**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

**Інститут Прикладного Системного Аналізу**
**Кафедра Системного Проектування**

До захисту допущено:
Завідувач кафедри
Вадим МУХІН
«___»___________2023 р.

# Дипломна робота

**на здобуття ступеня бакалавра**
**за освітньо-професійною програмою**
**«Інтелектуальні сервіс-орієнтовані розподілені обчислення»**
**спеціальності Комп'ютерні науки на тему:**
«Використання фреймворку gRPC для реалізації комунікації між веб-сервісами»

Виконав:
студент IV курсу, групи ДА-91
Ахмедов Магомед Шамільович
__________

Керівник:
доцент, к.т.н.
Булах Богдан Вікторович
__________

Рецензент:
__________

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент __________

Київ – 2023

**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Навчально-науковий інститут прикладного системного аналізу**

**Кафедра системного проектування**

Рівень вищої освіти: перший (бакалаврський)

Спеціальність: 122 «Комп'ютерні науки»

Освітньо-професійна програма:

«Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_______ Вадим МУХІН

«___»_____________2023 р.

**ЗАВДАННЯ**
**на дипломну роботу студенту**
**Ахмедова Магомеда Шамільовича**

1. Тема роботи «Використання фреймворку gRPC для реалізації комунікації між веб-сервісами», керівник роботи Булах Богдан Вікторович, доцент, затверджені наказом по університету від «___»_________ 20__ р. №_____
2. Термін подання студентом роботи – «___»_________ 20__ р.
3. Вихідні дані до роботи
4. Зміст роботи:
    1. Вступ.
    2. Особливості протоколу gRPC
    3. Практичне використання gRPC у розподілених системах
    4. Порівняння gRPC з іншими технологіями
    5. Реалізація мікросервісної архітектури за допомогою платформи gRPC
    6. Функціонально вартісний аналіз програмного продукту
    7. Висновки
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)
    1. Презентація до захисту роботи.
6. Дата видачі завдання «___»_________ 20__ р.

Календарний план

| № з/п | Назва етапів виконання дипломної роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання | 02.12.2022 | |
| 2 | Дослідження платформи gRPC | 01.04.2023 | |
| 3 | Аналіз можливостей щодо використання gRPC у веб-сервісах | 05.04.2023 | |
| 4 | Реалізація інфраструктури на основі gRPC | 08.04.2023 | |
| 5 | Порівняння gRPC з іншими технологіями для побудови API | 19.04.2023 | |
| 6 | Розгляд поточних архітектур, які використовують gRPC для комунікації | 27.04.2023 | |
| 7 | Порівняння та підведення підсумків | 05.05.2023 | |

Студент Ахмедов М. Ш.

**Керівник** **Булах Б. В.**

# АНОТАЦІЯ

до дипломної роботи Ахмедова Магомеда Шамільовича на тему «Використання фреймворку gRPC для реалізації комунікації між веб-сервісами»

**Структура дипломної роботи:** Загальний об’єм пояснювальної записки: 149 сторінок, 68 рисунків, 5 таблиць, 56 посилання.

**Актуальність теми.** Використання gRPC у веб-сервісах може призвести до значного підвищення продуктивності, кращої сумісності та ефективнішого зв’язку між мікросервісами, що робить його важливою темою для вивчення в сфері розробки програмного забезпечення.

**Мета дипломної роботи:** всебічно розглянути технологію gRPC, дослідити її переваги та недоліки, зокрема на практиці шляхом реалізації тестового мікросервісного додатку.

**Об’єкт дослідження:** фреймворк gRPC.

**Предмет дослідження.** Використання протоколу grpc для розробки систем на основі сервіс-орієнтованої, зокрема - мікросервісної, архітектури.

Було досліджено основі компоненти на яких працює фреймворк gRPC. Виконано огляд реальних прикладів використання фреймворку gRPC. Розроблено мікросервісну архітектуру за допомогою фреймворку gRPC. Було порівняно фреймворк gRPC з іншими технологіями.

