МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Центр післядипломної освіти

Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

. другий (магістерський) .

(рівень вищої освіти)

Дослідження методів оптимізації трафіку між сервісами через використання системи gRPC у мікросервісній архітектурі

Виконав:

студент групи ІПЗздм-21-1

Лисяков М. С.

(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-наукова

Керівник доц., к.т.н, Вечур О. В.

(посада, прізвище, ініціали)

Допускається до захисту

Зав. Кафедри _____ З.В. Дудар

2023 р.

Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти
Кафедра Програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«_____»_____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Лисякова Микити Сергійовича
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів оптимізації трафіку між сервісами через використання системи gRPC у мікросервісній архітектурі»
затверджена наказом по університету від 03.04.2023 р. № 83Стз
2. Термін подання студентом роботи до екзаменаційної комісії 15.05.2023 р.
3. Вихідні дані до роботи мікросервісна архітектура, між процесна взаємодія, порівняння продуктивності мережевих викликів
3. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз предметної галузі, дослідження актуальності між процесної комунікації, порівняння двох протоколів, побудова тестової системи для апробації теоретичних відмінностей, аналіз отриманих результатів.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
2	Аналіз предметної галузі	01.04.2023 – 10.04.2023	Виконано
3	Розробка вимог для програмного забезпечення	10.04.2023	Виконано
4	Проектування програмного забезпечення	10.04.2023 - 11.04.2023	Виконано
5	Розробка програмного забезпечення	11.04.2023 - 15.04.2023	Виконано
6	Підготовка пояснювальної записки	15.04.2023 - 25.04.2023	Виконано
7	Підготовка презентації та доповіді	25.04.2023 – 01.05.2023	Виконано
8	Перевірка на плагіат	09.05.2023	Виконано
9	Перевірка на нормоконтроль	10.05.2023	Виконано
10	Попередній захист кваліфікаційної роботи	13.05.2023	Виконано
11	Захист кваліфікаційної роботи	16.05.2023	Виконано

Дата видачі завдання «01» квітня 2023 р.

Студент _____ Лисяков М. С.
(підпис)

Керівник роботи _____ доц., к.т.н, Вечур О. В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 75 с., 21 рис., 4 табл., 35 джер. та 7 додатків.

МІКРОСЕРВІС, ВЗАЄМОДІЯ МІЖ ПРОЦЕСАМИ, ЕФЕКТИВНІСТЬ МЕРЕЖЕВОЇ КОМУНІКАЦІЇ, REST, HTTP, JSON, GRPC, PROTOBUF

Об’єктом дослідження виступає між процесна взаємодія в архітектурі мікросервісів. Предметом дослідження виступає ефективність системи в залежності від обраного методу між процесної взаємодії.

Метою кваліфікаційної роботи є оцінка впливу на продуктивність сервісу використання фреймворку gRPC у порівнянні з REST API. Для досягнення поставленої мети було розглянуто теоретичні відмінності між двома підходами та побудовано тестову систему для отримання відносних показників продуктивності за допомогою навантажувального тестування.

Отримані результати підтверджують результати аналогічних досліджень про перевагу gRPC, але відносні показники менше ніж в розглянутих дослідженнях. Також, в дослідженні було отримано тестовий сценарій за яким пропускна здатність REST була вище ніж gRPC.

Результати дослідження можуть бути використані розробниками при прийнятті рішення щодо вибору методу між процесної взаємодії при побудові систем на мікросервісній архітектурі. Також в роботі запропоновано дизайн системи для використання переваг обох фреймворків.

За результатами дослідження було отримано наступні результати: для систем з великим навантаженням gRPC продемонстрував значно кращі результати. В середньому пропускна здатність, а відповідно і час обробки запиту, від 4 до 7 разів вища ніж REST. Лише за одним сценарієм REST показав кращий результат, при

передачі малих за розміром повідомлень в низько навантаженій системі. Також gRPC використовував менше ресурсів системи для обробки запитів.

MICROSERVICE, INTER-PROCESS COMMUNICATION, EFFICIENCY OF NETWORK COMMUNICATION, REST, HTTP, JSON, GRPC, PROTOBUF

The research focuses on inter-process communication in microservices architecture. Specifically, it evaluates the efficiency of the system depending on the chosen method of inter-process communication.

The aim of the study is to compare the performance of gRPC and REST API frameworks. Theoretical differences between the two approaches were considered and a test system was built to obtain relative performance indicators using load testing.

Results from the study confirm the superiority of gRPC, but its relative performance is less than in the studies reviewed. Additionally, a test scenario was found where REST throughput was higher than gRPC.

The study's results can help engineers decide on the method of inter-process communication when building systems on a microservice architecture. The paper also proposes a system design that takes advantage of both frameworks.

The study found that for systems with high loads, gRPC demonstrated significantly better results, achieving a throughput 4 to 7 times higher than REST. In only one scenario did REST show better results, when transmitting small messages in a low-load system. Also, gRPC used fewer system resources to process requests.

Умови публікації пояснювальної записки

Я, Лисяков Микита Сергійович

(прізвище, ім’я, по батькові)

студент групи ІПЗздм-21-1 здобувач вищої освіти на другому (магістерському) рівні

кафедра програмної інженерії

(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему «Дослідження методів оптимізації трафіку між сервісами через використання системи gRPC у мікросервісній архітектурі», що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ПЕРЕЛІК СКОРОЧЕНЬ

API	Application Programming Interface
RPC	Remote procedure call
REST	Representational State Transfer
HTTP	HyperText Transfer Protocol
gRPC	Google Remote Procedure Call
Protobuf	Protocol Buffers
MSA	Microservice archirecture
IPS	Inter-Process Communication
JSON	JavaScript Object Notation
XML	Extensible Markup Language

ЗМІСТ

ВСТУП

Стиль мікросервісної архітектури став одним з найпопулярніших архітектурних стилів розробки програмного забезпечення в останні роки. Згідно з дослідженням, проведеним компанією Statista [1], 81,5% компаній вже використовують мікросервісну архітектуру, а 17,5% компаній планують перейти на цей тип архітектури. Лише 1% респондентів задоволені монолітною архітектурою. Згідно з іншими дослідженнями проведеними компанією RedHat [2], 73 % компаній вже використовують мікросервісну архітектуру.

Такі науковці ХНУРЕ як Дудар З.В. та Широкопетлєва М.С. досліджували питання вибору оптимальної архітектури для побудови сучасних систем та також дійшли висновків щодо переваги мікросервісної архітектури [3].

Оскільки системи на основі мікросервісів є розподіленими, однією з ключових проблем при розробці програми є вибір механізму, за допомогою якого сервіси взаємодіють один з одним. Існує декілька підходів для реалізації між процесної взаємодії (англ. inter-process communication, IPC) в мікросервісах, і кожен з них має свої переваги та компроміси.

Допомогти знайти оптимальний вибір для між процесної комунікації є ключовою проблемою, на вирішення якої спрямована дана робота. Комунікація відіграє набагато важливішу роль в архітектурі мікросервісів, ніж в монолітній архітектурі, оскільки в останній, модулі можуть отримувати доступ та викликати один одного на рівні мови програмування, в той час, як мікросервіси структурують систему як набір незалежних розподілених сервісів з можливістю того, що кожен сервіс може працювати на власному сервері.

Метою кваліфікаційної роботи є оцінка впливу на ефективність сервісу використання фреймворку gRPC у порівнянні з REST API. Для досягнення

поставленої мети було розглянуто теоретичні відмінності між двома підходами та побудовано тестову систему для отримання відносних показників ефективності за допомогою навантажувального тестування.

Об’єктом дослідження виступає міжпроцесна взаємодія в архітектурі мікросервісів. Предметом дослідження виступає ефективність системи в залежності від обраного методу між процесної взаємодії.

Аналогічні дослідження стверджують, що gRPC є швидшим та має вищу пропускну здатність порівняно з REST, зокрема при використані функції потокової передачі даних. Дослідження також підтверджують перевагу gRPC над REST з точки зору безпеки зв'язку та мережевої затримки. Отримані результати цього дослідження також підтверджують перевагу gRPC, але відносні показники менше ніж в аналогічних дослідженнях. Також, в дослідженні було отримано тестовий сценарій за яким пропускна здатність REST була вище ніж gRPC.

Результати дослідження можуть бути використані розробниками при прийнятті рішення щодо вибору методу між процесної взаємодії. Також в роботі запропоновано дизайн системи для використання переваг обох фреймворків.

1 АКТУАЛЬНІСТЬ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА МІЖ ПРОЦЕСНОЇ ВЗАЄМОДІЇ

1.1 Теоретичні аспекти монолітної архітектури, її переваги та недоліки

Монолітний дизайн - це підхід, при якому всі обробники бізнес-логіки містяться в одній великій системі. Зазвичай, компоненти в такій системі взаємопов'язані та залежать один від одного [4]. Кожен компонент є необхідним для функціонування всієї системи. Якщо один компонент виходить з ладу, то код всього проєкту, як правило, не компілюється.

У монолітній архітектурі єдину систему підтримує одна база даних, тому всі сервіси будуть посилатися на неї. Нижче наведено приклад архітектури монолітної системи для сайту електронної комерції.

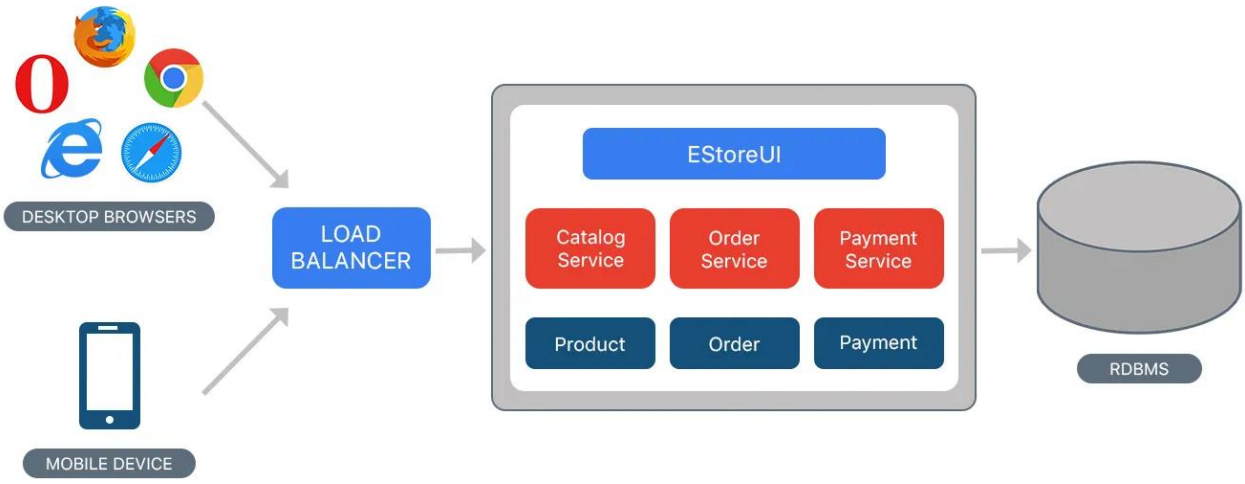

Рисунок 1.1 - Приклад монолітної архітектури [4]

Як показано на малюнку 1, обмін даних відбувається виключно між єдиним сервером та клієнтами. Весь сервіс звертається до єдиної бази даних. Усі обробники бізнес-логіки, включно з обробником аутентифікації та іншими допоміжними

сервісами тісно пов'язані. Попри наявність декількох модулів, увесь код компілюється і розгортається як єдиний великий застосунок.

Монолітна архітектура є поширеним вибором для сучасних застосунків, пропонуючи низку переваг, які роблять її доцільним варіантом у певних сценаріях.

Однією з її основних переваг є простота налаштування, що дозволяє розробникам майже одразу розпочати кодування. Саме тому монолітна архітектура є першим вибором для проєктів "з нуля". У сучасному світі люди зазвичай створюють мінімально життєздатний продукт або швидкий демонстраційний варіант перед ухваленням рішення про проєкт. Монолітна архітектура в цьому випадку особливо корисна.

Простота розробки призводить до простоти тестування. Монолітна архітектура також забезпечує простий підхід до тестування повного циклу. Оскільки все працює в межах одного сервісу, тестування повного циклу не вимагає комунікації між різними сервісами. Крім того, монолітні системи легко масштабуються за допомогою горизонтального підходу. Для цього використовують балансувальника навантаження, який запускає кілька серверів у разі надзвичайних ситуацій. Нарешті, головною перевагою монолітного підходу є простота процесу розгортання, оскільки розгортається лише один пакет.

Хоча початкове налаштування монолітної архітектури є простим, вона має кілька недоліків, які роблять її непридатною для великих і складних систем.

Обслуговування монолітної системи може стати обтяжливим у міру зростання системи. На відміну від мікросервісів, кожен компонент в монолітній архітектурі використовується в різних місцях і впливає на багато областей всередині системи, що ускладнює зміну навіть невеликої частини коду без змін багатьох компонентів системи. Крім того, надійність монолітного підходу не на найвищому рівні. Єдина помилка в одному компоненті може бути фатальної для всієї системи оскільки масштабування монолітної системи відбувається горизонтально.

Хоча розгортання монолітної системи може бути простим, воно не є ефективним, оскільки при публікації будь-яких змін треба заново розгортати всю систему. Це також означає, що при монолітному підході майже неможливо впровадити нову технологію без значних змін. Зазвичай, для впровадження нової технології потрібно переписати всю кодову базу [4].

1.2 Еволюція розподілених систем

Більшість сучасних корпоративних систем є розподіленими через мережу Інтернет або як сукупність локальних мереж [5]. Попит на розробку розподілених систем значно зріс за останні роки, оскільки додатки тепер повинні обслуговувати широке коло споживачів у різних географічних точках у режимі реального часу. Спочатку більшість програмних систем не повинні були враховувати мережевий рівень, щоб дозволити іншим пристроям взаємодіяти один з одним. З розвитком мережевих можливостей протягом останнього десятиліття набула популярності клієнт-серверна архітектура, яка дозволяє різним користувачам надсилати запити до одного або більше серверів, очікувати відповіді та обробляти їх при отриманні. У моделі клієнт-сервер передбачається, що клієнт має обмежені обчислювальні можливості, а отже, вся обробка повязана з бізнес-логікою повинна відбуватися на стороні сервера.

