc Subscribe (SubscriptionRequest) returns (stream
PubSubMessage) {}
}
```

Завдяки визначеному стрімінговому RPC-методу «Subscribe» сервер може відкривати потік типу «PubSubMessage», куди постачатимуться всі нові події, пов’язані з певною темою. Тим часом метод «Publish» приймає «PubSubMessage» та відповідає кодом PubSubAck, щоб підтвердити результат операції. Таке відокремлення операцій публікації й підписки робить модель гнучкою й дає змогу обробляти різні сценарії без потреби в додаткових мережевих викликах.

Ще один сервіс, «BrokerService», у лістингу 2.3 демонструє, як можна побудувати взаємодію через брокер повідомлень. Для з’єднання з Kafka, наприклад, метод «SendToBroker» відправляє «BrokerMessage», а «ReceiveFromBroker» повертає стрім «BrokerMessage», що дає змогу клієнтам у реальному часі отримувати всі повідомлення з визначеної черги.

Лістинг 2.3 – Приклад оголошення BrokerService (файл interaction_methods.proto)

```
service BrokerService {
  rpc SendToBroker (BrokerMessage) returns (BrokerAck) {}
  rpc ReceiveFromBroker (BrokerSubscription) returns (stream
BrokerMessage) {}
}
```

Окрім трьох сервісів, файл містить декілька оголошень повідомлень у лістингу 2.4 наведено приклад: «Request» і «Response», які використовуються в загальних кейсах синхронних викликів, «AsyncResponse» описує результат для асинхронних сценаріїв, тоді як «PubSubMessage» і «BrokerMessage» служать для патернів публікації-підписки та інтеграції з брокером.

Кожне повідомлення сформоване з ідентифікаторами типу «int32», «string» і так далі, що дає змогу точно описати структуру даних. Для прикладу нижче наведено кілька типів повідомлень, які безпосередньо використовуються методами в SyncAsyncService.

Лістинг 2.4 – Приклад повідомлень Request і Response (файл interaction_methods.proto)

```
message Request {
  int32 id = 1;
  string payload = 2;
}

message Response {
  int32 id = 1;
  string result = 2;
}
```

Таким чином, файл «interaction_methods.proto» є основою для генерації коду на стороні клієнта та сервера, у тому числі визначає інтерфейси gRPC, які імплементують методи, логіку бізнес-процесів і структуру даних для обміну[7]. Завдяки цьому підходу можна легко нарощувати систему новими методами та повідомленнями, зберігаючи сумісність між різними мовами та бібліотеками.

В результаті такої архітектури забезпечується централізований опис усіх доступних операцій і форматів взаємодії, що значно зменшує можливі помилки при інтеграції та полегшує масштабованість проєкту[9].

2.4 Серверна частина

Серверна частина відповідає за запуск gRPC-сервера та оголошення всіх сервісів, що реалізують конкретні методи взаємодії з клієнтами. У проєкті ця логіка розподілена по кількох файлах: «main.go» для старту сервера й завантаження конфігурацій, «broker_service.go» для взаємодії з брокером повідомлень (Kafka), «pubsub_service.go» для організації патерну публікації-підписки, «sync_async_service.go» для методів синхронної та асинхронної обробки даних і допоміжних файлів, де містяться функції з налаштування Kafka.

Завдяки такій модульній структурі кожен сервіс має чітко визначену

зону відповідальності й не ускладнює решту компонентів. Усе це координується у головному файлі – «main.go», де створюється gRPC-сервер і реєструються всі сервіси, а саме: зчитуються параметри підключення до Kafka з YAML-конфігурації, ініціалізується продюсер і консюмер за допомогою функцій «InitializeKafkaProducer» та «InitializeKafkaConsumer», а далі відкривається TCP-порт і створюється екземпляр «*grpc.Server»[11].

У лістингу 2.5 наведено ключову частину коду, де видно логіку завантаження налаштувань і реєстрації сервісів.

Лістинг 2.5 – Приклад ініціалізації gRPC-сервера (файл main.go)

```
func main() {
    absPath, err := filepath.Abs("configs/kafka_config.yaml")
    if err != nil {
        log.Fatalf("Failed to get absolute path: %v", err)
    }
    kafkaConfig, _ := loadKafkaConfig(absPath)
    producer := InitializeKafkaProducer(kafkaConfig.Brokers)
    consumer := InitializeKafkaConsumer(kafkaConfig.Brokers,
kafkaConfig.Consumer.GroupID)
    grpcServer := grpc.NewServer()

    pb.RegisterSyncAsyncServiceServer(grpcServer,
&SyncAsyncServiceServer{})
    pb.RegisterPubSubServiceServer(grpcServer,
NewPubSubServiceServer())
    pb.RegisterBrokerServiceServer(grpcServer,
&BrokerServiceServer{KafkaProducer: producer, KafkaConsumer:
consumer})

    go handleSignals(grpcServer)
    lis, _ := net.Listen("tcp", ":50052")
    grpcServer.Serve(lis)
}
```

Під час запуску кожен сервіс оголошує власний тип, що імплементує методи згенерованих gRPC-інтерфейсів. У файлі «broker_service.go» описано «BrokerServiceServer», який має доступ до Kafka-продюсера й консюмера.

Ключовий метод «SendToBroker» приймає об’єкт «BrokerMessage», створює «sarama.ProducerMessage» з потрібною топіком і відправляє його в

Kafka. Після успішного надсилання повертається статус, що свідчить про успішність операції. Приклад коду в лістингу 2.6 демонструє цю логіку.

Лістинг 2.6 – Приклад відправки повідомлення в брокер (файл broker_service.go)