**Ключові слова:** gRPC, HTTP/2, protocol buffers, двонапрямлене потокове передавання, протоколи міжсервісної комунікації, веб-сервіси, архітектура мікросервісів.

## ANNOTATION

bachelor's thesis of Akhmedov Mahomed Shamilovych on “Application of the gRPC framework to implement the web services communication”

**Structure of the thesis:** Total volume of the explanatory note:

149 pages, 68 figures, 5 tables, 56 references.

**Relevance of the topic.** The use of gRPC in web services can lead to significant performance improvements, better interoperability, and more efficient communication between microservices, making it an important topic for study in software development.

**The purpose of the thesis:** to take a comprehensive look at gRPC technology, to investigate its advantages and disadvantages, in particular in practice by implementing a test microservice application.

**Object of research:** the gRPC framework.

**Subject of research.** The use of the grpc protocol for the development of systems based on service-oriented, in particular, microservice architecture.

The basic components on which the gRPC framework works were investigated. A review of real-world examples of using the gRPC framework is made. A microservice architecture was developed using the gRPC framework. The gRPC framework was compared with other technologies.

**Keywords:** gRPC, HTTP/2, protocol buffers, bidirectional streaming, inter-service communication protocols, web services, microservice architecture.

# ЗМІСТ

ЗМІСТ....................................................................................................9
ВСТУП...................................................................................................11
1. Особливості протоколу gRPC...............................................................13
1.1. Віддалений виклик процедур...........................................................13
1.2. Технологія Protocol Buffers..............................................................20
1.3. Протокол HTTP/2...............................................................................27
1.4. Висновки до першого розділу...........................................................34
2. Практика використання gRPC у розподілених системах...........................36
2.1. gRPC: Основні концепції, архітектура та життєвий цикл.....................36
2.2. Приклади використання gRPC у реальних проектах...........................42
2.3. Проблеми розподілених систем.........................................................59
2.4. Використання платформи gRPC для вирішення проблем розподілених систем....67
2.5. Висновки до другого розділу.............................................................73
3. Порівняння gRPC з іншими технологіями.................................................75
3.1. Реалізація протоколу RPC..................................................................75
3.2. Побудова API.....................................................................................81
3.3. Висновки до третього розділу.............................................................85
4. Реалізація мікросервісної архітектури за допомогою платформи gRPC.....87
4.1. Унарна комунікація...........................................................................87
4.2. Двостороння потокова комунікація...................................................106
4.3. Порівняння gRPC та REST сервісів....................................................117
5. Функціонально вартісний аналіз програмного продукту..........................123
5.1 Постановка задачі проектування........................................................123
5.2 Обґрунтування функцій програмного продукту...................................124
5.3 Обґрунтування системи параметрів програмного продукту..................126
5.4 Аналіз експертного оцінювання параметрів........................................129

5.5 Аналіз рівня якості варіантів реалізації функцій..................................132
5.6 Економічний аналіз варіантів розробки ПП.........................................134
5.7 Вибір кращого варіанту ПП техніко-економічного рівня.....................137
5.8 Висновки до четвертого розділу............................................................138
ВИСНОВОК..................................................................................................139
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.............................................................140

# ВСТУП

Сучасний стан технологічного сектору характеризується збільшенням кількості веб-сервісів і зростаючою складністю додатків і систем. Доволі популярною стає підхід розробки розподілених додатків. Тому ефективна взаємодія між веб-сервісами стає дедалі важливішою.