Мобільний агент [6] - ще одна парадигма розподілених систем, що з'явилася після моделі клієнт-сервер. Мобільні агенти - це автономні програми, які можуть переміщатися від комп'ютера до комп'ютера в мережі, в час і в місця за власним вибором. Стан запущеної програми зберігається, передаючись до місця призначення

Згодом виникла сервісно-орієнтована архітектура (англ. Service Oriented Architecture, SOA) [7], яка запропонувала інтеграційне рішення, що дозволило різним розподіленим системам взаємодіяти одна з одною. У підході SOA клієнт обмінюється повідомленнями з іншими компонентами за допомогою віддаленого виклику процедур, таких як Simple Object Access Protocol (SOAP) [8]. SOA дозволяє програмним компонентам працювати один з одним через декілька мереж незалежно від платформ або мов програмування.

Сервіс-орієнтована архітектура є масштабованою та високонадійною, а також забезпечує незалежність від платформи. Однак SOA є складною і повільною з високими накладними витратами. Кожен сервіс повинен забезпечувати своєчасну доставку повідомлень. Кількість цих повідомлень можеперевищувати мільйон за один раз, що ускладнює управління всіма службами. Також, всі вхідні дані перевіряються до того, як один сервіс взаємодіє з іншим сервісом. При використанні декількох сервісів це збільшує час відгуку і знижує загальну продуктивність системи.

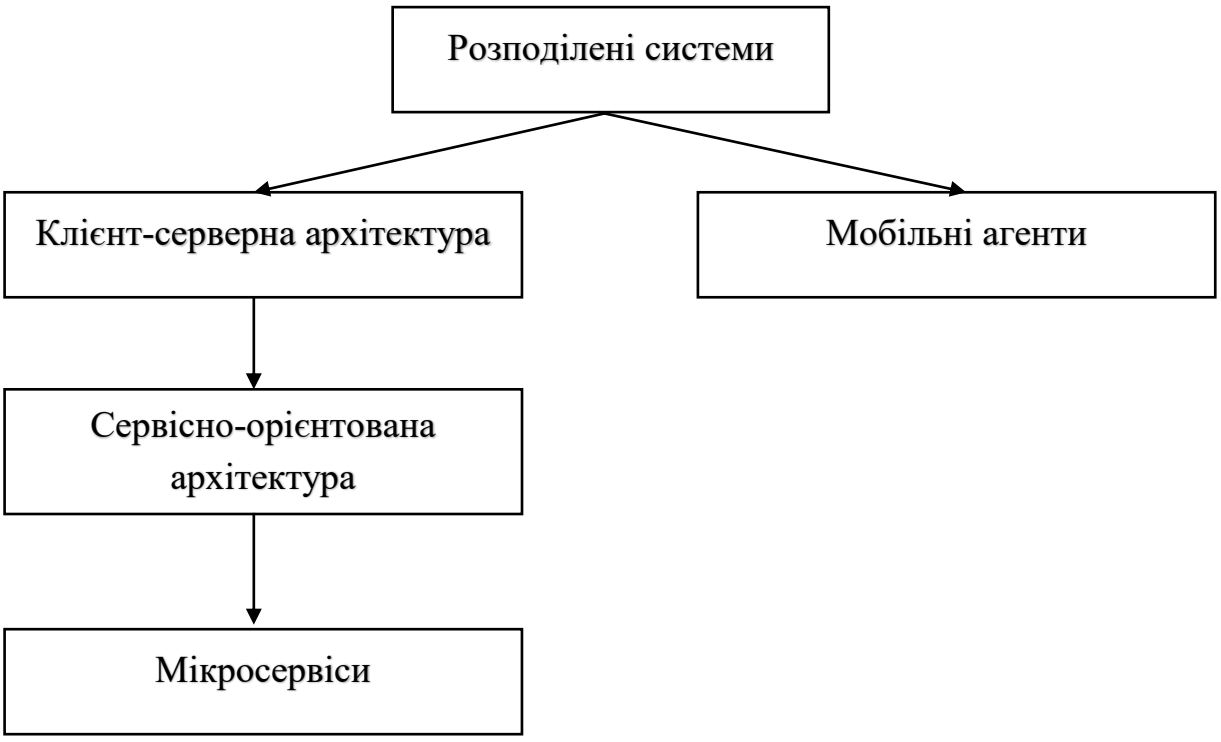

Рисунок 1.2 – Еволюція розподілених систем [9]

Через ці недоліки були потрібні більш легкі комунікаційні механізми, щоб задовольнити потреби бізнесу, зберігаючи при цьому масштабованість і надійність, які пропонує SOA, що стало мотивом для створення більш легких комунікаційних механізмів, таких як REST (скор. англ. Representational State Transfer), і більш легкої архітектури, відомої як мікросервіси.

1.3 Теоретичні аспекти мікросервісної архітектури, її переваги та недоліки

Мікросервіси - це один з архітектурних підходів до побудови розподілених програмних систем. Організації переходять на мікросервісну архітектуру, щоб досягти швидкого процесу релізу нових функцій, кращої відмовостійкості та більшої безпеки в міру збільшення масштабу своїх систем [10]. Однією з важливих відмінностей між мікросервісами та монолітною архітектурою є спосіб розбиття системи на менші функції, які працюють незалежно в обмеженому контексті. Це дозволяє розширювати та розгортати кожен мікросервіс, не впливаючи на інші запущені мікросервіси, що робить його більш гнучким і масштабованим рішенням.

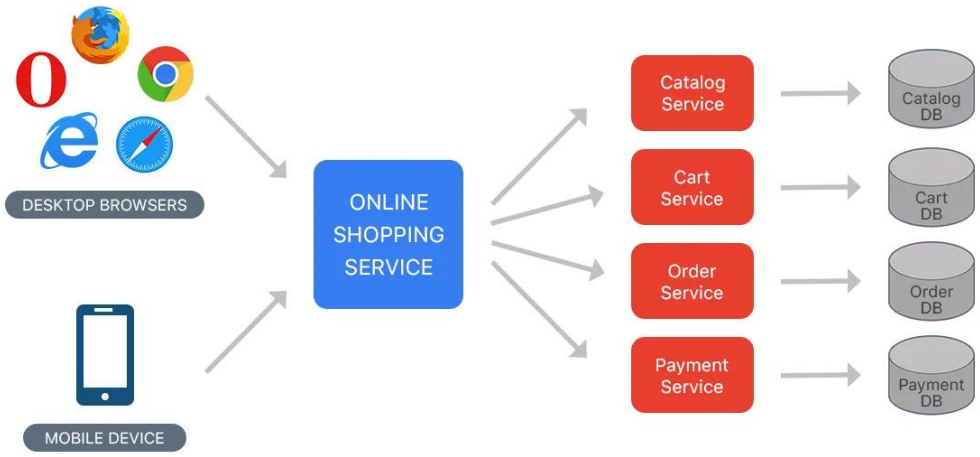

Рисунок 1.3 - Приклад мікросервісної архітектури [4]

Наразі не існує точного визначення терміну "мікросервіси", проте є загальні характеристики, якими повинно володіти будь-яке програмне забезпечення, щоб мати можливість претендувати на термін "мікросервіс". Ці характеристики включають, але не обмежуються наступними пунктами:

- розмір додатка. Цей показник є відносним поняттям і може змінюватися залежно від різних факторів. Дизайн сервісу може базуватися на бізнес-вимогах високого рівня, таких як автентифікація/авторизація клієнта, або на функціональному рівні, наприклад, на функціях входу/виходу з системи. Опитування різних команд у 40 компаніях [11] показало, що сервіси, які мають менше ніж 100 або понад 1000 рядків коду, використовуються рідко. Опитування також показало, що сервіси сфокусовані й звужені до виконання однієї конкретної задачі [11][12];
- обмежений контекстом. В архітектурі мікросервісів кожна послуга інкапсульована навколо певної бізнес-потреби. Згруповані разом, ці можливості утворюють домен [13]. Кожен інкапсульований сервіс працює тільки в межах свого домену та не піддається впливу решти системи. Це забезпечує кращу автономність, децентралізоване управління та простішу заміну сервісів [12]. Якщо сервісу потрібно спілкуватися за межами свого домену, він повинен використовувати явний інтерфейс, наприклад REST API [14].
- розподіленість. Розподілені системи не обмежуються мікросервісною архітектурою. Сервісно-орієнтована архітектура, яка використовується довше, також є розподіленою. Однак мікросервіси підняли цю характеристику на новий рівень, децентралізувавши дані між сервісами. Кожна бізнес-можливість та пов'язані з нею дані управляються незалежним сервісом, який може бути розгорнутий і запущений на власному хості, що робить його більш автономним, ніж SOA [15].

- незалежне розгортання. Кожен мікросервіс може бути розгорнутий незалежно від інших сервісів у будь-який час за допомогою неперервної доставки та інтеграції. Іншими словами, кожен мікросервіс може бути розгорнутий без узгодження з власниками інших сервісів [16].

Мікросервісна архітектура має ряд переваг які виділяють її серед інших підходів:

- системи, побудовані на архітектурі мікросервісів, складаються з незалежних сервісів, які взаємодіють один з одним через мережу. Ця особливість дозволяє використовувати різні мови програмування, фреймворк або моделювання даних для кожного сервісу, замість того, щоб мати універсальний стек технологій для побудови кожного компонента системи. У цій архітектурі розробники мають свободу використовувати технології, які найкраще підходять для виконання завдання та людей, які виконують це завдання [16];
- масштабованість є однією з найважливіших мотивацій для використання архітектури мікросервісів [16]. У монолітній архітектурі, коли виникає потреба в масштабуванні, все повинно масштабуватися разом, оскільки всі компоненти пов'язані між собою. Однак, мікросервісна архітектура дозволяє здійснювати цільове масштабування, що означає, що масштабуються лише ті компоненти, які мають навантаження. Наявність такої можливості масштабування є важливою для додатків, які працюють у хмарі, оскільки більшість хмарних провайдерів беруть з клієнтів плату за ресурси, які вони використовують. Для компаній було б дорого масштабувати всю програму в хмарі, якщо тільки одна частина програми має більше навантаження;
- архітектура мікросервісів може забезпечити вищу доступність програмного забезпечення завдяки можливості реплікації та розподілу між центрами обробки даних у різних географічних точках. Це забезпечує розподіл навантаження і відмовостійкість, на відміну від монолітної архітектури [15].

Крім того, мікросервіси пропонують простоту розгортання, що може ще більше підвищити доступність системи. Традиційно, будь-яка невелика зміна в системі вимагала вивільнення всієї системи. На відміну від цього, з мікросервісами окремі компоненти можна запускати у виробництво, не впливаючи на всю систему. Це призводить до швидшого розгортання зі зниженим ризиком внесення змін.

Мікросервіси, як і будь-яка інша програмна архітектура чи шаблон проєктування, не є панацеєю чи універсальним архітектурним рішенням для кожного програмного додатка. Однією з ключових проблем мікросервісів є складність, яку вони вносять у систему. Розробка, розгортання, тестування та налагодження мікросервісів часто є складнішими, ніж монолітні додатки через їх розподілену природу [17].

В архітектурі мікросервісів не існує конкретного визначення того, наскільки малим або великим повинен бути кожен сервіс. Через відсутність рекомендацій, якщо команда неправильно визначиться з розміром кожного мікросервісу, можна отримати розподілену, але сильно пов'язану систему, що суперечить меті мікросервісів [17]. Нарешті, додаткова затримка вноситься в додаток в результаті комунікації процесів в мікросервісах. Це пов'язано з тим, що один запит повинен пройти через декілька сервісів, щоб бути повністю обробленим та повернути відповідь.

1.4 Взаємодія між процесами у мікросерверній архітектурі

Рішення про те, як мікросервіси взаємодіють один з одним, є одним з найбільш важливих і фундаментальних рішень, які необхідно прийняти при впровадженні системи, заснованої на архітектурі мікросервісів [14]. Взаємодія між процесами (англ.

Inter-Process Communication, IPC) відіграє набагато важливішу роль в архітектурі мікросервісів, ніж у монолітних системах [17]. Причина такої важливості IPC в мікросервісах полягає в тому, що, на відміну від монолітної архітектури, модулі не можуть отримувати доступ та викликати один одного на рівні мови. Натомість мікросервіси структурують систему як набір незалежних розподілених сервісів. Це означає, що кожен сервіс потенційно може працювати на іншому хості, ніж інший.

При впровадженні IPC в архітектуру мікросервісів необхідно враховувати декілька факторів. Наприклад, вибір протоколу обміну повідомленнями, формату серіалізації та транспортного протоколу може суттєво вплинути на продуктивність та масштабованість системи. Крім того, важливо зважити компроміс між простотою та гнучкістю. Загалом, хоча рішення про те, як мікросервіси взаємодіють один з одним, може здатися технічною деталлю, насправді це критично важливе рішення, яке може суттєво вплинути на успіх системи, заснованої на мікросервісах.

Для IPC можуть використовуватися різні протоколи та фреймворки мережевого зв’язку. Розглянемо декілька загальновживаних, зокрема:

- віддалений виклик процедур, рідше виклик віддалених процедур (анг. remote procedure call, RPC) - протокол, який дозволяє одній комп'ютерній програмі виконувати код на іншому комп'ютері без необхідності програміста явно кодувати для зв'язку між ними. Іншими словами, RPC дозволяє програмі викликати функцію на віддаленому комп'ютері, як якщо б це був локальний виклик функції. [18]. Він використовується для забезпечення розподілених обчислень і може застосовуватися в різних контекстах, включаючи клієнт-серверні програми та веб сервіси;
- протокол обміну структурованими повідомленнями (англ. simple object access protocol, SOAP) - протокол обміну структурованими повідомленнями в розподілених обчислювальних системах, що базується на форматі XML (скор.