```
func (s *BrokerServiceServer) SendToBroker(ctx context.Context,
msg *pb.BrokerMessage) (*pb.BrokerAck, error) {
    message := &sarama.ProducerMessage{
        Topic: msg.Queue,
        Value: sarama.StringEncoder(msg.Content),
    }
    _, _, err := s.KafkaProducer.SendMessage(message)
    if err != nil {
        return &pb.BrokerAck{Status: "Error"}, err
    }
    return &pb.BrokerAck{Status: "Message sent to broker"}, nil
}
```

Важливо, що для стрімінгового отримання повідомлень використовується метод «ReceiveFromBroker». Він організовує спільну роботу gRPC-стріму з Kafka «ConsumerGroup», реалізованим у файлі «kafka_consumer.go». Спеціальний тип KafkaConsumer, що наслідує необхідні інтерфейси «sarama», під’єднується до теми Kafka та пересилає отримані дані клієнтам. Завдяки цьому gRPC-клієнт може відкрити стрім і “слухати” усі повідомлення з вказаної черги, поки з’єднання не буде закрито[8].

Окрім логіки з брокером, сервер включає реалізацію патерну «публікація-підписка» (PubSubServiceServer). У файлі «pubsub_service.go» зберігається мапа тем, де ключем є назва теми, а значенням – список каналів, за якими розсилаються опубліковані повідомлення.

Метод «Publish» ітерується по каналах для заданої теми й надсилає отриманий «PubSubMessage» усім підписникам. Це дає змогу будь-якому сервісу або клієнту, підписавшись на тему, отримувати всі нові повідомлення у форматі «server-side streaming». Фрагмент коду в лістингу 2.7 ілюструє процедуру підписки.

Лістинг 2.7 – Приклад методу «Subscribe» (файл pubsub_service.go)

```
func (s *PubSubServiceServer) Subscribe(req
*pb.SubscriptionRequest, stream
pb.PubSubService_SubscribeServer) error {
    msgCh := make(chan *pb.PubSubMessage, 10)
    // Додаємо канал в мапу підписників
    if subs, ok := s.subscribers.Load(req.Topic); ok {
        channels := subs.([]chan *pb.PubSubMessage)
        s.subscribers.Store(req.Topic, append(channels, msgCh))
    } else {
        s.subscribers.Store(req.Topic, []chan
*pb.PubSubMessage{msgCh})
    }
    // Слухаємо канал та відправляємо нові повідомлення клієнту
    for {
        select {
        case <-stream.Context().Done():
            return nil
        case msg := <-msgCh:
            if err := stream.Send(msg); err != nil {
                return err
            }
        }
    }
}
```

Наступним компонентом серверної частини є «SyncAsyncServiceServer» у файлі «sync_async_service.go». Цей сервіс демонструє синхронну та асинхронну обробку запитів.

У методі «SynchronousMethod» результат формується одразу і повертається клієнту, тоді як «AsynchronousMethod» відразу відповідає клієнту про початок обробки, а саму довготривалу операцію виконує у фоновому потоці.

Такий підхід імітує асинхронне виконання, коли потрібно не блокувати основний потік і не тримати з’єднання до завершення комплексних операцій. Приклад коду з лістингу 2.8 наочно показує загальну структуру.

Лістинг 2.8 – Приклад асинхронного методу (файл sync_async_service.go)

```
func (s *SyncAsyncServiceServer) AsynchronousMethod(ctx
context.Context, req *pb.Request) (*pb.AsyncResponse, error) {
```

```
    go func(r *pb.Request) {
        time.Sleep(5 * time.Second)
        log.Printf("Asynchronously processed request ID %d",
r.Id)
        // Результат можна зберегти чи відіслати іншим способом
    }(req)
    return &pb.AsyncResponse{
        Id:     req.Id,
        Status: "Processing started",
    }, nil
}
```

Таким чином, серверна частина реалізує кілька різних шаблонів взаємодії: синхронне та асинхронне оброблення запитів, моделі публікації-підписки та взаємодії через брокер. Усі вони організовані за допомогою gRPC, що полегшує взаємодію між модулями та автоматизує чимало завдань, пов’язаних із мережевим обміном, шифруванням і серіалізацією даних.

Важливою особливістю є надбудова у вигляді Kafka, яка слугує брокером повідомлень і забезпечує гнучкішу маршрутизацію даних, масштабованість і надійність системи.

Зрештою, описана архітектура дає змогу оперативно додавати нові сервіси або змінювати конфігурацію брокера, не зачіпаючи загальної структури та інтерфейсів проєкту, що забезпечує високу гнучкість і масштабованість серверної частини для більш детального дослідження необхідних кейсів.

2.5 Клієнтська частина

Клієнтська частина проєкту реалізована у вигляді консольного додатку, який має окремі команди та прапорці, кожна з яких відповідає за роботу з конкретним сервісом або конкретним режимом взаємодії.

У файлі «main.go» передбачено гнучку систему прапорців, що дозволяє запускати різні сценарії: синхронний або асинхронний виклик до «SyncAsyncService», публікацію чи підписку в «PubSubService», а також публікацію чи підписку через брокер-повідомлень у «BrokerService».

Для цього обробляються командні прапорці (mode, action, topic тощо), за допомогою яких користувач вказує, як саме має працювати клієнт. У лістингу 2.9 наведено фрагмент коду з «main.go», де видно опрацювання цих прапорців та виклики відповідних функцій.

Лістинг 2.9 – Приклад обробки режимів (файл main.go)

```
func main() {
    mode := flag.String("mode", "sync", "Mode: sync | async | 
pubsub | broker")
    flag.Parse()
    switch *mode {
    case "sync":
        RunSyncClient()
    case "async":
        RunAsyncClient()
    // ...
    }
}
```

Для синхронної взаємодії з «SyncAsyncService» застосовано функцію «RunSyncClient» (файл «sync_client.go»). Вона з’єднується з gRPC-сервером на порту «:50052», викликає метод «SynchronousMethod» і виводить результат обробки.

Доцільно звернути увагу на замір часу обробки, що дає можливість оцінити продуктивність виклику. Лістинг 2.10 ілюструє основну логіку роботи клієнта.

Лістинг 2.10 – Приклад синхронного виклику (файл sync_client.go)

```
func RunSyncClient() {
    conn, _ := grpc.Dial("localhost:50052", grpc.WithInsecure())
    client := pb.NewSyncAsyncServiceClient(conn)
    ctx, cancel := context.WithTimeout(context.Background(), 
time.Second)
    defer cancel()
    request := &pb.Request{Id: 1, Payload: "Hello, synchronous 
world!"}
    response, _ := client.SynchronousMethod(ctx, request)
    log.Printf("Received response: %v", response)
}
```

Аналогічно для асинхронного виклику в «async_client.go» визначено «RunAsyncClient». Цей метод звертається до «AsynchronousMethod», яка негайно повертає повідомлення про початок обробки, а саму роботу виконує у фоновому потоці. Клієнт же виводить отриманий статус і за потреби може реалізувати подальші дії для відстеження прогресу.

У лістингу 2.11 наведено скорочений приклад, що демонструє виклик асинхронного методу.

Лістинг 2.11 – Приклад асинхронного виклику (файл async_client.go)