Одним з перших кроків при розробці додатку є правильне проектування початкової архітектури додатку. При проектуванні архітектури, постає важливе питання: «Чи є можливість спроектувати застосунок так, щоб він не був розподіленим?» або «Чи можливо зробити так, щоб весь функціонал був одним цілим, з фізичної точки зору?». Відповідь на це питання залежить від сфери для якої робиться додаток. Наприклад, у сфері Internet of things (IOT) це, за замовчуванням, не можливо. У сфері Web Development це можливо, але при умовах, що застосунком не буде користуватись багато людей та він загалом не буде великим. З часом, одна з цих умов (або обидві) порушуються, що призводить до купи проблем, як, наприклад, затримки на відповідь користувачів через перенавантаження застосунку. Для їх вирішення, розробники починають змінювати поточну архітектуру на розподілену.

Розподілений підхід до розробки породжує такий термін, як мікросервісна архітектура. При мікросервісній архітектурі додатки можна розкласти на дрібніші, слабко пов'язані сервіси, які можна розробляти, розгортати і керувати незалежно один від одного. Така архітектура пропонує безліч переваг, зокрема підвищену гнучкість, масштабованість і відмовостійкість.

Такий підхід також породжує багато проблем: затримки, комунікація між сервісами, балансування навантаження та інші. Для усіх цих проблем є готові паттерни, але їх реалізація може зайняти доволі багато часу. На цьому етапі можна знайти багато готових рішень, одним з таких є платформа gRPC.

Наступним кроком розробники мають обрати стек технологій, на основі якого буде написаний майбутній додаток.

Існує безліч мов програмування, які підходять для різних потреб. У мікросервісній архітектурі може бути багато сервісів і усі вони можуть задовольняти різні потреби. Таким чином, буває корисно написати один сервіс на одній мові програмування, а інший на іншій. Такий підхід не є стандартизованим, тому у більшості випадків, розробники будуть писати своє додаткове рішення, яке буде реалізовувати комунікацію між двома мовами. Все стає набагато складніше, коли мов стає все більше.

Це доволі відома проблема, тому для її вирішення існує багато підходів. У складі платформи gRPC наявний такий інструмент як Protobuf. Цей інструмент вирішує таку проблему шляхом надання стандартизації. Ця робота оглядово розгляне як за допомогою нього можна створювати API для мікросервісів, які будуть незалежні від мови на якій написаний мікросервіс.

Метою даної роботи є дослідження інструментів платформи gRPC для вирішення проблем, які виникають на етапі розробки розподілених систем.

Практичне завдання є реалізація мікросервісної інфраструктури на основі платформи gRPC, де будуть продемонстровані її можливості. gRPC використовує такий спосіб побудови API, як віддалений виклик процедур (RPC). Він буде порівняний з іншими, а саме: SOAP, REST та GraphQL.

Нарешті, у роботі надається уявлення про прийняття gRPC у промисловості та його майбутній потенціал у контексті мереж 5G, пристроїв IoT. Загалом, мета цієї роботи — всебічно розглянути технологію gRPC, дослідити її переваги та недоліки, зокрема на практиці шляхом реалізації тестового мікросервісного додатку.

## 1. Особливості протоколу gRPC

Цей розділ дипломної роботи має на меті забезпечити поглиблене вивчення різних особливостей протоколу gRPC.

Протокол gRPC базується на концепції віддаленого виклику процедур (RPC), дозволяючи клієнтам безперешкодно викликати віддалені методи в серверних додатках так, ніби вони є локальними методами. Однак, gRPC відрізняється від традиційних протоколів RPC завдяки включенню декількох особливостей, що робить його привабливим вибором для розробки сучасних розподілених систем.

Далі буде розглянуто Protobuf (протокольні буфери), двійковий формат серіалізації, який використовується в gRPC. Protobuf пропонує численні переваги над звичайними форматами обміну даними, включаючи компактний розмір, ефективне кодування та мовну незалежність. Всебічно розглянувши Protobuf, буде з'ясовано його роль у забезпеченні безперешкодної взаємодії між клієнтами та серверами gRPC.