анг. extensible markup language) [8]. Може використовуватися в різних транспортних протоколах, включаючи HTTP;

- передача репрезентативного стану або REST - це архітектурний стиль для розробки програмного забезпечення, який використовується для створення вебсервісів та API (скор. англ. Application Programming Interface) [19]. У цьому стилі архітектури інформація зберігається на сервері, а клієнти можуть отримувати цю інформацію, звертаючись до сервера через HTTP-запити;
- протокол MQTT (скор. англ. message queue telemetry transport) - це легкий протокол обміну повідомленнями, розроблений для використання в інтернеті речей та інших середовищах з обмеженими ресурсами [20]. Він використовує модель публікації/підписки для зв'язку і часто використовується в системах, де пристроям потрібно надсилати та отримувати невеликі обсяги даних через ненадійні мережі;
- віддалений виклик методів (анг. remote method invocation, RMI) - це протокол, що дозволяє об’єкту, що працює на одній віртуальній машині Java, викликати методи об’єкта, що працює на іншій віртуальній машині Java. RMI забезпечує віддалений зв'язок між програмами, написаними мовою програмування Java [21];
- gRPC - це високопродуктивний фреймворк для віддаленого виклику процедур, розроблений компанією Google. Він використовує формат серіалізації даних Protocol Buffers і забезпечує підтримку потокового, двонаправленого зв'язку та балансування навантаження [22]. Є частною реалізацією протоколу RPC.

Ці протоколи та фреймворки можна використовувати в різних контекстах, включаючи веб-сервіси, інтернет речей, розподілені системи тощо. Вибір методу між процесної комунікації залежать від багатьох факторів. Як правило єдиною правильної відповіді може не бути. Важливо ретельно оцінити ваші конкретні вимоги та

обмеження, щоб визначити, який метод найкраще підійде для кожного конкретного додатка.

На сьогодні в галузі прийнято вважати архітектуру REST з використанням текстового протоколу JSON як стандарт для взаємодії у мікросервісній архітектурі. Відносно нова технологія gRPC починає заміщувати REST архітектуру, особливо в великих компаніях як Google, Netflix, Spotify та інші. Дане дослідження сфокусовано на порівнянні двох фреймворків: REST та gRPC.

1.5 Огляд існуючих досліджень

Цей розділ має на меті надати огляд досліджень, які були проведені навколо архітектури мікросервісів та зокрема щодо порівняння ефективності механізмів пов'язаних з методами серіалізації даних, які можуть бути застосовані при реалізації між процесної взаємодії для систем на основі мікросервісів.

У дослідженні проведеному групою дослідників з IBM [23], метою було розробити інфраструктуру, оптимізовану для систем побудованих за допомогою мікросервісної архітектури. Команда побудувала дві версії зразка програми - одну на основі монолітної архітектури, а іншу на основі мікросервісів. Команда виявила значні накладні витрати на продуктивність та більше споживання апаратних ресурсів у версії системи з мікросервісами порівняно з монолітною версією. У статті відзначено поганий дизайн взаємодії процесів в мікросервісній системі як одну з причин значного погіршення продуктивності, а отже, розкрито потенціал для подальших досліджень.

Що стосується формату обміну повідомленнями та серіалізації даних, то в роботі [24] автори порівняли два протоколи - REST API та gRPC для зв'язку на рівні

додатків в програмно-конфігурованих мережах (SDN). Хоча порівняння проводилося для SDN, однак, результати дослідження можуть бути застосовні для реалізації між процесної взаємодії в системах побудованих на архітектурі мікросервісів.

Дослідження надало перевагу gRPC над REST та аргументувало це тим, що погіршення мережевої затримки може мати більший негативний вплив, коли комунікація відбувається за допомогою транспортного протоколу HTTP/1.1, у порівняні з gRPC, який працює на версії HTTP/2.0. Крім того, в документі зазначено що gRPC пропонує кращу безпеку зв'язку шляхом використання TLS через створення захищеного каналу в HTTP/2.0.

Інші дослідження також сходяться на твердженні що gRPC є набагато швидшим у порівняні з REST.

Результати дослідження порівняння ефективності gRPC та REST [25] показують, що лише в одному сценарії REST показав кращий результат - отримання великого за розміром повідомлення. Але в середньому gRPC приблизно в 7 разів швидший за REST при отриманні даних і приблизно в 10 разів швидший при надсиланні даних.

Друге дослідження [26], присвячене одночасній передачі даних, показує, що gRPC здатний робити 48 запитів за секунду (1 запит за 20,81 мс), в той час, як REST - лише 4 запити в секунду (1 запит за 260 мс). Результати показують, що gRPC має приблизно в 10 разів вищу пропускну здатність, ніж REST. Крім того, gRPC використовує набагато менше процесорного часу для обробки кожного повідомлення. У gRPC загальний процесорний час становив 832.00 мс/с на 48 запитів які було зроблено в секунду, що становить 17.33 мс/с на 1 запит. У випадку REST-зв'язку процесорний час становив 404,00 мс/с що становить 101 мс/с на один запит.

Останнє згадане дослідження [27] порівнювало gRPC і REST з різними типами навантаження. Дослідження показало очікуваний результат, що у випадку паралельних запитів до сервера з невеликим або нормальним навантаженням,

відбувається швидше за допомогою REST (1 запит за 3,023 мс), ніж gRPC (1 запит за 4,196 мс). Велике навантаження було приблизно в 1,5 раза швидшим при застосуванні gRPC (1 запит за 2,579 мс), ніж у REST (1 запит за 4,0195 мс).

У висновки можна зазначити що результати всіх досліджень підтверджують що gRPC-з'єднання є швидшим за REST-з'єднання. Крім того, було задекларовано вищу пропускну здатність протоколу gRPC.

2 ПОРІВНЯННЯ ФРЕЙМВОРКІВ REST ТА GRPC

2.1 Основні поняття та визначення REST API

REST - це архітектурний стиль, який використовується для побудови API на основі протоколу HTTP. Він використовується для розробки вебсервісів та комунікації в Інтернеті або хмарних обчисленнях. REST дозволяє розробникам створювати масштабовані та ефективні програмні додатки, що робить їх цінним інструментом для широкого спектра цілей. Багато великих технологічних компаній, включаючи Google, Facebook та Amazon, використовують REST API у своїх сервісах. Фреймворк став популярним рішенням для хмарних обчислень, надаючи можливість надавати послуги та спілкуватися з клієнтами. Більшість популярних мов програмування, таких як Java, Go, Python та JavaScript, дозволяють розробникам створювати REST API. Ресурси, доступні та модифіковані за допомогою REST API, називаються "ресурсами". Ці ресурси можуть бути отримані та змінені за допомогою існуючих методів HTTP. Нижче наведені найбільш поширені методи, що використовуються в REST:

- POST для створення нового ресурсу;
- GET для отримання існуючого ресурсу;
- PUT для оновлення існуючого ресурсу;
- DELETE для видалення існуючого ресурсу.

Існує ще кілька методів, що використовуються в REST API, таких як HEAD, OPTION або PATCH, але вони не так часто використовуються, як чотири методи, описані вище. REST підтримує декілька форматів даних для представлення ресурсів (JSON, XML, форми). Однак JSON є найбільш часто використовуваним форматом завдяки своїй простоті та легкій структурі. Порівняно з XML, JSON легше читати та розуміти, оскільки він не містить великої кількості конструкторів і повторюваних

компонентів. Крім того, його корисне навантаження набагато менше, що призводить до меншого використання пропускної здатності та швидшого та ефективнішого зв'язку через REST API, на відміну від SOAP, який базується виключно на форматі XML.

REST базується на архітектурі клієнт-сервер, що означає, що в комунікації беруть участь два різних компоненти: клієнт і сервер.

Клієнт - це компонент, який ініціює запит і отримує відповідь від сервера. Він відповідає за створення і надсилання HTTP-запитів на сервер, які містять інформацію про ресурс, до якого клієнт хоче отримати доступ, наприклад, URL-адресу, метод HTTP і будь-які параметри або заголовки. Клієнтом може бути веббраузер, мобільний додаток або будь-яка інша програма, якій потрібен доступ до ресурсів з сервера.

Сервер - це компонент, який отримує запит від клієнта, обробляє його і надсилає відповідь назад клієнту. Він відповідає за управління ресурсами, до яких клієнт хоче отримати доступ, наприклад, даними в базі даних, файлами у файловій системі або іншими вебсервісами. Сервер повинен вміти розуміти та інтерпретувати HTTP-запити, надіслані клієнтом, і повертати відповідну відповідь.

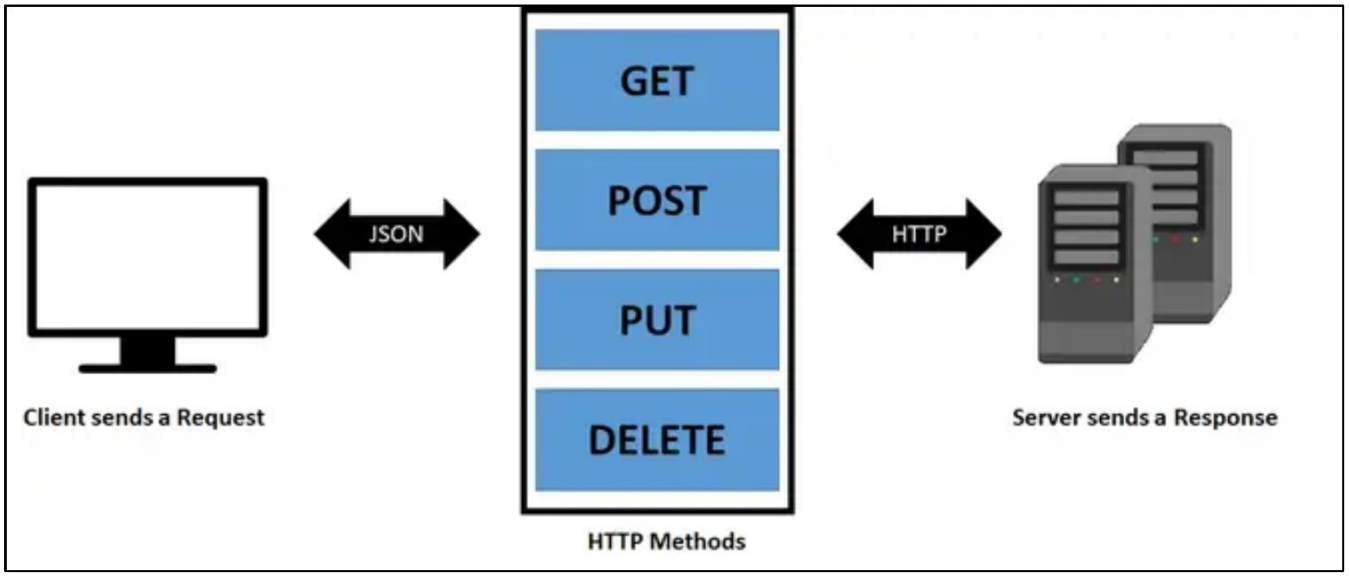

Рисунок 2.1 – Реалізація механізму REST [28]

Клієнт-серверна архітектура REST дозволяє чітко розділити обов'язки між клієнтом і сервером, що може покращити масштабованість вебсистем та полегшити їх підтримку. Клієнти можуть бути розроблені з використанням будь-якої мови програмування або технології, якщо вони дотримуються обмежень архітектурного стилю REST, а сервери можуть бути розроблені незалежно від клієнтів, яких вони обслуговують.

Одним з ключових принципів REST є відсутність стану, що означає, що сервер не зберігає жодної інформації про стан клієнта або контекст між запитами. Кожен запит від клієнта повинен містити всю необхідну інформацію для виконання запиту сервером. Такий підхід до комунікації забезпечує масштабованість, надійність і простоту проєктування вебсистем.

Відсутність стану REST-серверів також означає, що вони не прив'язані до конкретного клієнта або сесії, що дозволяє підвищити продуктивність і зменшити складність сервера. Сервер може зосередитися на негайному запиті та не турбуватися про збереження будь-якої інформації про попередні запити або сесії клієнта. Це може полегшити масштабування REST вебсервісів шляхом додавання більшої кількості серверів для обробки запитів у міру зростання навантаження.

Однак відсутність стану також може створювати певні проблеми, особливо коли мова йде про безпеку. Оскільки сервер не зберігає жодної інформації про стан або контекст клієнта, може бути складніше реалізувати механізми безпеки, такі як автентифікація та контроль доступу. Вебсервіси повинні покладатися на інші механізми, такі як токени та файли cookie, щоб підтримувати безпеку між запитами.

Хоча REST є широко використовуваним і популярним архітектурним стилем, він також має свої проблеми та недоліки:

- відсутність стандартизації: Хоча архітектурний стиль REST чітко визначений, не існує стандартних специфікацій або протоколів для реалізації REST-орієнтованих вебсервісів;

- складність: REST може бути складнішим у реалізації, ніж інші архітектури вебсервісів, особливо для розробників, які не знайомі з обмеженнями архітектурного стилю REST;
- безпека: Оскільки відсутність стану є одним з головних принципів REST може бути складно реалізувати механізми безпеки, такі як управління сеансами та автентифікація. Це може зробити його більш вразливим до таких атак, як міжсайтова підробка запиту (скор. англ. CSRF) та міжсайтовий скриптинг (скор. англ. XSS);
- накладні витрати на продуктивність: Хоча REST розроблений як легкий, він все одно може спричинити певні накладні витрати на продуктивність через необхідність передачі даних між клієнтом і сервером у кожному запиті;
- обмежена функціональність: REST в першу чергу призначений для CRUD-операцій (створення, читання, оновлення, видалення) над ресурсами, а це означає, що він може не підходити для складніших операцій або транзакцій;
- керування версіями: Оскільки REST вебсервіси з часом розвиваються, може бути важко керувати змінами в API та підтримувати зворотну сумісність, особливо якщо різні клієнти можуть використовувати різні версії API.

Загалом, хоча REST є потужним і гнучким архітектурним стилем, він може не підходити для всіх випадків використання і може вимагати ретельного розгляду і планування для подолання його проблем і обмежень.

2.2 Основні поняття та визначення gRPC

Віддалений виклик процедур або RPC - це протокол, який дозволяє комп'ютерній програмі виконувати підпрограму або процедуру на віддаленій системі.

Вперше RPC був представлений у 1984 році Ендрю Біреллом та Брюсом Нельсоном з Digital Equipment Corporation (DEC) [29] як засіб для забезпечення між процесного зв'язку між різними процесами, що виконуються на різних комп'ютерах.