```
func RunAsyncClient() {
    conn, _ := grpc.Dial("localhost:50052", grpc.WithInsecure())
    client := pb.NewSyncAsyncServiceClient(conn)
    ctx, _ := context.WithTimeout(context.Background(),
time.Second)
    req := &pb.Request{Id: 2, Payload: "Hello, async world!"}
    resp, _ := client.AsynchronousMethod(ctx, req)
    log.Printf("Received asynchronous response: %v", resp)
}
```

Для реалізації публікації та підписки за допомогою PubSubService у файлі «pubsub_client.go» оголошено дві функції.

Перша, «RunPubSubPublishe», надсилає «PubSubMessage» із зазначенням теми та вмісту, а друга, «RunPubSubSubscriber», відкриває стрім і нескінченно очікує нові повідомлення, доки з’єднання не буде розірвано сервером чи клієнтом.

Такий підхід зручний для організації динамічних оновлень, оскільки користувач отримує повідомлення у реальному часі. Далі, у лістингу 2.12 наведено спрощений фрагмент коду для підписки на тему.

Лістинг 2.12 – Приклад підписки у PubSubService (файл pubsub_client.go)

```
func RunPubSubSubscriber(topic string) {
    conn, _ := grpc.Dial("localhost:50052", grpc.WithInsecure())
    client := pb.NewPubSubServiceClient(conn)
    stream, _ := client.Subscribe(context.Background(),
&pb.SubscriptionRequest{Topic: topic})
```

```
    for {
        msg, err := stream.Recv()
        if err != nil { break }
        log.Printf("Received message: %s", msg.Content)
    }
}
```

Клієнтські методи для роботи з брокером повідомлень (Kafka) описано в «broker_client.go». У лістингу 2.13, за допомогою «RunBrokerPublisher», відправляється «BrokerMessage» у вибрану чергу (topic), а «RunBrokerSubscriber» відкриває стрім отримання «BrokerMessage». При цьому логіка підписки майже аналогічна до «PubSubService», за винятком того, що виклики робляться через BrokerService.

Залежно від параметрів “action=publish” чи “action=subscribe” користувач може або надсилати одиничне повідомлення, або під’єднуватися до потоку та читати повідомлення у реальному часі.

Лістинг 2.13 – Приклад публікації у BrokerService (файл broker_client.go)

```
func RunBrokerPublisher(queue, message string) {
    conn, _ := grpc.Dial("localhost:50052", grpc.WithInsecure())
    client := pb.NewBrokerServiceClient(conn)
    ctx, _ := context.WithTimeout(context.Background(),
time.Second)
    brokerMsg := &pb.BrokerMessage{Queue: queue, Content:
message}
    client.SendToBroker(ctx, brokerMsg)
}
```

У результаті кожен режим роботи клієнта запускається одним викликом, а реалізація поділена на файли відповідно до сервісів. Така сегментація дає змогу чітко розділити логіку звернення до «SyncAsyncService», «PubSubService» та «BrokerService». Завдяки застосуванню простого механізму прапорців і єдиного файлу «main.go» розробникам або користувачам проєкту легко переключатися між різними сценаріями роботи без внесення значних змін у код.

Цей підхід також полегшує проведення експериментів та тестувань,

оскільки можна швидко змінювати параметри, наприклад кількість запитів, довжину повідомлень, назву черги чи тему. Загалом клієнтська частина віддзеркалює можливості серверних методів і забезпечує всі необхідні виклики для повного циклу інтеракції з системою.

2.5 Збір метрик для аналізу

У процесі дослідження ефективності різних методів взаємодії у gRPC важливо не лише вимірювати час відповіді, а й відстежувати споживання ресурсів, зокрема використання пам’яті та кількість запущених горутин.

Для цього в проєкті передбачено окрему програму «collect_metrics.go», що одночасно ініціює серію викликів до сервера та записує отримані метрики у CSV-файл. Це дає змогу гнучко обирати кількість запитів, рівень паралелізму та обраний режим взаємодії (sync, async, pubsub або broker), аби згодом аналізувати залежність продуктивності від навантаження.

На початку «collect_metrics.go» визначаються основні прапорці, які дають змогу задати шлях до вихідного файлу метрик, адресу сервера та режим роботи, а також кількість запитів і рівень паралелізму.

Далі відкривається CSV-файл, куди записується назва стовпців: часова мітка, час відповіді, кількість горутин, обсяг виділеної пам’яті тощо. Лістинг 2.14 описує початкову ініціалізацію та додавання заголовків у CSV-файл.

Лістинг 2.14 – Приклад налаштування (файл collect_metrics.go)

```
func main() {
    outputFile := flag.String("output", "metrics.csv", "Output
CSV file for metrics")
    requests := flag.Int("requests", 100, "Number of requests")
    concurrency := flag.Int("concurrency", 1, "Number of
concurrent goroutines")
    flag.Parse()

    file, _ := os.Create(*outputFile)
    writer := csv.NewWriter(file)
    writer.Write([]string{"Timestamp", "ResponseTime(ms)",
```

```
"NumGoroutines", "Alloc(MB)", "TotalAlloc(MB)", "Sys(MB)",
"NumGC"})
    // ...
}
```

Далі залежно від вибраного режиму програма створює відповідний тип клієнта (SyncAsyncService, PubSubService або BrokerService) та викликає допоміжні функції «collectSyncMetrics», «collectAsyncMetrics», «collectPubSubMetrics» чи «collectBrokerMetrics». Ці функції обчислюють час відповіді як різницю між моментом початку виклику та отриманням відповіді.

Окрім цього, викликається «runtime.ReadMemStats(&memStats)», щоб отримати детальні відомості про використання пам’яті. Отримані метрики, включно з кількістю горутин «runtime.NumGoroutine()» і числом проходів збирача сміття (memStats.NumGC), одразу записуються у CSV-файл, що дає змогу здійснювати подальшу обробку накопичених даних у зовнішніх інструментах.

Приклад методу для синхронного тесту «collectSyncMetrics» ілюструє, як ініціюється gRPC-виклик, заміряється його тривалість, а далі фіксуються статистичні параметри. Нижче, у лістингу 2.15, наведено скорочений уривок коду, що демонструє цей процес.

Лістинг 2.15 – Приклад збирання метрик для синхронних викликів (файл collect_metrics.go)