Ще одним важливим аспектом протоколу gRPC є використання протоколу транспортного рівня HTTP/2. Буде розглянуто причину вибору HTTP/2, досліджено як цей протокол підвищує продуктивність і ефективність gRPC-зв'язку. Також буде розглянуто ключові особливості HTTP/2, які роблять його добре придатним для сучасних веб-додатків, і як gRPC використовує ці особливості для оптимізації зв'язку.

### 1.1. Віддалений виклик процедур

Комунікація в розподілених системах характеризується передачею інформації та управління між автономними додатками, що працюють на різних (віртуальних) машинах. Один з підходів до комунікації на рівні мови полягає у тому, щоб віддалений виклик процедур між розподіленими програмами виглядав і поводився як локальний виклик процедур у традиційних програмах.

Концепція віддаленого виклику процедур (далі RPC) дуже проста. Вона базується на спостереженні, що виклики процедур є добре відомим і зрозумілим механізмом управління та передачі даних у програмі, що працює на одному комп'ютері. Тому пропонується розширити цей механізм, щоб уможливити керування та передачу даних через комунікаційну мережу. Коли викликається віддалена процедура, середовище, що її викликає, припиняє роботу, а параметри передаються мережею в середовище, де виконується процедура (це середовище називається середовищем виклику), де виконується необхідна процедура. Коли процедура завершується і отримується її результат, результат передається назад у середовище виклику, і виконання продовжується, як при простому одиночному машинному виклику. Поки середовище виклику призупинено, інші процедури на цій машині можуть (потенційно) продовжувати виконуватися (залежно від особливостей паралелізму цього середовища та реалізації RPC).

**1.1.1. Визначення**

Терміни "віддалена процедура" та "віддалений виклик процедури" вже багато років використовуються членами мережевої групи Arpa для опису викликів процедур в Arpanet [55]. Вона неодноразово обговорювалася у відкритій літературі [54] принаймні з 1976 року, а кандидатська дисертація Нельсона [56] була поглибленим дослідженням можливостей проектування систем RPC і містила посилання на більшість попередніх досліджень RPC. Однак повномасштабні реалізації RPC були рідкіснішими, ніж дисертаційний проект.

Ідея RPC має багато привабливих аспектів. Один з них - чиста і проста семантика, яка полегшить побудову розподілених обчислень і зробить її правильною. Інший - ефективність. Процедурні виклики здаються досить простими, щоб значно пришвидшити комунікацію; третій - загальність. В одномашинних обчисленнях процедури часто є найважливішим механізмом зв'язку між частинами алгоритму.

Оскільки віддалені виклики процедур відбуваються між автономними програмами, помилка в одній програмі зазвичай не спричиняє помилки в іншій і зазвичай може бути виявлена іншою програмою (наприклад, якщо віддалений виклик не повертається). Така ізольованість і можливість виявлення помилок є ключовою особливістю розподілених програм і відрізняє їх від локальних програм (які використовують локальні виклики), де помилка зазвичай зупиняє все (наприклад, і програму, що викликає, і програму, що викликається).

### 1.1.2. Основи RPC

RPC технічно не є протоколом, його краще розглядати як загальний механізм конфігурації розподілених систем. RPC популярний тому, що він базується на семантиці локальних викликів процедур. Застосунок робить виклик процедури, незалежно від того, локальна вона чи віддалена, і блокується до тих пір, поки виклик не повернеться. Оскільки розробник програми може майже нічого не знати про те, чи є процедура локальною або віддаленою, завдання розробника значно спрощується. Якщо викликана процедура насправді є методом віддаленого об'єкта в об'єктно-орієнтованій мові, RPC відомий як віддалений виклик методів (RMI) [54]. Хоча концепція RPC проста, є дві основні проблеми, які роблять її більш складною, ніж локальні виклики процедур:

- Мережа між викликаючим і викликаним процесами має набагато складнішу природу, ніж материнська плата комп'ютера. Наприклад, вона може обмежувати розмір повідомлень і має тенденцію втрачати або змінювати порядок повідомлень.
- Комп'ютери, на яких виконуються процеси, що називаються процесами-викликувачами, можуть мати дуже різну архітектуру та формати представлення даних.