RPC зазвичай має клієнтську програму, яка робить запит до серверної програми, яка потім обробляє запит і повертає відповідь клієнту. Зв'язок між клієнтом і сервером зазвичай забезпечується за допомогою мережевого протоколу, такого як TCP/IP.

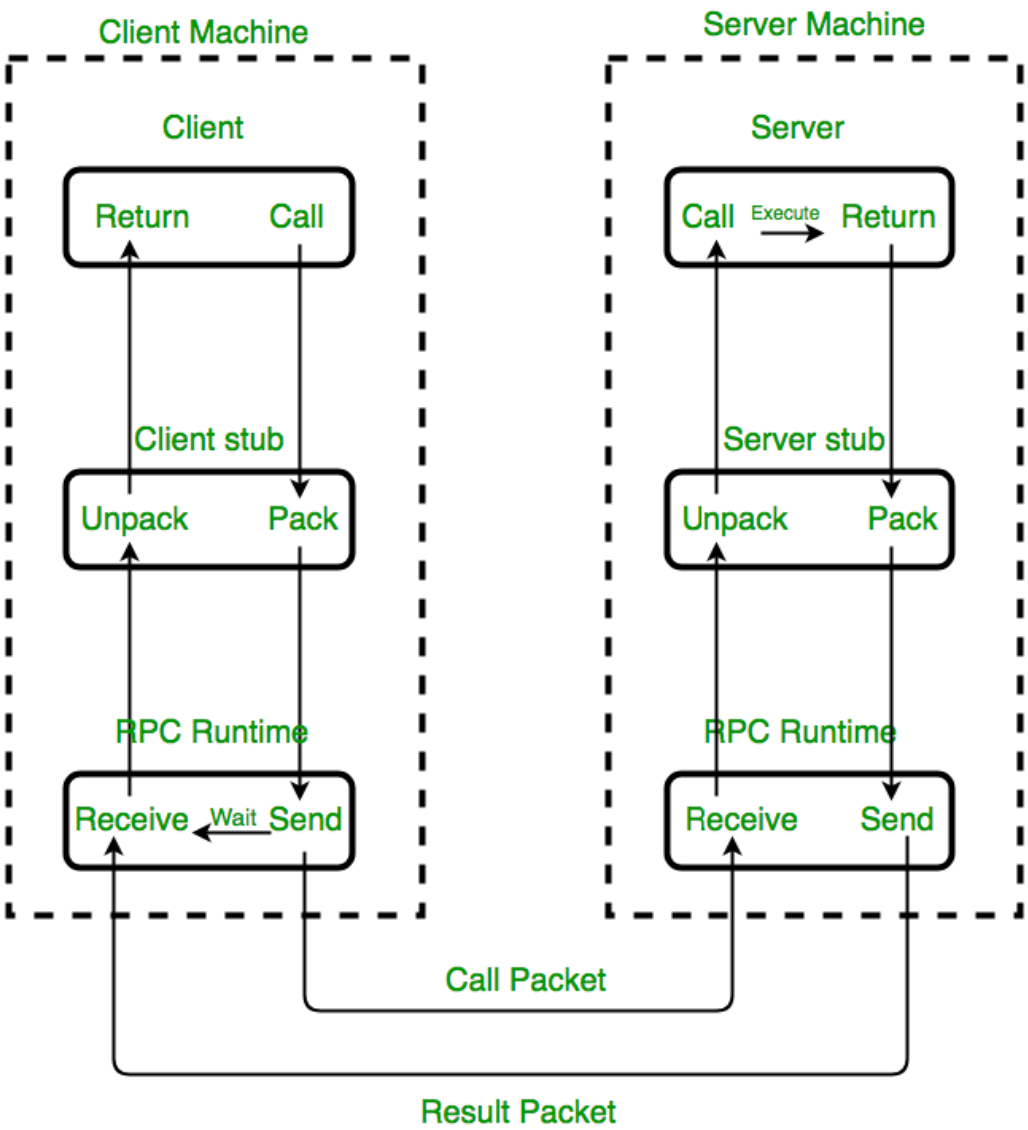

Рисунок 2.2 – Реалізація механізму RPC [30]

Рисунок 2.2 ілюструє реалізацію механізму RPC. Клієнтська програма викликає метод RPC, наданий сервером у клієнтській заглушці, з усіма необхідними параметрами. Заглушка аналізує параметри, упаковує ім'я методу та параметри в повідомлення запиту і надсилає його до виконуваного модуля RPC. Модуль обробляє мережеву взаємодію і відправляє повідомлення клієнта на сервер, де виконуваний модуль RPC сервера приймає його. Повідомлення розпаковується серверною заглушкою і надсилається серверному додатку для обробки з усіма вхідними параметрами із запиту. Сервер обробляє запит і готує повідомлення-відповідь, яке надсилається назад клієнту тим же шляхом, яким надійшло повідомлення від клієнта.

Фреймворк gRPC - це сучасна, високопродуктивна реалізація віддаленого виклику процедур (RPC), розроблена компанією Google як проєкт з відкритим вихідним кодом. Він дозволяє розробникам легко створювати ефективний та масштабований зв'язок між розподіленими системами. gRPC використовує буфери протоколів Google (Protobuf) як формат серіалізації даних за замовчуванням, що забезпечує ефективне кодування та декодування структурованих даних.

Однією з ключових особливостей gRPC є його здатність генерувати клієнтський код на різних мовах, таких як C++, Java, Python та Go. Це дозволяє розробникам легко створювати клієнтські та серверні додатки на обраній ними мові, використовуючи при цьому переваги ефективного комунікаційного протоколу gRPC.

Фреймворк також підтримує двонапрямлену потокову передачу, що дозволяє здійснювати зв'язок між клієнтськими та серверними додатками в режимі реального часу. Крім того, він надає перехоплювачі - функції проміжного програмного забезпечення, які можуть перехоплювати та змінювати повідомлення gRPC для таких цілей, як автентифікація, ведення журналів або виконання користувацьких функцій.

Компілятор буферного протоколу gRPC генерує весь необхідний код не тільки на стороні сервера, але і на стороні клієнта. Сервер реалізує процедури, оголошені вхідними та вихідними параметрами, визначеними повідомленнями в proto файлі, які

віддалено викликаються клієнтами. Сервер gRPC декодує вхідні виклики й виконує код віддаленої процедури на стороні сервера. Після завершення обробки сервер кодує повідомлення-відповідь і надсилає його клієнту.

Щоб відкрити RPC-з'єднання, клієнтам необхідні локальні файли, створені gRPC-сервером, які називаються заглушками. Ці заглушки забезпечують специфічний для мови програмування інтерфейс, який додає процедури, надані сервером, до коду. Ці локальні файли дозволяють викликати віддалені процедури безпосередньо з коду, виглядаючи як ще одна локальна функція. Клієнтам потрібно лише підготувати вхідні параметри для віддаленого виклику. Заглушка кодує параметри зі специфічних для мови об'єктів у відповідне повідомлення protobuf. Сервер отримує повідомлення, обробляє запит і надсилає відповідь. Вони декодуються клієнтською заглушкою у відповідний об'єкт мовою програмування клієнта [30].

Загалом, gRPC - це потужний і гнучкий фреймворк RPC, який забезпечує сучасний і ефективний спосіб побудови розподілених систем. Підтримка Proto визначень сервісів, заглушок на стороні клієнта, двонапрямний потік та перехоплювачів робить його популярним вибором для побудови мікросервісів, хмарних додатків та інших розподілених систем.

Фреймоворк gRPC підтримує чотири типи зв'язку - унарний, серверну потокову передачу, клієнтську потокову передачу та двонапрямлену потокову передачу.

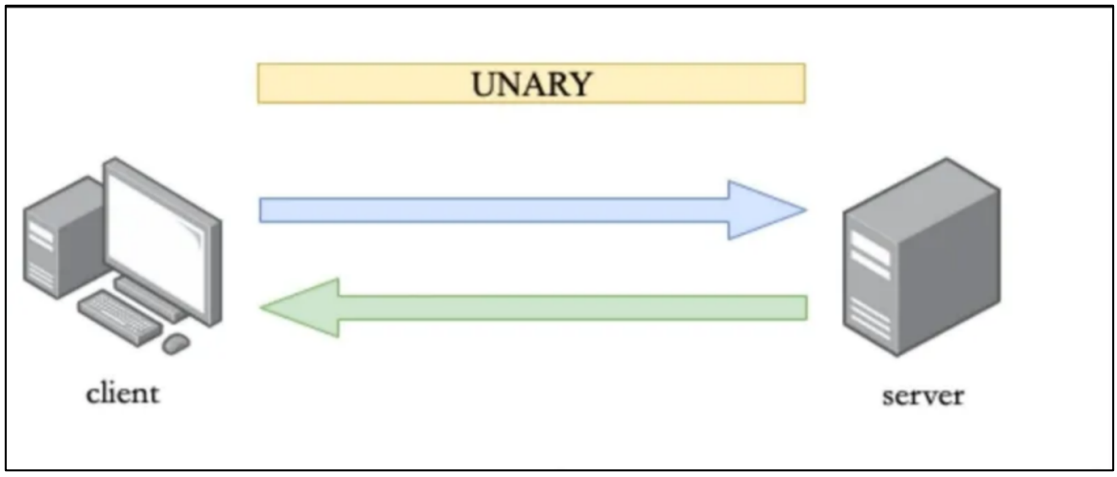

Рисунок 2.3 – Унарний тип зв’язку [31]

Унарні RPC-виклики - це базовий тип зв'язку побудований за схемою запит/відповідь як і REST API. Клієнт надсилає одне повідомлення на сервер і отримує одну відповідь від сервера. Доцільно використовувати цей тип зв'язку при передачі невеликих та конкретних повідомлень. Рекомендується починати створювати систему використовуючи унарний тип та переходити до складніших потокових зв'язків при проблемах з продуктивністю.

Потокова передача даних це новий тип зв’язку для віддалених викликів методів який став можливим завдяки впровадженню HTTP/2. При серверній потоковій передачі даних клієнт надсилає одне повідомлення на сервер та починає отримувати нескінченну кількість відповідей від сервера до тих пір, доки одна зі сторін не завершить з’єднання. Цей тип зв’язку підходить до ситуацій коли серверу потрібно відправити клієнту великий фрагмент даних або в ситуації, коли серверу потрібно передати дані до клієнта без додаткових запитів з його сторони, наприклад, чати або стрічки новин.

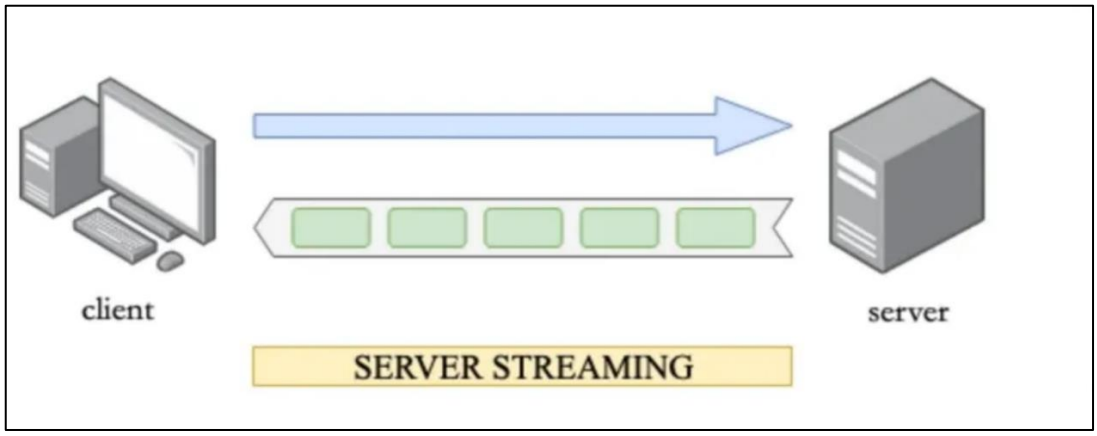

Рисунок 2.4 – Серверна потокова передача даних [31]

Потокова передача даних від клієнта працює аналогічно до серверної, але з протилежним направленням даних. Клієнт надсилає до сервера нескінчену кількість повідомлень та отримує одну або зовсім не отримує відповідь від сервера. Цей тип зв’язку підходить до ситуацій коли клієнту потрібно відправити велику кількість

даних до сервера не очікуючи відповіді або коли нема потреби відповідати на кожне повідомлення, наприклад, обмін повідомленнями в інтернеті речей.

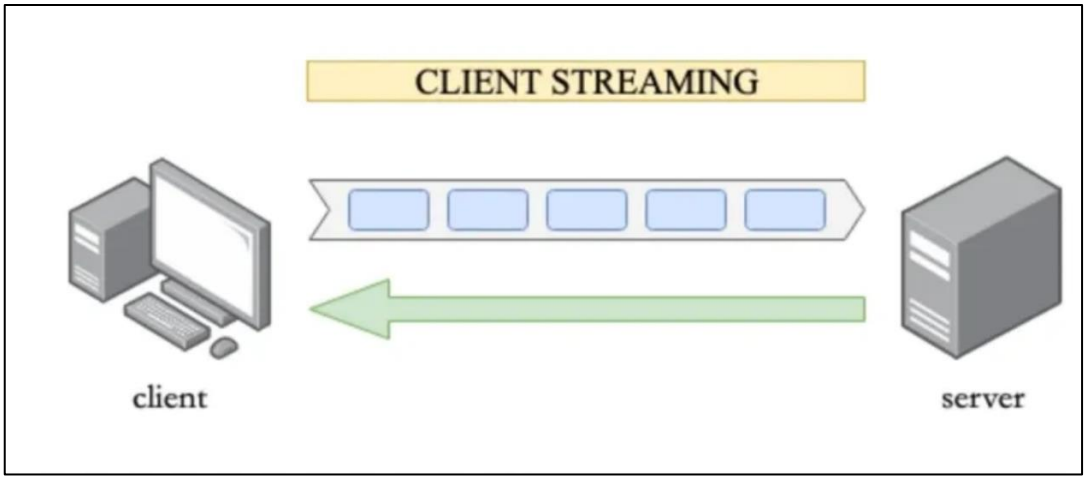

Рисунок 2.5 – Клієнтська потокова передача даних [31]

Останній тип зв’язку є комбінацією серверної та клієнтської потокової передачі утворюючи двонапрямний потік даних. Застосовуються коли двом сторонам зв’язку необхідно передавати велику за обсягом інформацію без попередніх викликів.