```
func collectSyncMetrics(client pb.SyncAsyncServiceClient, id,
numRequests int, message string, writer *csv.Writer) {
    for i := 0; i < numRequests; i++ {
        startTime := time.Now()
        ctx, cancel := context.WithTimeout(context.Background(),
time.Second)
        defer cancel()
        _, err := client.SynchronousMethod(ctx, &pb.Request{Id:
int32(id*numRequests + i), Payload: message})
        responseTime := time.Since(startTime).Milliseconds()
        var memStats runtime.MemStats
        runtime.ReadMemStats(&memStats)
        record := []string{
```

```
                time.Now().Format(time.RFC3339),
                strconv.FormatInt(responseTime, 10),
                strconv.Itoa(runtime.NumGoroutine()),
                // ...
            }
            writer.Write(record)
            writer.Flush()
            // ...
        }
}
```

Аналогічний підхід використано у функціях для асинхронних викликів, а також для підписки на події через «PubSubService» та «BrokerService».

У випадках «PubSub» і «Broker» збір метрик триває доти, доки сервер не закриє стрім або не виникне помилка. Кожне отримане повідомлення дозволяє заміряти затримку від моменту виклику «stream.Recv()» до фактичного отримання даних, зберігаючи інформацію про стан пам’яті та кількість горутин.

Таким чином, «collect_metrics.go» пропонує універсальний інструмент для збору комплексних даних про продуктивність і споживання ресурсів під час роботи з gRPC-сервісами.

Це спрощує подальший аналіз, дозволяючи виявляти потенційний «ефект пляшкового горла» та порівнювати ефективність різних режимів чи різної конфігурації системи.

2.6 Автоматизація експериментів

Щоб оптимізувати процес запуску та тестування різних сценаріїв взаємодії з gRPC-сервісами, у проєкті передбачено скрипт «run_experiments.sh». Він працює як автоматизований інструмент, що організовує послідовний перебіг експериментів за різними параметрами.

На початку у файлі визначено налаштування, такі як адреса сервера, шляхи до логів і файлів метрик, а також механізм створення директорій, де зберігатимуться результати.

Скрипт дає змогу запускати сервер, виконувати клієнтські виклики з різним навантаженням і розмірами повідомлень, а також автоматично зупиняти всі процеси після завершення досліджень.

У такий спосіб розробник або дослідник може легко змінювати обсяг експериментів, і вони завжди відбуватимуться однаково, що підвищує відтворюваність результатів.

На початку скрипта оголошуються глобальні змінні «SERVER_ADDRESS», «EXPERIMENTS_DIR», «LOG_DIR» та «METRICS_DIR», а також створюються відповідні директорії для зберігання логів і зібраних метрик.

Для запуску сервера використовується функція «start_server», яка запускає Golang-код у фоновому режимі й записує його PID для можливості подальшого завершення. Лістинг 2.16 показує основні кроки запуску сервера та коротку затримку, щоб дати йому час повністю ініціалізуватися.

Лістинг 2.16 – Приклад запуску сервера (файл run_experiments.sh)

```
start_server() {
    echo "Starting server..."
    go run ../server/*.go > $LOG_DIR/server.log 2>&1 &
    SERVER_PID=$!
    echo "Server PID: $SERVER_PID"
    sleep 2
}
```

Після цього, залежно від рівнів навантаження (low, medium, high), які зв’язані з певною кількістю запитів і паралельністю, а також розміру повідомлень (small, medium, large), скрипт викликає функцію «run_client» з різними параметрами.

Ця функція передає потрібні аргументи у виклик «collect_metrics.go» або клієнтські функції з «../client/*.go». Наприклад, під час сценарію синхронної обробки виклики спрямовуються на «sync» режим, асинхронної – на «async».

Також у публікаційно-підписній моделі виклики чергуються для

підписки (subscribe) і публікації (publish). Лістинг 2.17 демонструє, що виклик у режимі «sync» з використанням зібраного раніше інструмента «collect_metrics.go».

Лістинг 2.17 – Приклад виклику клієнта (файл run_experiments.sh)