Таким чином, повний механізм RPC фактично складається з двох основних компонентів:

- Протокол для керування повідомленнями, що надсилаються між клієнтськими та серверними процесами, та обробки потенційно небажаних характеристик базової мережі.
- Мова програмування та підтримка компілятора (ця частина механізму RPC зазвичай називається компілятором-заглушкою) для пакування аргументів у повідомлення запиту на клієнтській машині, повернення цих повідомлень до аргументів на серверній машині та виконання тих самих дій із значенням, що повертається.

Спочатку клієнт викликає локальну заглушку процедури, передаючи їй аргументи, необхідні для виконання процедури. Заглушка приховує той факт, що процедура є віддаленою, перетворюючи аргументи в повідомлення-запит, а потім викликає протокол RPC для відправки повідомлення-запиту на серверну машину. На сервері протокол RPC надсилає повідомлення запиту серверній заглушці, яка перетворює його в аргументи процедури і викликає локальну процедуру. Після завершення серверна процедура повертає повідомлення-відповідь, яке передається протоколу RPC для відправки назад клієнту. Клієнтський протокол RPC передає це повідомлення клієнтській заглушці, яка перетворює його у значення і повертає клієнтській програмі.

### 1.1.3. Ідентифікатори в RPC

Дві функції, які повинен виконувати будь-який протокол RPC:

- Підтримка простору імен для однозначної ідентифікації процедури, яку потрібно викликати.
- Можливість зіставлення повідомлення-відповідь з відповідним повідомленням-запитом.

Перша проблема дещо схожа на проблему ідентифікації вузлів у мережі (наприклад, що роблять IP-адреси). Одним з варіантів проектування ідентифікації об'єктів є плаский або ієрархічний простір імен. У плоскому просторі імен кожній процедурі просто присвоюється неструктурований

унікальний ідентифікатор (наприклад, ціле число), і цей номер міститься в єдиному полі в повідомленні RPC-запиту. У цьому випадку необхідна певна центральна координація, щоб уникнути присвоєння одного і того ж номера процедури двом різним процедурам. Альтернативно, протокол може реалізувати ієрархічний простір імен, подібний до того, що використовується для імен файлів, який вимагає лише того, щоб "базове ім'я" файлу було унікальним в межах цього каталогу. Таким чином, завдання забезпечення унікальності імен процедур може бути спрощено: ієрархічний простір імен для RPC може бути реалізований шляхом визначення набору полів у форматі повідомлення запиту, по одному для кожного рівня іменування, наприклад, двох або трьох рівнів, які можуть бути ієрархічним простором імен.

Ключ до зіставлення повідомлення-відповіді з відповідним запитом полягає в унікальній ідентифікації пари запит-відповідь за допомогою поля ідентифікатора повідомлення. У повідомленні-відповіді поле ідентифікатора повідомлення має те саме значення, що й у повідомленні-запиті; коли клієнтський модуль RPC отримує відповідь, він використовує ідентифікатор повідомлення для пошуку відповідного невиконаного запиту Щоб зробити RPC-транзакцію схожою на виклик локальної процедури виклик блокується до отримання повідомлення-відповіді. Коли відповідь отримано, заблокований виклик ідентифікується на основі номера запиту у відповіді, значення повернення віддаленої процедури витягується з відповіді, і блок знімається, щоб виклик міг повернутися з цим значенням повернення.

Постійним викликом для RPC є робота з неочікуваними відповідями, які можна ідентифікувати за ідентифікатором повідомлення. Наприклад, розглянемо наступну хворобливу (але реалістичну) ситуацію. Клієнтська машина надсилає повідомлення-запит з ідентифікатором повідомлення 0, падає і перезавантажується, а потім надсилає не пов'язане з ним повідомлення-запит,

також з ідентифікатором повідомлення 0. Сервер може не знати, що клієнт впав і перезапустився. Клієнт ніколи не отримає відповідь на запит.