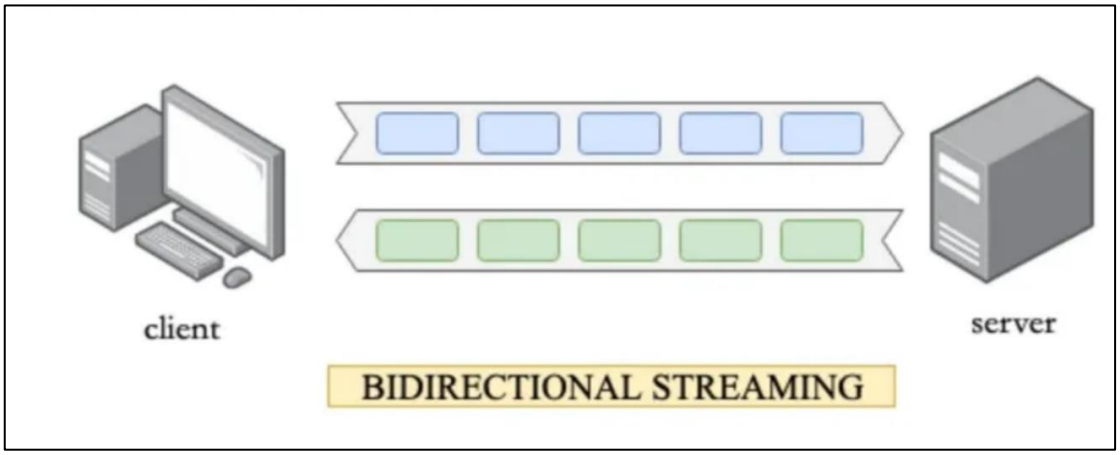

Рисунок 2.6 – Двонаправлена потокова передача даних [31]

Фреймворк gRPC розроблений як технологія між процесної взаємодії в масштабі Інтернету, яка може подолати більшість недоліків традиційних технологій між процесної взаємодії. До основних переваг gRPC відносять:

- ефективність між процесної взаємодії: Замість використання текстових форматів, таких як JSON або XML, використовується двійковий протокол Protobuf для зв’язку зі службами та клієнтами gRPC. Крім того, фреймворк використовує переваги HTTP/2 що робить його однією з найефективніших технологій між процесної взаємодії;
- контрактний підхід: Фреймворк опирається на контрактний підхід при розробці додатків. Спочатку визначаються інтерфейси сервісів, а потім додаються деталі реалізації. Контрактний підхід є простим, послідовним, надійним та масштабованим досвідом розробки додатків. Наявність контракту програмного інтерфейсу дозволяє клієнту розпочати розробку на своїй стороні не очікуючи повної готовності реалізації на сервері прискорюючи випуск нових функцій та можливостей;
- строго типізовані типи повідомлень: Оскільки gRPC використовує Protobuf як мову опису інтерфейсів, контракти та типи повідомлень чітко визначені. Це робить розробку розподілених додатків набагато стабільнішою, оскільки статична типізація допомагає подолати більшість помилок виконання та сумісності, з якими ви можете зіткнутися при створенні хмарних додатків, що охоплюють декілька команд та технологій;
- підтримка потокової передачі даних: Можливість створювати звичайні повідомлення в стилі "запит-відповідь", а також клієнтські та серверні потоки є ключовою перевагою над звичайним REST стилем обміну повідомленнями. Потокова передача вбудована в фреймворк та не потребує додаткових зусиль від розробників;
- наявність вбудованих допоміжних функцій: Фреймворк пропонує вбудовану підтримку стандартних функцій, таких як автентифікація, шифрування, відмовостійкість (дедлайни і таймаути), обмін метаданими, стиснення, балансування навантаження, виявлення сервісів та інші.

Як будь яка інша технологія gRPC має ряд недоліків:

- непідходить до сервісів з зовнішними клієнтами: Коли ви хочете надати додаток або послуги зовнішньому клієнту через Інтернет, gRPC може виявитися не найкращим протоколом. Контрактна, сильно типізована природа послуг gRPC може перешкоджати гнучкості послуг, які ви надаєте зовнішнім сторонам, і споживачі отримують набагато менше контролю (на відміну від таких протоколів, як GraphQL). Шлюз gRPC розроблений як обхідний шлях для подолання цієї проблеми;
- радикальні зміни в програмному інтерфейсі потребують регенерації коду: Модифікації схеми є досить поширеним явищем у сучасних сценаріях використання між сервісної комунікації. Коли відбуваються радикальні зміни у визначенні сервісу gRPC, зазвичай нам потрібно регенерувати код як для клієнта, так і для сервера. Це має бути включено в наявний процес безперервної інтеграції й може ускладнити загальний життєвий цикл розробки;
- новизна технології: gRPC все ще є відносно новою технологією. Навіть найкращі технології спочатку ставляться під сумнів, оскільки вони невідомі переважній більшості розробників. Особливо, якщо існують робочі альтернативи, такі як REST, які перевірені спільнотою протягом багатьох років. Також, підтримка gRPC в браузерах і мобільних додатках все ще знаходиться на примітивних стадіях;
- відсутність підтримки розробників: Як було зазначено в попередньому пункті, gRPC все ще є відносно новою технологією. Це є причиною того, що розробники стикаються з нестачею інструментів для. Багато інструментів побудовано на основі HTTP/1.1, тому вони непридатні для роботи з gRPC. Однак справа не тільки в інструментах, а й у відсутності належної документації, навчальних посібників і невирішених проблемах.

Отже, gRPC не є технологією, яку слід використовувати для всіх випадків між процесної комунікації. Потрібно оцінити сценарій використання та вимоги бізнесу і вибрати відповідний протокол обміну повідомленнями

2.3 Порівняння REST API та gRPC

Обидві технології надають сервісам можливість взаємодіяти один з одним для обміну даними. Попри те, що обидві технології мають однакову мету, способи її реалізації відрізняються. REST - це стандарт, де всі правила не є обов'язковими. Розробники самі вирішують, які функції REST вони хочуть використовувати в своїх додатках. При побудові додатків за допомогою REST розробники можуть використовувати безліч мов програмування, фреймворків, бібліотек і протоколів на вибір, які були створені спільнотою за всі роки використання архітектури. Реалізувати REST API можна навіть на асемблері. Фреймворк gRPC є більш вимогливим до реалізацій. Він не є стандартом, як REST, а більше схожий на протокол, і зазвичай існує лише один підхід як реалізувати певну функціональність.

REST - це ресурсно-орієнтований стандарт. Він надає спосіб створити, отримати, оновити або видалити ресурс. Наприклад, створити новий банківський рахунок, показати депозит, додати кошти на рахунок або видалити рахунок. Зазвичай він реалізується як комбінація URL-адреси, що представляє місце знаходження ресурсу, і відповідного HTTP-методу (GET, POST, PUT, DELETE). gRPC зосереджений на процедурах, а не на ресурсах. Не кожен раз, коли сервіси взаємодіють один з одним один з одним потрібно ділитися будь-яким ресурсом. Іноді одному сервісу просто потрібно повідомити інший сервіс про якусь конкретну дію.

Схема API в REST є абсолютно необов'язковою. Дуже часто зустрічається ситуація, коли API не мають схеми та документації. Це часто призводить до неочікуваної поведінки при використанні систем, а інтеграція до таких API може бути складним завданням для розробників. Їм потрібно з'ясувати, якого типу відповіді очікувати, розглянути всі поля як необов'язкові та реалізувати багато непотрібної логіку синтаксичного аналізу в клієнті API, щоб запобігти помилкам. Завдяки Protobuf, схема API в gRPC є чіткою, однозначною та вимагає від розробників сурова дотримуватись визначених типів. Повідомлення запиту і відповіді визначені в proto файлах. Без цих визначень неможливо викликати процедури. Клієнт знає, як саме має виглядати повідомлення запиту, а також чого очікувати у відповіді. Крім того, компілятор protoc генерує весь необхідний специфічний для мови код для клієнта та сервіса, тому немає потреби реалізовувати будь-яку додаткову логіку та адаптерів навколо цих викликів процедур. Таким чином, розробка стає швидша та менш вразлива до помилок. Існує можливість примусового використання суворої схеми API в REST. Існують популярні технології, такі як OpenAPI, в яких розробники можуть визначати схему та документацію. Однак, це не є обов'язковою частиною REST-реалізації.

Вся ідея між процесної взаємодії полягає в тому, щоб дозволити мікросервісам обмінюватися повідомленнями між собою. Повідомлення містять інформацію, яка необхідна мікросервісам для обробки їхніх бізнес-вимог. Окрім вибору методу між процесної взаємодії, ще одним важливим рішенням є вибір формату повідомлень, який повинні використовувати сервіси при спілкуванні один з одним. Вибір формату повідомлень може вплинути на ефективність взаємодії, зручність використання API та її спроможність до змін [17].

Мікросервіси можуть обмінюватися повідомленнями у текстовому форматі, такими як JSON або XML. Обидва формати певною мірою пропонують однакові функції, такі як портативність, безпека, безпека, читабельність для людини тощо. Крім

того, сервіси можуть використовувати двійкові формати, такі як Google Protocol Buffer або Apache Avro, для обміну повідомленнями між собою, що може забезпечити вищу ефективність і меншу надмірність метаданих [32].

JSON було обрано замість XML, оскільки останній втратив свою популярність за останні роки. Більшість сучасних систем більше не покладаються на XML для обміну інформацією між собою [32]. JSON пропонує набагато швидший та ефективніший обмін даними, будучи при цьому значно менш багатослівним та маючи менший розмір ніж XML.

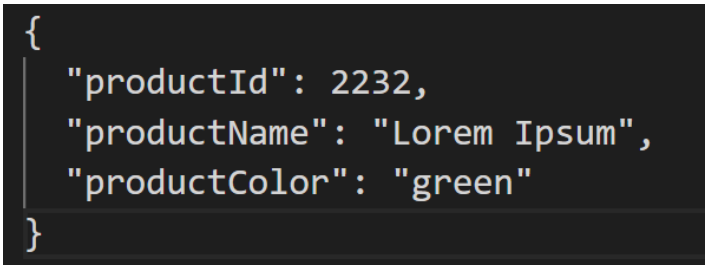

Рисунок 2.7 - Приклад повідомлення у форматі JSON

Хоча JSON пропонує значно вищу продуктивність, ніж інші текстові формати повідомлень, однак, існують аргументи проти нього. Йому не вистачає підтримки простору імен та перевірки вхідних даних. Ці функції повинні бути виконані на клієнті або сервері під час обробки повідомлення. Також варто зазначати основний недолік всіх текстових форматів – вони передаються мережею як звичайні текстові дані що мають більшу вагу та потребують більше ресурсів для опрацювання у порівнянні з двійковими форматами.

Protobuf - це багатоплатформовий двійковий формат повідомлень для серіалізації структурованих даних розроблений компанією Google. Цей формат використовується за замовчуванням для протоколу gRPC. Це швидкий і компактний формат, який підтримується різними мовами програмування, такими як C++, Java, C#, Python та Javascript.

Кожна властивість, подана в повідомленні Protobuf, має код типу з унікальним номером. Одержувач повідомлення може витягувати потрібні йому поля, ігноруючи нерозпізнані поля. Ця функція дозволяє API, які взаємодіють за допомогою gRPC, розширюватися, залишаючись при цьому сумісними з попередніми версіями [11].

Основною перевагою двійкових форматів над текстовими виступає менший розмір повідомлення для однакового набору даних.

```
message Product {
    required int32 productId = 1;
    required string productName = 2;
    optional string productColor = 3;
}
```

Рисунок 2.8 - Приклад схеми повідомлення у форматі Protobuf

Обидва протоколи REST та gRPC використовують транспортний протокол HTTP для передачі повідомлень. Але якщо REST використовує версію HTTP/1.1, то gRPC використовує більш новітню версію HTTP/2.

Протокол HTTP/1.1 був вперше випущений в 1977 році. Для кожного запиту від клієнта HTTP/1.1 відкриває нове TCP-з'єднання з сервером. С учасні браузери мають можливість відкривати декілька паралельних TCP-з'єднань використовуючи обхідні механізми, але це не є достатньо ефективним для великих додатків [33].

HTTP/1.1 не стискає заголовки й працює тільки з механізмом запит/відповідь. Сучасні вебдодатки в середньому завантажують до сотні ресурсів на сторінку, кожен з яких містить власні заголовки. Ці обмеження збільшують розмір мережевих пакетів та знижують загальну ефективність системи.

HTTP/2 - це новітніший стандарт інтернет-зв'язку випущений у 2015 році, який безпосередньо усуває деякі недоліки традиційного HTTP/1.1. Більшість браузерів вже повністю або частково підтримують його.

Нова версія розв'язує проблеми попередньої за допомогою мультиплексування і розпаралелювання потоків та стисненню HTTP-заголовків. Мультиплексування - це метод, за допомогою якого HTTP/2 дозволяє кілька запитів між клієнтом і сервером для кожного TCP-з'єднання.

Популярність REST не викликає сумнівів. Завдяки цьому існує безліч інструментів, фреймворків та бібліотек які допомагають з продуктивністю, документацією або просто полегшують розробку. Інструменти, доступні для gRPC, наразі дуже обмежені що принижує швидкість впровадження технології в реальні системи. Підсумкове порівняння обох технологій наведено у таблиці 2.1.