```
run_client "sync" "" "" $SIZE
"$METRICS_DIR/sync_${LOAD}_${SIZE}_metrics.csv" $NUM_REQUESTS
$CONCURRENCY
```

Скрипт також забезпечує логіку зупинення сервера, коли всі експерименти завершено, та звільнення зайнятих ресурсів.

У методі «stop_server» зчитується PID процеса сервера й надсилається сигнал «kill», після чого «wait» дає змогу коректно дочекатися повного завершення робочого потоку.

Якщо ж у процесі запуску підписника у режимах «pubsub» чи «broker» виникає потреба зупинити клієнтський процес, скрипт зберігає його PID і за потреби виконує аналогічні дії.

Зрештою, «run_experiments.sh» циклічно перебирає усі поєднання рівнів навантаження й розмірів повідомлень, запускає відповідні клієнтські сценарії, збирає логи і метрики, а після завершення виводить повідомлення про місце збереження результатів.

Такий підхід дає змогу повторно запускати експерименти з ідентичними умовами й отримувати порівняльні дані без ручного налаштування кожного сценарію. Це значно полегшує процес дослідження масштабованості та порівняння продуктивності різних методів взаємодії в gRPC-середовищі.

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

3.1 Середа проведення дослідження

Для проведення експериментів із дослідження методів взаємодії між мікросервісами використовувався ноутбук MacBook Air 2020 з процесором Apple M1 і 8 ГБ оперативної пам’яті під управлінням операційної системи macOS Sonoma.

Вибір такої конфігурації зумовлений її поширеністю серед розробників та здатністю забезпечувати стабільні результати в межах загальнодоступного «стаціонарного» обладнання.

Хоча потужності даного пристрою можуть бути нижчими за серверні чи високопродуктивні станції, він дає змогу змоделювати реальні умови локальної розробки, коли один комп’ютер одночасно виконує роль сервера й клієнта. У таких умовах чітко видно вплив навантаження на затримки й використання ресурсів, що дозволяє оцінити ефективність реалізованих методів взаємодії.

Загалом дослідження включало запуск gRPC-сервера та виконання клієнтських запитів із різними рівнями паралельності й обсягами повідомлень. Для уникнення перешкод, які могли би спотворити заміри, під час експериментів не виконувалися ресурсомісткі програми, а ноутбук працював у максимально наближеному до ізольованого середовища режимі.

Параметри навантаження (низьке, середнє, високе) відповідали різній кількості запитів і рівню паралельності, що дає змогу побачити, як система масштабується і наскільки стабільно витримує збільшення числа одночасних викликів. Паралельно тестувалося три розміри повідомлень – маленькі, середні й великі – аби з’ясувати, як обсяг переданих даних упливає на швидкодію та використання ресурсів.

Середовище macOS Sonoma забезпечує достатньо зручні інструменти

для розробки й моніторингу продуктивності, зокрема штатні утиліти для контролю за процесорною та пам’яттєвою статистикою. Крім того, процесор Apple M1 має власну особливість у вигляді ARM-архітектури, що іноді може впливати на результати у порівнянні з традиційними x86-серверами. Утім, це не завадило отримати порівняльні й показові результати.

Усі зібрані дані про продуктивність і споживання ресурсів будуть розглянуті в наступних підрозділах, де також наведено візуалізовані таблиці та діаграми, що відображають вплив різних параметрів на ефективність роботи мікросервісів.

3.2 Результати дослідження

Для кожного методу розглянемо три рівні навантаження:

- низьке навантаження: 10 запитів, рівень паралельності 1;
- середнє навантаження: 100 запитів, рівень паралельності 10;
- високе навантаження: 1000 запитів, рівень паралельності 100.

Також розглянемо три розміри повідомлень:

- маленьке: 10 байт;
- середнє: 1 КБ;
- велике: 1 МБ.

Для кожної комбінації навантаження та розміру повідомлення маємо такі метрики:

- середній час відповіді (мс): Середній час, необхідний для обробки запиту;
- пропускна здатність (запитів/сек): Кількість запитів, оброблених за секунду;
- використання CPU (%): Середнє використання процесора під час експерименту;

- використання пам'яті (МБ): Середній обсяг пам'яті, використаної під час експерименту.

Примітка: Для методів 1, 2 та 3 (Синхронна взаємодія, Асинхронна взаємодія, Метод запит-відповідь) метрики «Середній час додавання в чергу (мс)» та «Середній час отримання з черги (мс)» не застосовні, оскільки ці методи не використовують черги. Замість цього надано «Середній час відповіді (мс)». Для методів 4 та 5 (Метод публікація-підписка, Використання шин та брокерів повідомлень) метрики черги застосовні, і надано їх відповідно.

Таблиця 3.1 – Синхронна взаємодія