Одним із способів вирішення цієї проблеми є використання ідентифікатора завантаження. Ідентифікатор завантаження комп'ютера - це число, яке збільшується при кожному перезавантаженні комп'ютера. Це число зчитується з енергонезалежного сховища (наприклад, диска або флеш-накопичувача) під час процедури завантаження машини, збільшується і записується назад на пристрій зберігання даних. Цей номер додається до всіх повідомлень, надісланих з цього хоста. Якщо повідомлення з новим ідентифікатором завантаження отримано зі старим ідентифікатором повідомлення, його буде розпізнано як нове повідомлення. Коротше кажучи, ідентифікатор повідомлення та ідентифікатор завантаження об'єднуються, щоб сформувати унікальний ідентифікатор для кожної транзакції.

### 1.1.4. Подолання мережевих обмежень

Протоколи RPC часто виконують додаткові функції, щоб впоратися з тим, що мережі не є ідеальними каналами. Дві такі функції є:

- Забезпечення надійної доставки повідомлень.
- Підтримка великих розмірів повідомлень завдяки фрагментації та повторному збиранню.

Протокол RPC може "вирішити цю проблему", обравши роботу поверх надійного протоколу, такого як TCP, але в багатьох випадках протокол RPC реалізує свій власний надійний рівень доставки повідомлень поверх ненадійного субстрату (наприклад, UDP/IP). Такий протокол RPC, ймовірно, реалізує надійність за допомогою підтверджень і тайм-аутів, подібно до TCP.

Повідомлення, що містять дані (запити або відповіді), або підтвердження, надіслані для підтвердження таких повідомлень, можуть бути загублені в мережі. Щоб врахувати цю можливість, і клієнт, і сервер зберігають копію кожного надісланого повідомлення до отримання підтвердження про його

отримання. Кожна сторона також встановлює таймер RETRANSMIT і повторно передає повідомлення, коли таймер закінчується. Обидві сторони обнуляють цей таймер і повторюють спробу узгоджену кількість разів, перш ніж здадуться і видалять повідомлення.

Коли RPC-клієнт отримує повідомлення-відповідь, очевидно, що відповідне повідомлення-запит було отримано сервером. Таким чином, саме повідомлення-відповідь є неявним підтвердженням, і ніякого подальшого підтвердження з боку сервера логічно не потрібно. Аналогічно, повідомлення-запит може неявно підтверджувати попереднє повідомлення-відповідь. Передбачається, що протокол робить транзакції запит-відповідь послідовними, і що одна транзакція повинна бути завершена до початку наступної. На жаль, така послідовність сильно обмежує продуктивність RPC.

Щоб подолати цю ситуацію, протокол RPC реалізує абстракцію каналу. У межах каналу транзакції запиту/відповіді є безперервними, і в будь-який момент часу на даному каналі може бути лише одна активна транзакція, але може існувати декілька каналів. Іншими словами, абстракція каналу дозволяє мультиплексувати декілька транзакцій запиту/відповіді RPC між парами клієнт/сервер.

Кожне повідомлення містить поле ідентифікатора каналу, яке вказує, до якого каналу належить повідомлення. Повідомлення запиту для певного каналу неявно підтверджується, якщо попередня відповідь для цього каналу ще не була підтверджена. Програма може відкрити декілька каналів, якщо вона хоче мати декілька транзакцій запиту/відповіді з сервером одночасно (програма потребує декількох потоків). Повідомлення-відповідь використовується для підтвердження повідомлення-запиту, а наступний запит підтверджує попередню відповідь. Зверніть увагу, що паралельні логічні канали, дуже схожий підхід як спосіб підвищення продуктивності механізму надійності stop-and-wait, було описано в попередньому розділі.