Таблиця 2.1 - Підсумкове порівняння REST та gRPC

Характеристика	REST API	gRPC
Контракт	За бажанням	Обов’язково (.proto)
Протокол коммунікації	HTTP/1.1	HTTP/2
Формат повідомлень	JSON	Protobuf
Сериалізація даних	Повільніша	Швидша
Специфікація	Вільна	Сувора
Форма комунікації	Запит/відповідь	Унарна або потокова
Потокова передача даних	Ні	Так, двоспрямований
Підтримка браузерів	Так	Ні, потребує додаткових зусиль
Безпека	TLS/SSL	TLS/SSL/ALTS
Генерація коду	Так, з використанням сторонніх інструментів	Так, вбудована у фреймворк
Кінцеві точки	Традиційний REST	Будь яка
Мультиплатформеність	Так	Так
Залежність клієнту від сервера	Ні	Так, потребує .proto файлів
Продуктивність	Повільніша	Швидша
Реалізація	Швидша	Повільніша

В другому розділі було розглянуто основні переваги та недоліки обох протоколів та проведене їх теоретичне порівняння. За результатами дослідження можна зробити висновки щодо ефективності gRPC у порівнянні з REST API. Ця перевага досягається використанням протоколу HTTP/2 що вирішує ряд проблем з ефективністю попередніх версій що використовуються в REST. Також на ефективність впливає використання двійкового формату Protobuf, тому що він компактніший за JSON. Слід зазначити що JSON також можна компресувати для зменшення розміру, але тоді втрачається одна з основних його переваг – читабельність для користувача. Протокол gRPC також підтримує одно та двоспрямовану передачу даних що може значно вплинути на ефективність систем в деяких варіантів використання.

Одне з ключових відмінностей полягає у визначенні зв’язку. Функції gRPC визначені суворим контрактом зі строго закодованими повідомленнями що обмежені визначеним API, а не самою архітектурою як у REST.

Однак є ще одне сфера, де REST має перевагу - підтримка браузерами. REST повністю підтримується всіма браузерами, коли використання gRPC обмежене і вимагає gRPC-web з проксі-рівнем для конвертації в різні версії HTTP, оскільки gRPC використовує функції HTTP/2.

3 РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ ТА ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

3.1 Формування вимог до програмного додатку

Тестування продуктивності є важливою частиною розробки програмного забезпечення. Воно допомагає виявити вузькі місця та помилки, а також гарантує, що додатки швидко реагують на запити користувачів. Одним з особливо важливих аспектів є час, необхідний веб інтерфейсу для формування відповіді на дії користувача.

Від швидкості формування відповіді прямо залежить скільки запитів від користувачів система може обробити одночасно. А це, своєю чергою, прямо впливає на витрати компанії на утримання інфраструктури. Неефективний код потребує більше ресурсів на виконання однакових функцій що відповідно потребує більше витрат на утримання додаткових серверів або хмарних послуг.

Як зазначалося в розділі 1, перехід на мікросервісну архітектуру збільшує кількість мережевих з'єднань між різними модулями системи. Тому, вибір найбільш продуктивного протоколу для між процесної взаємодії є одним з головних виборів при проєктув анні систем такого типу.

В даному дослідженні порівнюєтеся продуктивність двох веб інтерфейсів опираючись на наступні показники:

- процесорний час. Цей показник впливає на швидкість реакції програми. Високі стрибки у використанні процесора можуть вказувати на різні проблеми. Зокрема, це може означати, що програма витрачає багато часу на обчислення, що призводить до погіршення реакції програми. Високі сплески використання слід вважати помилкою продуктивності, оскільки це означає, що процесор досягнув свого порогу використання;

- використання пам'яті. Також є важливим показником продуктивності програми. Високе використання пам'яті вказує на високе споживання ресурсів сервером;
- час виконання запиту. Коли мова заходить про швидкість обміну повідомленнями в мережі як правило згадують два показники: латентність та час завантаження. В даному досліджені було вирішено вимірювати повний час виконання запиту що включає процес (де)серілазації даних на клієнті та сервері та час обміну повідомлення мережею;
- пропускна здатність. Показник що характеризує скільки веб запитів може обробити система за певний проміжок часу.

Щоб досягти мети дослідження були визначені наступні вимоги та обмеження для програмної системи:

- порівнювати лише чисту пропускну здатність, не залучаючи до процесу жодної бізнес-логіки;
- не читати дані з баз даних або диска та не генерувати відповіді на сервері під час запиту;
- використовувати ідентичні набори даних для тестування обох протоколів;
- використовувати різні типи даних в одному повідомленні;
- відключити логування та інші допоміжні модулі що можуть впливати на продуктивність системи;
- використовувати синхроні операції для обміну повідомленнями;
- за можливістю використовувати нативні або де-факто стандартні бібліотеки для коду сервісу та клієнта для зниження можливих похибок через використання неефективних за часом виконання бібліотек;
- система повинна мати попередню підготовку до тестування.

Для забезпечення варіативності проведених замірів пропонуємо перевіряти системи за наступними змінними:

- операція запису та читання від клієнта до сервера;
- різні за розміром повідомлення;
- різна кількість запитів за ітерацію;
- різна кількість клієнтів що звертаються до одного серверу за ітерацію.

За результатами проведених замірів побудуємо таблицю 3.1 для кожної з змінних.

Таблиця 3.1 - Приклад таблиці з результатами порівняння продуктивності

	Малий розмір повідомлення		Середній розмір повідомлення		Великий розмір повідомлення	
	REST	gRPC	REST	gRPC	REST	gRPC
Середня пропускна здатність, запитів/с						
Середній час завантаження, мс						
Використання процесору, %						

Розмір повідомлення контролюється кількістю об'єктів що передаються. Так, малий розмір складається з одного об'єкту, середній з двадцяти об'єктів, а великий зі 100 об'єктів. Як зазначалося у вимогах, для кожного тестового сценарію було використано однаковий набір даних, тобто розмір повідомлення константний.

Слід зазначати що абсолютні показники не є метою дослідження. Однакові тести запущені на різних машинах покажуть різні результати. Тому, метою дослідження є порівняння відносних показників продуктивності двох систем.

3.2 Архітектура та проектування програмного додатку

Для проведення дослідження було спроектовано систему наведену на рисунку 3.1. Система складається з двох сервісів: сервера та клієнта. Окремо на діаграмі

позначено додаток VisualVM, інструмент для профілювання систем написаних мовою програмування Java.

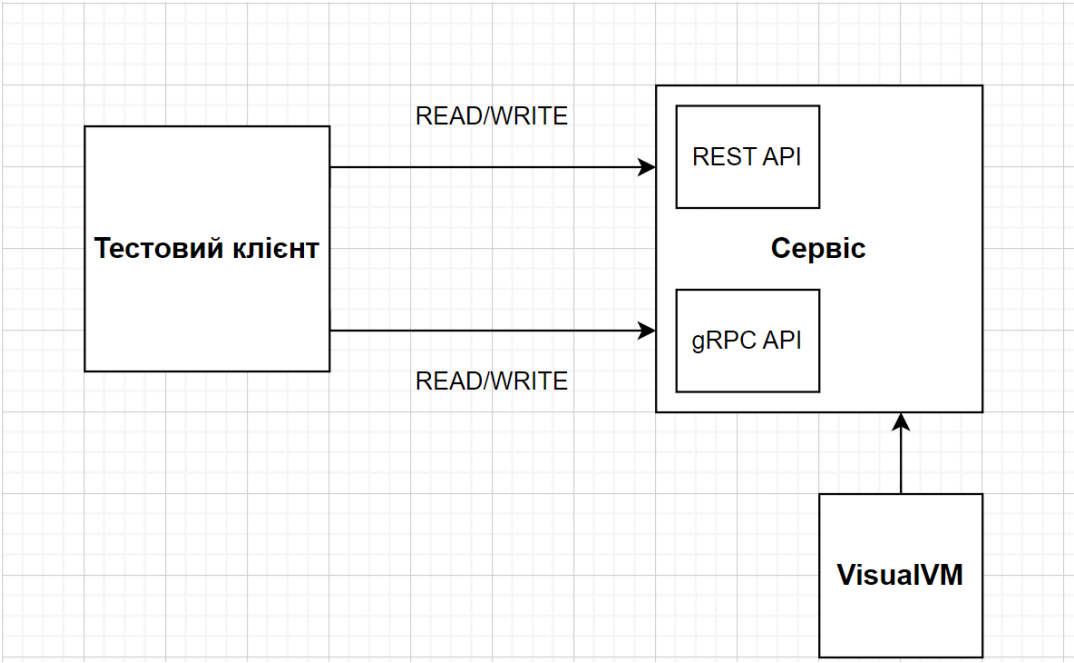

Рисунок 3.1 - Діаграма проекта системи

Сервіс має два веб інтерфейси. Один на протоколі REST з використанням текстового протоколу JSON, а інший використовувати протокол gRPC з двійковим форматом Protobuf. Обидва інтерфейси будуть реалізовувати API наведений на рисунку 3.2.

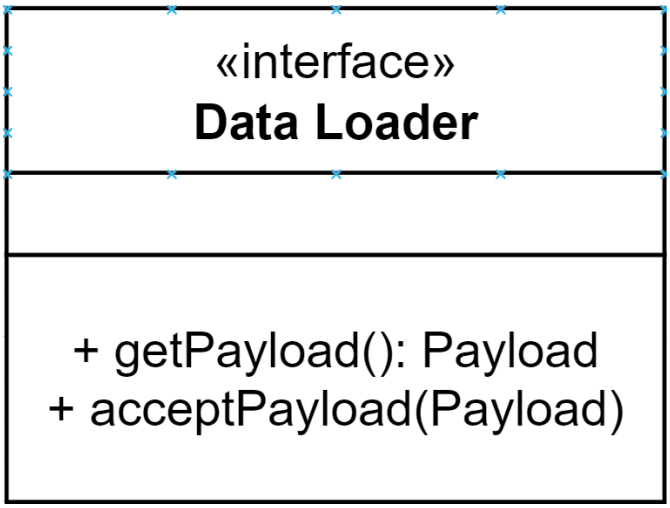

Рисунок 3.2 - Діаграма інтерфейсу для API сервісу

Сервіс не має жодної бізнес-логіки, підключення до баз даних або інших ресурсів. Метод getPayload повертає попередньо генерований статичний набір даних. Розмір повідомлення контролюєтеся через параметри запиту та мати три розмірності. Метод acceptPayload приймає дані на стороні сервера в тому самому форматі та розмірності. Як зазначалося вище, сервіс не має жодної бізнес-логіки. Точкою початку відповіді для клієнта є завершення перетворення запиту на внутрішні класи програми. Тобто, десеріалізація даних також включена в результати дослідження та відображена у часі завантаження.

Мовою програмування обрано Java. Ця мова залишається однією з найпопулярніших протягом десятиріч та дуже часто використовується у великих компаніях для побудови власних систем. Саме такі системи найбільше зацікавлені в оптимізації інфраструктури, тому що навіть незначна, у відсотках, оптимізація може зменшити витрати на власні або хмарні сервери у десятки та сотні тисячі умовних грошових одиниць.

Як систему збірки проєкту було обрано Maven. Структура проєкту складається з трьох програмних модулів: сервіс, клієнт та спільна тека з різними ресурсами, як підготовлені дані для тестування у вигляді .json файлів та опис функціонала сервісу та повідомлень у .proto форматі.

Рисунок 3.3 - Структура проекта

Для побудови бази сервісу та реалізації REST інтерфейсу було обрано фреймворк Spring. Він є де-факто стандартом в галузі Java програмування, різні його версії використовується у більшості наявних систем. Також фреймворк має дуже гарну підтримку для написання REST контролерів, а його популярність означає гарну оптимізацію. Але фреймворк не має підтримки протоколу gRPC. Для реалізації цього контролеру використовуються бібліотеки представлені самими розробниками з Google.

Для виміру часу завантаження та пропускної здатності системи використовується спеціально розроблений тестовий клієнт. Він не має жодної логіки окрім викликів інтерфейсів сервісу та вимірів результатів. Було прийнято рішення реалізувати логіку вимірів саме на клієнті, тому що в цьому випадку можна відокремити час виконання запитів від часу що необхідно витратити на фіксацію результатів.

Для виміру використання пам’яті та процесора було застосовано інструмент Java VisualVM. Java VisualVM - це інструмент, який надає візуальний інтерфейс для перегляду детальної інформації про Java-програми під час їх запуску на віртуальній машині Java (JVM), а також для усунення несправностей і профілювання цих додатків.

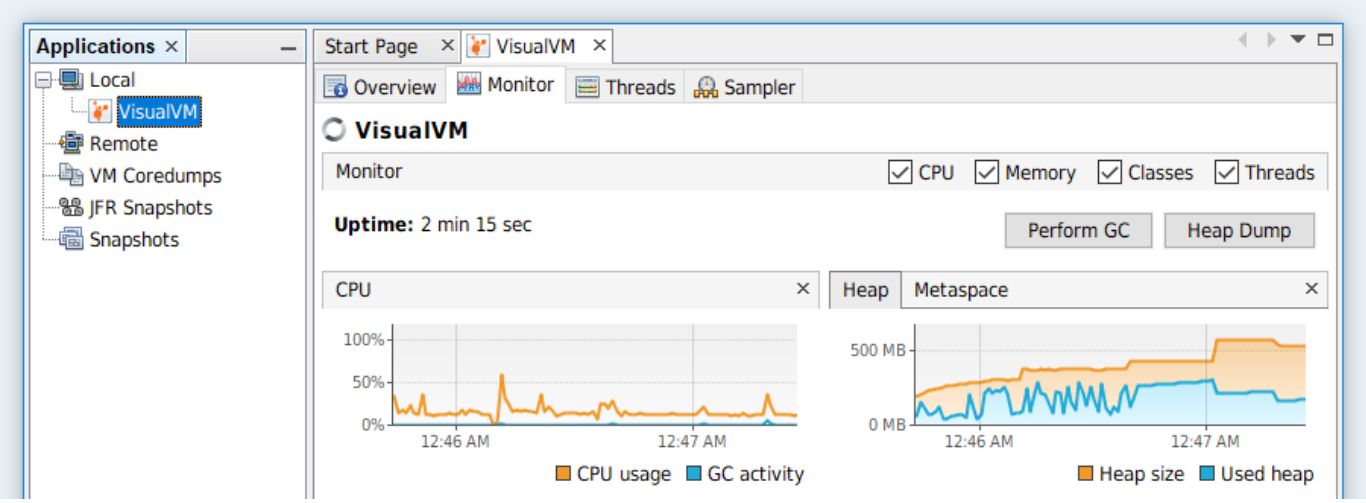

Рисунок 3.4 - Інтерфейс додатку VisualVM

Більшість інструментів JConsole, jstat, jinfo, jstack та jmap, які раніше були окремими інструментами, є частиною Java VisualVM. Додаток об'єднує ці інструменти для отримання даних з програмного забезпечення JVM, а потім реорганізовує і представляє інформацію графічно.