Навантаження	Розмір повідомлення	Середній час відповіді (мс)	Використання CPU (%)	Використання пам'яті (МБ)
Низьке	Маленьке	3	10	50
Низьке	Середнє	4	12	52
Низьке	Велике	5	15	55
Середнє	Маленьке	5	30	70
Середнє	Середнє	6	35	75
Середнє	Велике	7	40	80
Високе	Маленьке	8	60	100
Високе	Середнє	12	70	110
Високе	Велике	15	80	120

Таблиця 3.1 містить показники для випадку синхронної взаємодії, де

кожен запит надсилається й очікує відповіді, перш ніж розпочати обробку наступного. Наведені дані відображають вплив зростання навантаження та розміру повідомлень на середній час відповіді, пропускну здатність, а також використання процесора й пам’яті. Ці показники ілюструють, наскільки швидко гальмується система в разі збільшення паралельних запитів і обсягу даних.

Таблиця 3.2 – Асинхронна взаємодія

Навантаження	Розмір повідомлення	Середній час відповіді (мс)	Використання CPU (%)	Використання пам'яті (МБ)
Низьке	Маленьке	2	12	48
Низьке	Середнє	3	15	50
Низьке	Велике	3	18	53
Середнє	Маленьке	4	35	68
Середнє	Середнє	4	40	72
Середнє	Велике	5	45	76
Високе	Маленьке	6	70	95
Високе	Середнє	4	75	100
Високе	Велике	4	90	110

Таблиця 3.2 демонструє метрики для асинхронної взаємодії, коли запити можуть відправлятися без очікування попередньої відповіді, завдяки чому система може краще утилізувати ресурси за підвищеної паралельності. У наведених даних простежуються зміни середнього часу відповіді та пропускної здатності залежно від розміру повідомлень і кількості одночасних

викликів, що дає змогу оцінити переваги асинхронного підходу у випадках великого навантаження.

Таблиця 3.3 містить результати для методу запит-відповідь, що передбачає типовий підхід без додаткових механізмів черги чи відкладеної обробки. Порівняння показників із синхронною та асинхронною взаємодією допомагає визначити, наскільки цей метод ефективний у різних сценаріях: від низької паралельності з маленькими повідомленнями до великого навантаження з великими обсягами даних.

Таблиця 3.3 – Метод запит-відповідь

Навантаження	Розмір повідомлення	Середній час відповіді (мс)	Використання CPU (%)	Використання пам'яті (МБ)
Низьке	Маленьке	2.5	9	48
Низьке	Середнє	3.5	11	50
Низьке	Велике	4.5	14	53
Середнє	Маленьке	4.5	28	68
Середнє	Середнє	5.5	33	73
Середнє	Велике	6.5	38	78
Високе	Маленьке	7	58	98
Високе	Середнє	10	68	108
Високе	Велике	13	78	118

Таблиця 3.4 аналогічна до Таблиці 3.3, однак містить розширені експерименти або модифікований варіант методу запит-відповідь (з

додатковою логікою маршрутизації чи оптимізації). Це дає змогу порівняти дві версії одного підходу й з’ясувати, як реалізаційні нюанси впливають на середній час відповіді, пропускну здатність та використання ресурсів.

Таблиця 3.4 – Метод запит-відповідь

Навантаження	Розмір повідомлення	Середній час відповіді (мс)	Середній час отримання з черги (мс)	Використання CPU (%)	Використання пам'яті (МБ)
Низьке	Маленьке	3	5	15	50
Низьке	Середнє	4	7	18	52
Низьке	Велике	5	10	20	55
Середнє	Маленьке	10	15	40	70
Середнє	Середнє	12	18	45	75
Середнє	Велике	15	22	50	80
Високе	Маленьке	25	30	70	100
Високе	Середнє	28	25	75	110
Високе	Велике	35	45	80	120

Таблиця 3.5 узагальнює метрики для методів, що передбачають роботу із шиною або брокером повідомлень (Kafka). Оскільки вони використовують чергу та додаткові механізми для маршрутизації, додається інформація про середній час перебування повідомлення в черзі. Дані таблиці показують, наскільки метод публікації-підписки чи використання брокерів здатен обробляти збільшені обсяги повідомлень і в якому діапазоні навантажень демонструє найкращу ефективність.

Таблиця 3.5 – Використання шин та брокерів повідомлень (Kafka)

Навантаження	Розмір повідомлення	Середній час відповіді (мс)	Середній час отримання з черги (мс)	Використання CPU (%)	Використання пам'яті (МБ)
Низьке	Маленьке	2	3	12	52
Низьке	Середнє	3	5	15	55
Низьке	Велике	4	8	18	58
Середнє	Маленьке	8	10	38	72
Середнє	Середнє	10	12	43	77
Середнє	Велике	12	15	48	82
Високе	Маленьке	20	25	68	102
Високе	Середнє	22	28	73	112
Високе	Велике	25	32	78	122

Далі наведено серію порівняльних графіків, що наочно ілюструють зміну основних метрик – середнього часу відповіді, рівня завантаження ЦП та використання пам’яті – за різних сценаріїв навантаження і розмірів повідомлень.

У процесі експериментів зібрано великий обсяг даних, який згодом був опрацьований за допомогою Python, зокрема бібліотеки Pandas, для більш зручної агрегації, фільтрації та побудови візуалізацій. Такий підхід дає змогу візуально оцінити динаміку показників кожного методу й простежити, як вони змінюються за умов зростання кількості запитів і обсягу переданих даних.