Ще одна складність, яку має вирішити RPC, полягає в тому, що серверу може знадобитися як завгодно довгий час для отримання результату, або, що ще гірше, він може вийти з ладу до того, як згенерує відповідь. Варто зазначити, що тут йдеться про проміжок часу після того, як сервер підтвердив запит, але до того, як він відправив відповідь. Щоб допомогти клієнту відрізнити повільний сервер від сервера, який не працює, клієнтська частина RPC може періодично надсилати серверу повідомлення "Are you alive?", а сервер відповідатиме "ACK". Крім того, сервер може надсилати клієнту повідомлення "Я все ще живий" без попереднього запиту з боку клієнта. Цей підхід є більш масштабованим, оскільки він покладає більшу частину навантаження на клієнта (керування таймером очікування) на клієнтів.

Надійність RPC може включати в себе властивість, яку принаймні один раз називають семантикою. Це означає, що для кожного повідомлення-запиту, надісланого клієнтом, на сервер доставляється не більше однієї копії цього повідомлення. Кожного разу, коли клієнт викликає віддалену процедуру, ця процедура ніколи не викликається більше одного разу на серверній машині. Це пов'язано з тим, що мережа або машина сервера може вийти з ладу, і жодна копія повідомлення запиту не буде доставлена.

### 1.2. Технологія Protocol Buffers

Protocol Buffers (Protobuf) - це метод серіалізації даних, які можна передавати по мережі або зберігати у файлах. Інші формати, такі як JSON та XML, також використовуються для серіалізації даних. Хоча ці платформи зарекомендували себе як надзвичайно гнучкі та ефективні, вони не повністю оптимізовані для сценаріїв, коли дані мають передаватися між кількома мікросервісами в нейтральний до платформи спосіб. [1]

Саме ця проблема змусила Google створити формат ProtoBuf у 2008 році. З тих пір він широко використовується всередині Google і є форматом даних за замовчуванням для фреймворку gRPC. Спочатку Protobuf був створений для

трьох основних мов - C++, Java та Python. З роками багато мов, таких як Go, Ruby, JS, PHP, C# та Objective-C також почали підтримувати Protobuf. Поточна версія Protobuf називається proto3. [2]

Як і JSON та XML, protobuf не залежать від мови та платформи. Protobuf оптимізовано так, щоб він був швидшим за JSON та XML, оскільки з нього знято багато обов'язків, які зазвичай виконують формати даних, і він зосереджений лише на здатності серіалізувати та десеріалізувати дані якомога швидше. Інша важлива оптимізація стосується використання пропускної здатності мережі за рахунок зменшення обсягу даних, що передаються, наскільки це можливо. [3] [4]

Визначення даних для серіалізації записується у конфігураційних файлах, які називаються proto-файлами (.proto). Ці файли містять конфігурації, відомі як повідомлення. Proto-файли можна скомпілювати, щоб згенерувати код на мові програмування користувача.

### 1.2.1. Можливості

#### 1.2.1.1. Бінарний формат транспортування даних

Protobuf - це бінарний формат передачі, тобто дані передаються у двійковому вигляді. Це покращує швидкість передачі більше, ніж сирий рядок, оскільки займає менше місця і пропускної здатності.

Щодо навантаження на процесор, то при стандартному підході без стиснення буде надіслано більше даних. При стисненні буде відіслано меншу кількість даних, але операція стиснення теж буде завантажувати процесор. Таким чином, навантаження з мережі буде перенесено на процессор.

Єдиним недоліком є те, що файли Protobuf або дані не так добре читаються людиною, як JSON або XML

Якщо платформа не підтримує двійкові повідомлення, у Protobuf є можливість серіалізувати двійкові дані в рядок.