Java VisualVM може використовуватися розробниками Java додатків для усунення несправностей, моніторингу та покращення продуктивності додатків. Java VisualVM дозволяє розробникам генерувати та аналізувати дампи купи, відстежувати витоки пам'яті, виконувати та контролювати збір сміття, а також виконувати легке профілювання пам'яті та центрального процесора.

Повна реалізація програми наведена на сайті GitHub (Додаток Д).

3.3 Розробка тестового плану та огляд результатів дослідження

Для проведення дослідження було обрано навантажувальне тестування (англ. Load testing) та об’ємне тестування (англ. Volume testing). Навантажувальне тестування проводиться для того, щоб оцінити поведінку програми (додатка) із заданим очікуваним навантаженням, наприклад кількість користувачів що користуються системою одночасно. Об’ємне тестування передбачає зміну розміру даних що використовуються для оцінки поведінки системи. В даному досліджені було поєднано обидва підходи та проведено тестування з різною кількість користувачів та об’ємами даних одночасно.

Було вирішено скласти три тестові сценарії з різною кількістю користувачів. В першому сценарії кількість становила 10 користувачів, в другому 50 користувачів та в третьому 100 користувачів. Кожен користувач був представлений окремим потоком виконання, але метод та розмір повідомлень був однаковий для кожного з них.

Кожен сценарій тестував дві операції програмного інтерфейсу: отримання даних з сервера та посилання даних на сервер. Кожен користувач викликав один з двох методів програмного інтерфейсу 100 разів без додаткової затримки або обробки інформації.

Для використання як повідомлення у тестуванні було обрано готовий набір даних з мережі Інтернет від компанії NASA про місце та час падіння метеоритів [34]. Цей набір даних був обраний через наявність різних типів даних у кожному повідомлені що дозволяє більш широко розглянути можливості двох фреймворків щодо серіалізації даних. Приклад одного повідомлення з набору даних представлений на рисунку 3.5.

```
{
  "name": "Aachen",
  "id": "1",
  "nametype": "Valid",
  "recclass": "L5",
  "mass": "21",
  "fall": "Fell",
  "year": "1880-01-01T00:00:00.000",
  "reclat": "50.775000",
  "reclong": "6.083330",
  "geolocation": {
    "type": "Point",
    "coordinates": [
      6.08333,
      50.775
    ]
  }
}
```


Рисунок 3.5 – Приклад повідомлення з тестового набору даних

Для того, щоб дослідити залежність продуктивності фреймворка від розміру повідомлення було вирішено розділити тестові дані на три різні набори даних. Так в малому наборі даних представлено лише одне повідомлення, в середньому наборі

даних представлено 100 повідомлень та в великому 1000 повідомлень одночасно. Кожен з наборів даних протестували окремо під час кожного тестового сценарію. Тобто, загалом було виконано 9 різних тестових сценаріїв для заміру часу завантаження та пропускної здатності.

Після кожного тесту виводилась інформація щодо загальної кількості запитів для всіх користувачів під час тесту та розрахована середня швидкість виконання одного запиту та кількість запитів за секунду. Приклад наведено на рисунку 3.6.

```
Test execution start at: 2023-04-23T19:13:37.3408928952
Get payload gRPC API finished
Total number of threads: 10
Total number of requests per thread: 100
Total number of requests per test: 1000
Average execution for single call is: 8.352
Average number of requests per second 7183
Test execution ends at: 2023-04-23T19:13:38.2384099412
```

Рисунок 3.6 – Приклад інформація для кожного тесту

Так як кожний тест повертає різні результати було вирішено наступне: повторити кожний тестовий сценарій 11 раз та за допомогою медіани, за часом викоання, обрати показники для порівняння. Розглянемо результати дослідження у таблиці 3.2.

Згідно таблиці 3.2 можна зробити висновки що фреймворк gRPC є більш продуктивним для всіх розмірів повідомлень та низькому навантаженню на сервер. Пропускна здатність gRPC вище ніж REST приблизно у 7 разів для середнього розміру повідомлень та 4 рази для великого розміру повідомлень. Фреймворк gRPC також потребує менше процесорного часу на обробку повідомлень. Так, при середньому розмірі повідомлень gRPC потребує на 11,2 % та 6 % для великого розміру

повідомлень. Можна дійти висновку що gRPC найбільш ефективний, у порівнянні з REST, саме при середньому розмірі повідомлень.

Таблиця 3.2 - Результати тестового сценарію для 10 користувачів

	Малий розмір повідомлення		Середній розмір повідомлення		Великий розмір повідомлення	
	REST	gRPC	REST	gRPC	REST	gRPC
Середня пропускна здатність, запитів/с	857142	1818181	100166	800000	8771	46403
Середній час завантаження, мс	0,07	0,033	0.599	0.075	6.84	1.293
Використання процесору, %	5.9	2,6	16.5	5.3	20	14

Згідно з таблицей 3.3 можна зробити висновки що фреймворк REST є більш продуктивним при малому розміру повідомлень та високому навантаженню на сервер. Пропускна здатність REST на 45 відсотків більше ніж у gRPC. Але слід зазначити що REST використовує на 16,6 відсотка більше процесорного часу. При збільшенні розміру повідомлень фреймворк gRPC показує кращі результати. Пропускна здатність зростає приблизно на 407 та 484 відсотки при середньому та великому розмірі повідомлень та потребується менше процесорного часу для обробки повідомлень на 9,4 та 4,1 відсотка у порівняння з REST.

Таблиця 3.3 - Результати тестового сценарію для 100 користувачів

	Малий розмір повідомлення		Середній розмір повідомлення		Великий розмір повідомлення	
	REST	gRPC	REST	gRPC	REST	gRPC
Середня пропускна здатність, запитів/с	131521	90334	8842	44853	1029	6017
Середній час завантаження, мс	0.4562	0.6642	6.7853	1.3377	58.2989	9.9702
Використання процесору, %	27.4	10.8	28.2	18.8	31.3	27.2

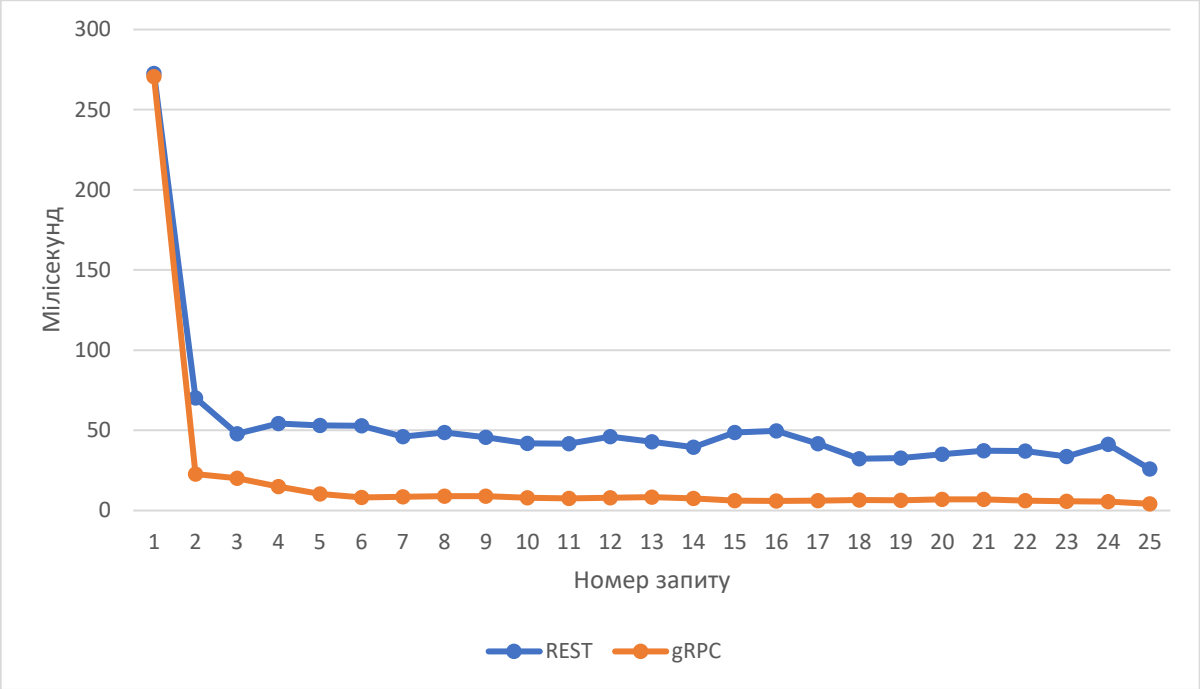

Рисунок 3.7 – Середня швидкість виконання запиту з запису повідомлень на сервер

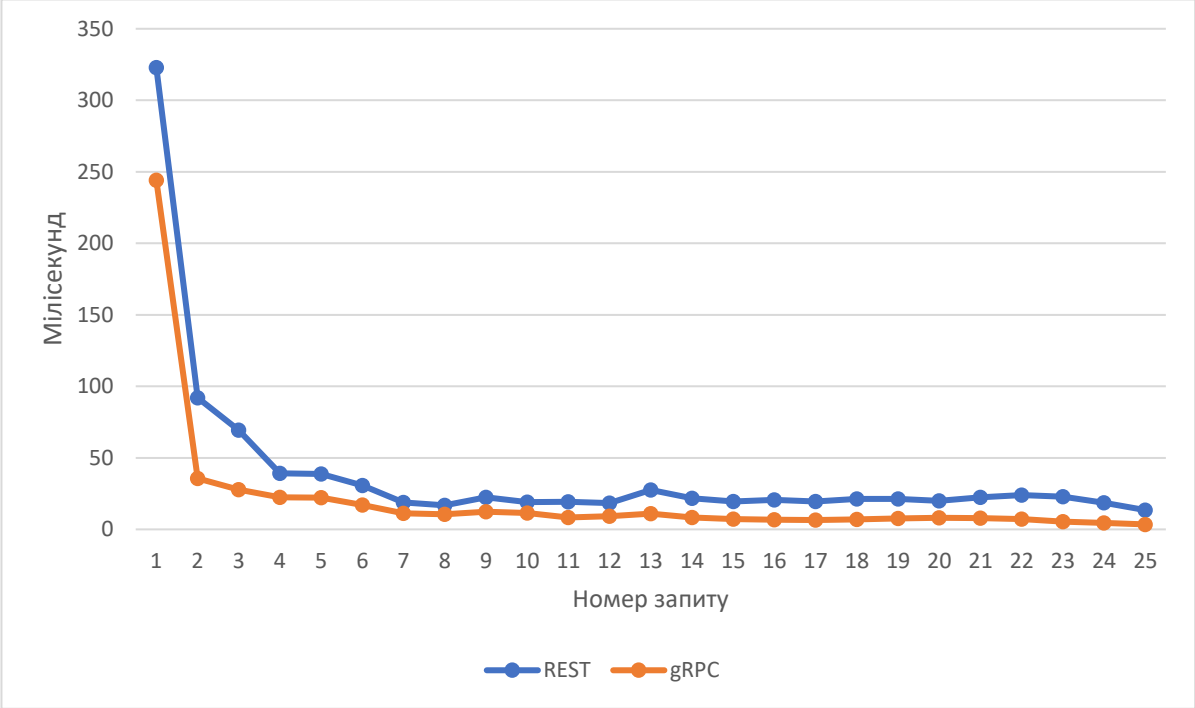

Рисунок 3.8 – Середня швидкість виконання запиту з отримання повідомлень

Для порівняння продуктивності також доцільно розглянути графік середньої швидкості виконання запитів. Для цього було проведено додаткові тестові сценарії при використанні великого розміру повідомлень та 10 користувачів. Кожен потік виконання виконував 25 запитів до сервера.

З рисунків 3.7 та 3.8 можна зробити висновок що середній час для виконання запиту у gRPC приблизно у 5 разів менше для запису даних та у 2,5 раза менше при зчитуванні даних. Слід зазначити що перше повідомлення у тесовому сценарії виконується набагато довше ніж інші через необхідність встановлення першого зв’язку між клієнтом та сервером. Накладні витрати на цю операцію приблизно однакові для обох фреймворків.

Для оцінки використання оперативної пам’яті було проведено додатковий тестовий сценарій. В ньому 10 користувачів повторювали запити до сервера з великим розміром повідомлень 300 разів.

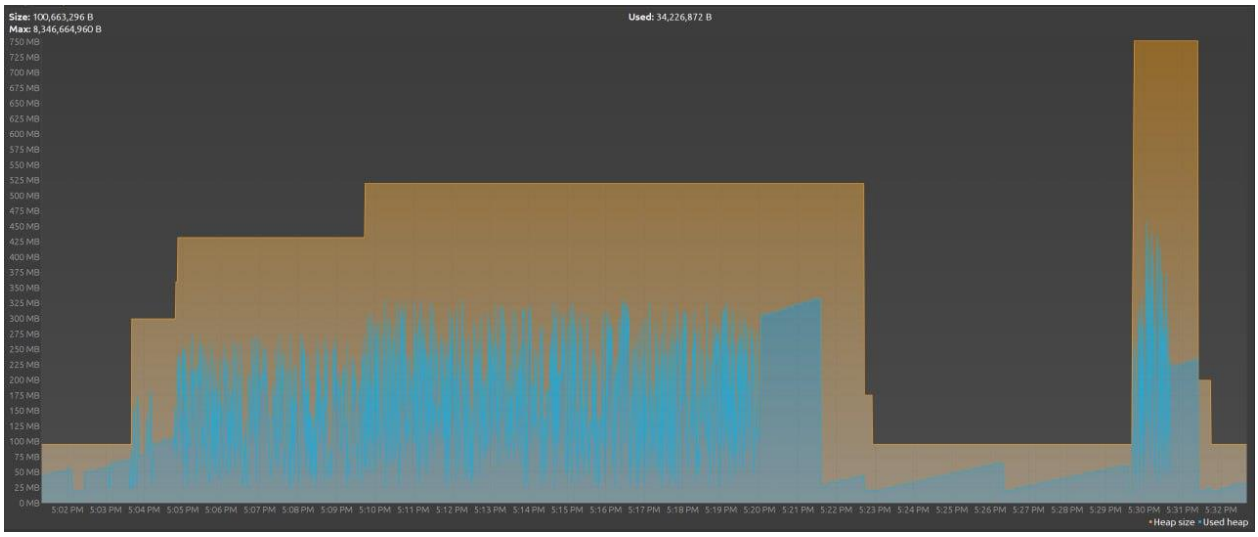

Рисунок 3.9 – Використання оперативної пам’яті

Ліва частина графіку відображає використання пам'яті фреймворком REST, а права фреймворком gRPC. Проміжок між ними відповідає відсутності будь-яких запитів. Можна побачити що в середньому REST на піку використав приблизно 325 мегабайтів оперативної пам’яті, тоді як gRPC приблизно 425 мегабайтів. Але слід

зазначити що тестовий сценарій з використанням REST зайняв 17 хвилин тоді як gRPC лише 1 хвилину, тобто в 17 разів менше. Це пов'язано з кращою можливістю REST для обробки паралельних запитів.