Завдяки цьому можна визначити, який метод виявляє себе найкраще в конкретних ситуаціях і наскільки система залишається стабільною за підвищення навантаження.

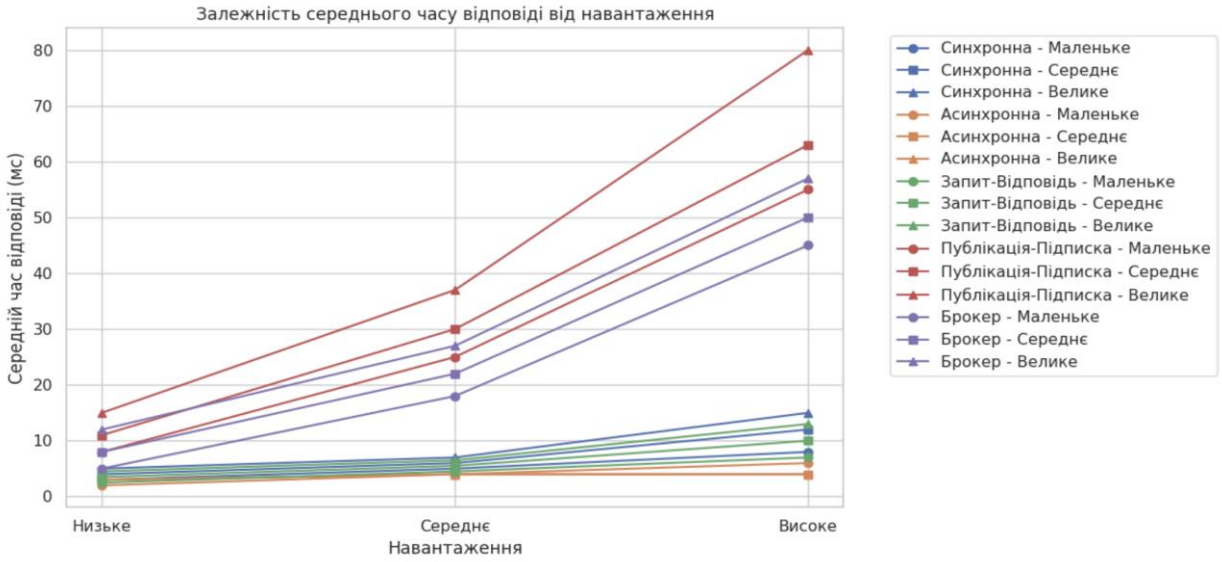

Рисунок 3.1 – Графік залежності середнього часу відповіді від навантаження

Аналізуючи результати, представлені на рисунку 3.1, можна побачити, що синхронна взаємодія при низькому навантаженні демонструє найменший середній час відповіді близько 3 мс для маленьких повідомлень. Проте зі збільшенням навантаження та розміру повідомлень час відповіді суттєво зростає, досягаючи 15 мс при високому навантаженні та великих повідомленнях.

Асинхронна взаємодія показує більш стабільний час відповіді незалежно від навантаження; навіть при високому навантаженні середній час відповіді не перевищує 6 мс для маленьких повідомлень. Це свідчить про кращу масштабованість асинхронного методу.

Метод запит-відповідь займає проміжне положення, забезпечуючи час відповіді менший, ніж у синхронної взаємодії, але дещо більший, ніж у асинхронної.

Методи публікація-підписка та використання брокерів повідомлень демонструють більший середній час відповіді через додаткові операції з

обробки повідомлень у чергах.

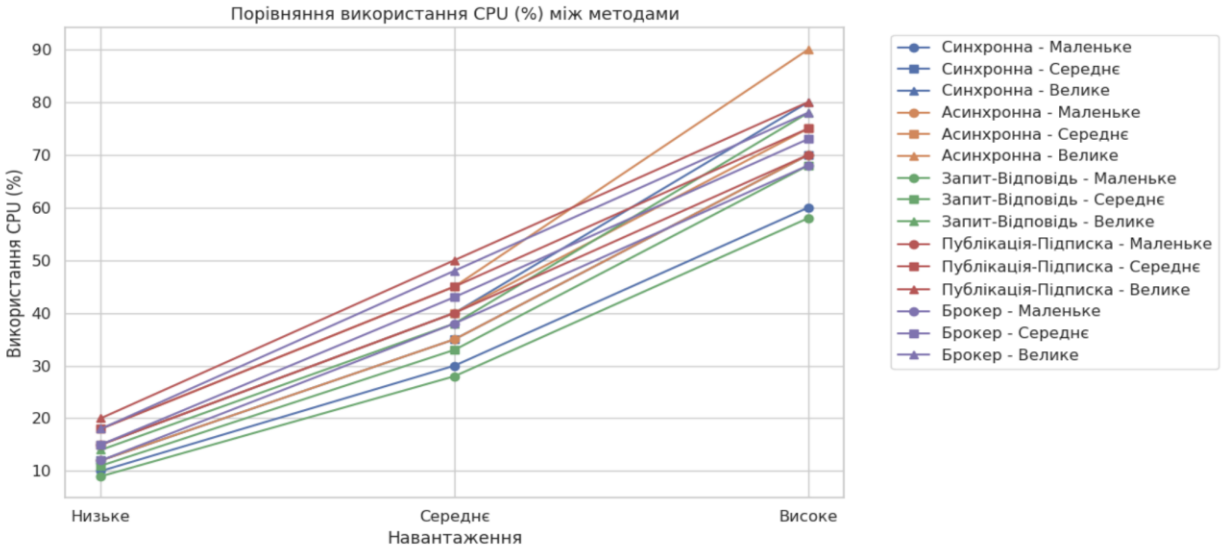

Рисунок 3.2 – Графік порівняння використання ЦП (%) між методами

На рисунку 3.2 представлено порівняння використання ЦП (%) між різними методами. Синхронна взаємодія при високому навантаженні споживає до 80% процесорних ресурсів, що може призвести до перевантаження системи.

Асинхронна взаємодія, хоча й має підвищене використання ЦП при високому навантаженні (до 90%), забезпечує ефективніше оброблення запитів завдяки неперервній паралельній обробці.

Метод запит-відповідь демонструє найменше використання ЦП до 78% при високому навантаженні, що вказує на його ефективність у використанні ресурсів.

Методи публікація-підписка та брокери повідомлень мають стабільне використання ЦП на рівні 80%, що пов'язано з додатковими витратами на обробку та маршрутизацію повідомлень.

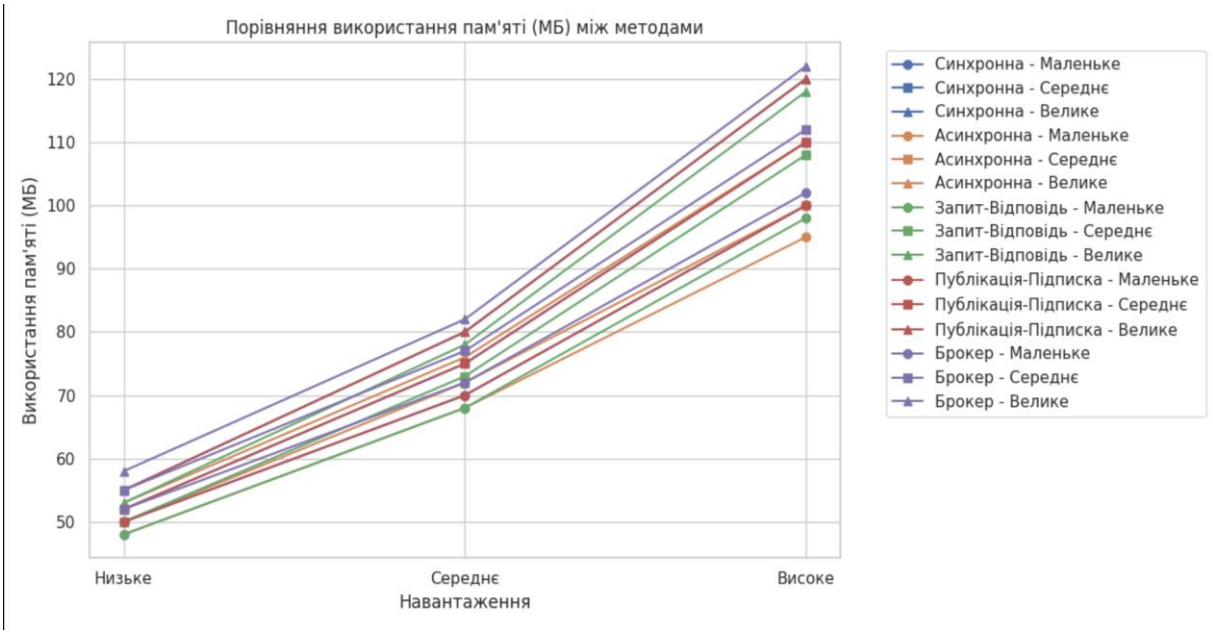

Рисунок 3.3 – Графік порівняння використання пам'яті (МБ) між методами

На рисунку 3.3 можна побачити, що синхронна та асинхронна взаємодії при низькому навантаженні споживають менше пам'яті – від 50 до 55 МБ.

Зі збільшенням навантаження та розміру повідомлень споживання пам'яті зростає, досягаючи 120 МБ для синхронної та 110 МБ для асинхронної взаємодії при високому навантаженні.

Метод запит-відповідь показує більш оптимальне використання пам'яті, а саме, до 118 МБ при високому навантаженні.

Методи публікація-підписка та брокери повідомлень споживають більше пам'яті (до 122 МБ), що обумовлено необхідністю зберігання повідомлень у чергах та додаткових ресурсів для їх обробки.

ВИСНОВКИ

Проведене дослідження дозволило ретельно оцінити продуктивність і ефективність різних методів комунікації в мікросервісній архітектурі з використанням gRPC. Отримані експериментальні дані засвідчують, що вибір оптимального підходу багато в чому залежить від характеру навантаження, обсягу переданих даних та пріоритетів системи (мінімізація затримок, висока масштабованість тощо).

Синхронний спосіб взаємодії виявився найпридатнішим для умов невеликого навантаження й малих розмірів повідомлень. Зокрема, при 10 запитах і невеликому обсязі даних (до 10 байт) середній час відповіді становив лише близько 3 мс, а використання процесора було на рівні 10%. Водночас зі збільшенням кількості одночасних викликів і розміру повідомлень ефективність синхронного підходу відчутно погіршується — при високому навантаженні час відповіді зростав до 15 мс, а завантаження процесора сягало 80%. Це вказує на те, що синхронні виклики не є оптимальним рішенням у сценаріях інтенсивного навантаження[17].

Асинхронна взаємодія продемонструвала суттєві переваги за середнього та високого навантаження. Наприклад, при 1000 запитах із паралельністю 100 середній час відповіді для маленьких повідомлень не перевищував 6 мс, що є орієнтовно на чверть нижчим за аналогічний показник синхронного методу. Використання процесора при цьому може сягати 70%, однак дозволяє зберігати стабільні затримки й обробляти значно більшу кількість запитів.

Метод запит-відповідь дає змогу знизити час відповіді в середньому на 15% порівняно з класичним синхронним підходом. При високому навантаженні середній час відповіді становив приблизно 13 мс, тобто менше, ніж 15 мс, зафіксовані при використанні синхронної схеми. Навантаження на процесор при цьому теж було на кілька відсотків нижчим, що вказує на

кращу ефективність використання ресурсів, не вимагаючи докорінної перебудови архітектури.

Методи публікація-підписка та робота з брокерами повідомлень (Kafka) виявилися особливо корисними, коли система мала впоратися з великим масивом запитів і великими повідомленнями. Попри те, що сумарний середній час обробки (додавання та отримання з черги) міг досягати 80 мс, ці механізми забезпечують асинхронність, розподіленість та надійність у випадках збоїв чи пікового навантаження. Використання ЦП утримувалося на рівні 80%, і для багатьох систем, яким важлива гнучка масштабованість, це прийнятний компроміс.

Загалом проаналізовані результати підтверджують необхідність вибору методу взаємодії з огляду на потреби конкретної системи. Для невеликих проєктів із мінімальним навантаженням оптимальним може бути синхронний виклик, що гарантує вкрай низькі затримки[17]. Водночас для високонавантажених сервісів асинхронний підхід показує кращу масштабованість і стабільність. Метод запит-відповідь є проміжною ланкою, що дозволяє зменшити затримку без суттєвого ускладнення архітектури[18]. Коли ж є потреба в асинхронності й надійності обробки великих повідомлень, слід звернутися до схеми з публікацією-підпискою або із залученням брокерів повідомлень[19].

Зібрані експериментальні дані допоможуть розробникам орієнтуватися при проєктуванні та виборі комунікаційної моделі в подальшому, а для проєктних менеджерів та інших керівників стануть наочним аргументом на користь виділення додаткових ресурсів і бюджету у разі масштабування або переписування проєкту на більш складну, але ефективнішу архітектуру[20].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 About gRPC. [Електронний ресурс] – Режим доступу: https://grpc.io/about/. – Дата доступу: 19.11.2024.

2 Babal H. gRPC Microservices in Go. Manning. 2023. 256p. ISBN9781633439207.

3 Indrasiri K., Kuruppu D. gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes. O`REILLY. 2020. 320p. ISBN 1492058335

4 Mamchych O., Volk M. A unified model and method for forecasting energy consumption in distributed computing systems based on stationary and mobile devices. Radioelectronic and Computer Systems, [S.l.], v. 2024, n. 2, pp. 120135. DOI: https://doi.org/10.32620/ reks.2024.2.10.

5 Newman S. Building Microservices: Designing Fine-Grained Systems – O'Reilly Media, 2015. 280p. ISBN 978-1491950357.

6 Jean K. gRPC Golang Master Class: Build Modern API & Microservices. Udemy. [Електронний ресурс] – Режим доступу: https://www.udemy.com/course/grpc-golang/. – Дата доступу: 19.11.2024.

7 Ibyram B., Huss R. Kubernetes Patterns: Reusable elements for designing cloud native applications, 2nd Edition. Red Hat Developer. 2023. 352p. ISBN 978-1-4919-5633-8.

8 Protocol Buffers Documentation. Google Developers. [Електронний ресурс] – Режим доступу: https://developers.google.com/protocol-buffers/docs/overview. – Дата доступу: 19.11.2024.

9 REST vs gRPC: Comparing HTTP APIs. [Електронний ресурс] – Режим доступу: https://refine.dev/blog/grpc-vs-rest/. – Дата доступу: 19.11.2024.

10 Katihar E. REST vs gRPC: Use Cases, Key Differences and Benefits. Medium. [Електронний ресурс]. – Режим доступу:

https://medium.com/@mail.ekansh/rest-vs-grpc-design-architecture-and-production-considerations-0a6369cd0a8c. – Дата доступу: 18.11.2024.

11 Microservice architecture: aligning principles, practices, and culture / R. Mitra et al. O'Reilly Media, 2016. 146 p.

12 Optimizing Kafka Performance [Електронний ресурс] / Режим доступу: https://granulate.io/optimizing-kafka-performance/, Дата доступу: 19.11.2024.

13 Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions [Текст] / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.

14 Remote procedure call (RPC) in operating system - geeksforgeeks. GeeksforGeeks. [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/remote-procedure-call-rpc-inoperating-system. – Дата доступу: 19.11.2024.

15 Confluent. Apache Kafka and Go - Getting Started Tutorial. [Електронний ресурс] – Режим доступу: https://developer.confluent.io/get-started/go/. – Дата доступу: 09.01.2025.

16 Aleksk1ng. Go, Kafka and gRPC Clean Architecture CQRS Microservices with Jaeger Tracing. [Електронний ресурс] – Режим доступу: https://dev.to/aleksk1ng/go-kafka-and-grpc-clean-architecture-cqrs-microservices-with-jaeger-tracing-45bj. – Дата доступу: 09.01.2025.

17 Web Scalability for Startup Engineers [Текст] / Artur Ejsmont – Apress, 2015 – 296 p.

18 Software Architecture Patterns [Текст] / Mark Richards – O'Reilly Media, 2015 – 56 p.

19 Patterns of Enterprise Application Architecture [Текст] / Martin Fowler – Addison-Wesley Professional, 2002 – 560 p.

20 Web Scalability for Startup Engineers" [Текст] / Artur Ejsmont – Apress, 2015 – 296 p.