Згідно з проведеним дослідженням можна сформувати пропозицію щодо оптимальної архітектури сучасної системи побудованої на архітектурі мікросервісів. Розглянемо її на рисунку 3.10.

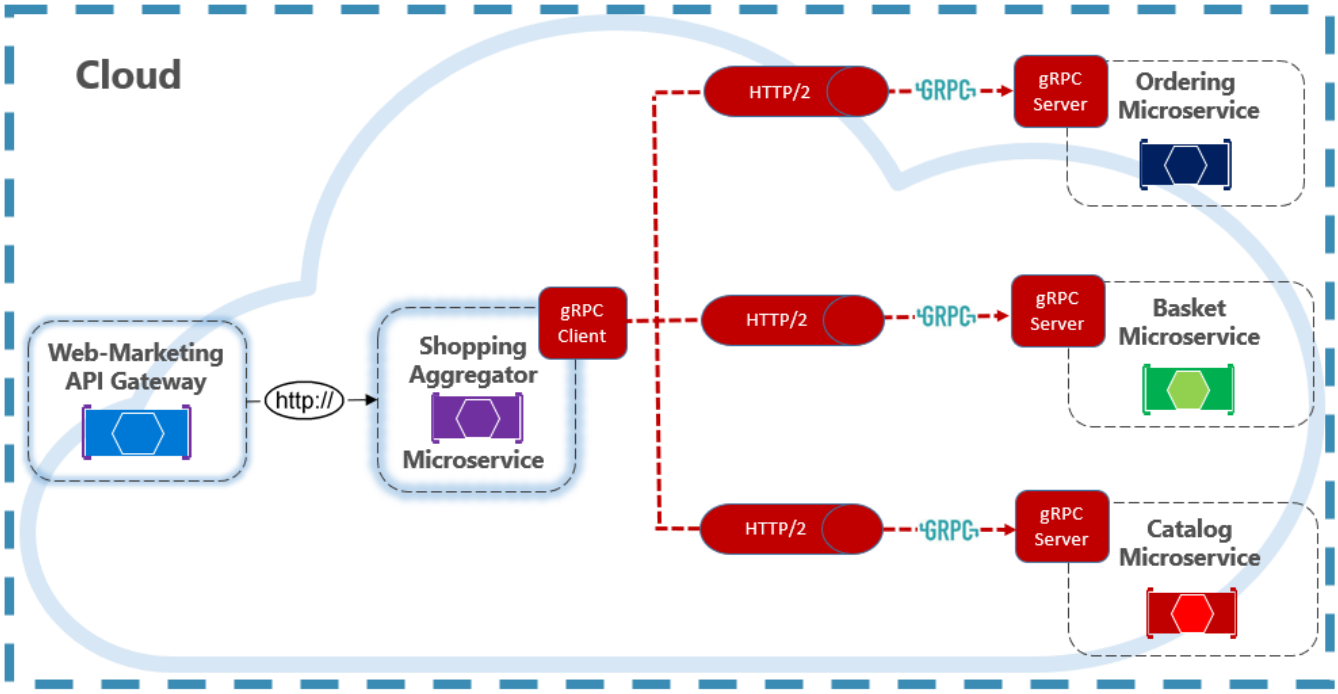

Рисунок 3.10 – Оптимальна архітектура сучасної системи побудованої на мікросервісній архітектурі [35]

Клієнти з'єднуються з системою через REST або GraphQL, використовуючи всі переваги браузерної підтримки для цих форматів, до спеціального сервісу побудованого за патерном BFF (скор. англ. Backend for Frontend) або іншими, схожими за функціями патернами. Своєю чергою цей сервіс отримує дані з інших сервісів, прихованих від кінцевого споживача, за допомогою фреймворку gRPC. Така комбінація використовує переваги обидвох фреймворків та оптимізує продуктивність всієї системи.

ВИСНОВКИ

За останні роки мікросервіси привернули до себе величезну увагу та завоювали популярність серед індустрії. Архітектура побудована на мікросервісах допомогла таким великим організаціям як Amazon та Netflix обслуговувати мільйони запитів за хвилину. Хоча формально визначити архітектуру мікросервісів складно, існують загальні характеристики, які можна з нею асоціювати, а саме:

- Вільно з'єднані та незалежні в процесі введеня в дію;
- Зв'язок між сервісами відбувається на рівні процесів через мережу, а не на рівні функцій чи класів;
- Кожен сервіс може мати власний стек мови програмування, фреймворків та моделі баз даних.

Особливу увагу в даній роботі приділено зміні у зв'язку між сервісами. Зростання кількості мережевих викликів веде до зростання необхідних ресурсів для їх обробки та відповідно витрати на ці ресурси. Також критичним стає час виконання запитів. Все це спонукає галузь до пошуку шляхів оптимізації трафіку між сервісами. Одним з варіантів оптимізації трафіку виступає використання новітніх протоколів. Так, протокол gRPC розроблений компанією Google використовую сучасну версію транспортного протоколу HTTP/2 що має ряд переваг у порівнянні з попередньою версією HTTP/1. Наприклад, мультиплексування запитів надає змогу використовувати одне TCP з'єднання для декількох запитів.

Для апробації теоретичних переваг від використання новітнього протоколу було сформовано вимоги для побудови тестової системи для перевірки ефективності та продуктивності двох найпопулярніших протоколів.

Згідно з поставленими вимогами було розроблено тестову систему що надавала можливість викликати однакові методи за допомогою обох фреймворків. Також, було

розроблено тестовий клієнт для цієї системи що реалізував заплановані тестові сценарії.

За результатами дослідження було отримано наступні результати: для систем з великим навантаженням gRPC продемонстрував значно кращі результати. В середньому пропускна здатність, а відповідно і час обробки запиту, від 4 до 7 разів вища ніж REST. Лише за одним сценарієм REST показав кращий результат, при передачі малих за розміром повідомлень в низько навантаженій системі. Також gRPC використовував менше ресурсів системи для обробки запитів.

Фреймворк gRPC - може не підходити для кожної архітектури та середовища. Важливо уникати сліпої відмови від REST і переходу на gRPC. Архітектура REST наразі є провідним рішенням для взаємодії сервісів і, ймовірно, залишиться такою в близькому майбутньому завдяки своїй популярності, підтримці та простоті. Тому дуже важливо ретельно зважити всі "за" і "проти" обох технологій, перш ніж приймати рішення про міграцію.

Хоча gRPC, безсумнівно, пропонує багато переваг і має великий потенціал, може бути розумно почекати ще рік або два. Це дозволить спільноті gRPC вирости, що призведе до кращої підтримки розробників і полегшить інтеграцію gRPC.

В завершені дослідження була сформована пропозиції щодо оптимальної архітектури системи побудованої на мікросервісах. Вона представлена на рисунку 3.10.

В процесі виконання роботи було проведено апробацію результатів на 27-му Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті (додаток Б).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microservices use in organizations worldwide 2021 | Statista. Statista. URL: https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/ (date of access: 28.03.2023).

2. 6 overlooked facts of microservices. Enable Architect. URL: https://www.redhat.com/architect/microservices-overlooked-facts (date of access: 28.03.2023).

3. Falatiuk H., Shirokopetleva M., Dudar Z. Investigation of architecture and technology stack for e-archive system. 2019 IEEE international scientific-practical conference problems of infocommunications, science and technology (PIC S&T), Kyiv, Ukraine, 8–11 October 2019. 2019. URL: https://doi.org/10.1109/picst47496.2019.9061407 (date of access: 09.05.2023).

4. Siraj ul Haq. Introduction to monolithic architecture and microservices architecture. Medium. URL: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63 (date of access: 28.03.2023).

5. Lamersdorf W. Paradigms of distributed software systems Services, processes, and self-organization. Proceedings of the International Conference on Security and Cryptography, Seville, 18 July 2011.

6. Alami-Kamouri S., Orhanou G., Elhajji S. Overview of mobile agents and security. 2016 international conference on engineering & MIS (ICEMIS), Agadir, Morocco, 22–24 September 2016. 2016. URL: https://doi.org/10.1109/icemis.2016.7745371 (date of access: 18.04.2023).

7. Erl T. Service-Oriented architecture (SOA): concepts, technology, and design (the prentice hall service-oriented computing series from thomas erl). Prentice Hall PTR, 2005. 792 p.

8. SOAP specifications. World Wide Web Consortium (W3C). URL: https://www.w3.org/TR/soap/ (date of access: 08.05.2023).

9. The evolution of distributed systems towards microservices architecture / T. Salah et al. 2016 11th international conference for internet technology and secured transactions (ICITST), Barcelona, Spain, 5–7 December 2016. 2016. URL: https://doi.org/10.1109/icitst.2016.7856721 (date of access: 18.04.2023).

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

11. Schermann G., Cito J., Leitner P. All the services large and micro: revisiting industrial practice in services computing. International conference on service-oriented computing, Berlin. Berlin, 2015. P. 36–47.

12. Mazlami G., Cito J., Leitner P. Extraction of microservices from monolithic software architectures. 2017 IEEE international conference on web services (ICWS), Honolulu, HI, USA, 25–30 June 2017. 2017. URL: https://doi.org/10.1109/icws.2017.61 (date of access: 18.04.2023).

13. Evans E. Domain-Driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2003. 576 p.

14. Newman S. Building microservices: designing fine-grained systems. Sebastopol, USA : O'Reilly Media, 2015. 280 p.

15. Perspectives of system informatics / ed. by A. K. Petrenko, A. Voronkov. Cham : Springer International Publishing, 2018. URL: https://doi.org/10.1007/978-3-319-74313-4 (date of access: 18.04.2023).

16. Jaramillo D., Nguyen D. V., Smart R. Leveraging microservices architecture by using Docker technology. SoutheastCon 2016, Norfolk, VA, USA, 30 March – 3 April 2016. 2016. URL: https://doi.org/10.1109/secon.2016.7506647 (date of access: 18.04.2023).

17. Richardson C. Microservices patterns: with examples in java. Manning Publications, 2018. 522 p.

18. Remote procedure call (RPC) | CQR. CQR. URL: https://cqr.company/ua/wiki/protocols/remote-procedure-call-rpc/ (date of access: 08.05.2023).

19. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. IRVINE, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (date of access: 28.03.2023).

20. MQTT – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/MQTT (дата звернення: 28.03.2023).

21. RMI. Oracle. URL: https://docs.oracle.com/javase/tutorial/rmi/index.html (date of access: 28.03.2023).

22. Introduction to gRPC. gRPC. URL: https://grpc.io/docs/what-is-grpc/introduction (date of access: 28.03.2023).

23. Ueda T., Nakaike T., Ohara M. Workload characterization for microservices. 2016 IEEE international symposium on workload characterization (IISWC), Providence, RI, USA, 25–27 September 2016. 2016. URL: https://doi.org/10.1109/iiswc.2016.7581269 (date of access: 23.04.2023).

24. Du S. G., Lee J. W., Kim K. Proposal of GRPC as a new northbound API for application layer communication efficiency in SDN. IMCOM '18: the 12th international conference on ubiquitous information management and communication, Langkawi Malaysia. New York, NY, USA, 2018. URL: https://doi.org/10.1145/3164541.3164563 (date of access: 23.04.2023).

25. Fernando R. Evaluating performance of REST vs. grpc. Medium. URL: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da (date of access: 23.04.2023).

26. Leung M. gRPC vs REST–performance comparison. Medium. URL: https://medium.com/analytics-vidhya/grpc-vs-rest-performancecomparison-1fe5fb14a01c (date of access: 23.04.2023).

27. YONEGO | strategisch slim & tactisch sterke marketing. Yonego. URL: https://www.yonego.com/nl/blogs/why-milliseconds-matter/ (date of access: 23.04.2023).

28. What are restful web services - go coding. Go Coding. URL: https://gocoding.org/what-are-restful-web-services/ (date of access: 22.04.2023).

29. Birrell A. D., Nelson B. J. Implementing Remote procedure calls. The ninth ACM symposium, Bretton Woods, New Hampshire, United States, 10–13 October 1983. New York, New York, USA, 1983. URL: https://doi.org/10.1145/800217.806609 (date of access: 22.04.2023).

30. Remote procedure call (RPC) in operating system - geeksforgeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/ (date of access: 22.04.2023).

31. Rei. A minimalist guide to gRPC. Medium. URL: https://itnext.io/a-minimalist-guide-to-grpc-e4d556293422 (date of access: 23.04.2023).

32. Gurpreet Kaur and Mohammad Muztaba Fuad. “An evaluation of protocol buffer”. In: Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon). IEEE. 2010, pp. 459–462

33. Introduction to HTTP/2. Google. URL: https://developers.google.com/web/fundamentals/performance/http2 (date of access: 28.03.2023).

34. Meteorite landings | NASA open data portal. NASA Open Data Portal. URL: https://data.nasa.gov/Space-Science/Meteorite-Landings/gh4g-9sfh/explore (date of access: 23.04.2023).

35. GRPC. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc (date of access: 23.04.2023).