#### 1.2.1.2. Відокремлення контексту та даних

У JSON і XML дані і контекст не розділені, тоді як у Protobuf вони розділені. Розглянемо приклад JSON: (рис. 1.1)

```
{
  "first_name": "Mahomed",
  "last_name": "Akhmedov"
}
```

Рисунок 1.1 - Приклад JSON

У цьому прикладі передані дані мають об'єктний літерал з двома властивостями, first_name і last_name, зі значеннями Mahomed і Akhmedov. Це добре читається, але може займати більше місця. Тут кожне JSON повідомлення має містити обидві ці частини кожного разу. Зі збільшенням обсягу даних час передачі значно збільшиться.

Але у випадку з Protobuf все інакше. На початку потрібно визначити повідомлення в конфігураційному файлі, наприклад, так: (рис. 1.2)

```
{
  string first_name = 1;
  string last_name = 2;
}
```

Рисунок 1.2 - Приклад Protobuf

Цей конфігураційний файл містить контекстну інформацію. Числа - це лише ідентифікатори полів. Формат повідомлення буде розглянутий більш детально пізніше. Використовуючи цю конфігурацію, дані будуть надсилатись у закодованому вигляді: (рис. 1.3)

```
127Mahomed228Akhmedov
```

Рисунок 1.3 - Закодовані дані

У випадку 127Mahomed, 1 означає ідентифікатор поля, 2 - тип даних (тобто рядок), а 7 - довжину тексту. Варто зазначити, що це трохи складніше для читання, ніж JSON; однак, це повідомлення буде займати набагато менше місця у порівнянні з форматом JSON.

### 1.2.2. Формат повідомлення

Як було показано раніше, дані передаються у форматі Protobuf на основі конфігурації, яка називається повідомленням. Повідомлення зберігаються у файлах .proto. Давайте розглянемо приклад повідомлення. (рис. 1.4)

У наведеному прикладі для оголошення повідомлення використовується ключове слово message, за яким слідує визначена користувачем назва повідомлення. Літерали або компоненти оголошуються у фігурних дужках. Кожне поле літералу можна розділити на чотири компоненти: правило, тип, ім’я та мітка поля.

```
enum Corpus {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
  Corpus corpus = 4;
}
```

Рисунок 1.4 - Приклад повідомлення

*Правила поля*. У версії proto2 Protobuf перед типом поля або типом даних потрібно було додавати такі правила, як обов'язковий, необов'язковий і повторення; у версії proto3 це було оптимізовано, і залишилося лише правило повторення. Якщо поле є масивом елементів одного типу, воно вважається

повторюваним. Якщо поле не повторюється, ніяких додаткових правил додавати не потрібно.

*Типи полів.* Існує три категорії типів даних, які може зберігати поле. Перша - це скалярні типи даних, такі як рядки і числа; друга - перелічувані типи даних. У нашому прикладі це Corpus. І останній тип даних - це вбудоване повідомлення.

Як і в JSON і XML, типи повідомлень дозволяють побудувати ієрархію повідомлень, що представляють будь-який тип даних. Скалярні типи даних, які доступні в Protobuf: float, int32, int64, uint32, uint64, sint32, sint64 fixed32, fixed64, sfixed32, sfixed6, bool, string, bytes.

*Ім'я поля.* При іменуванні полів у Protobuf слід дотримуватися певних правил. Це пов'язано з тим, що вони приймаються компілятором protoc, коли він генерує код на основі .proto-файлу для обраної мови. Перше правило полягає в тому, що імена полів повинні бути в нижньому регістрі. По-друге, якщо назва поля містить більше одного слова, вони повинні бути розділені символами підкреслення.

*Мітки полів.* Мітка поля - це числове представлення поля, яке можна використовувати для того, щоб мати детальні назви полів у визначенні без необхідності їх телеграфувати.

Є кілька моментів, на які слід звернути увагу при роботі з мітками полів. По-перше, мітки полів повинні бути унікальними в межах повідомлення. По-друге, мітки полів повинні бути цілими числами. По-третє, якщо поле потрібно вилучити з визначення, яке вже використовується, тег має бути оголошений як reserved (зарезервований, закріплений), щоб його не можна було перезаписати.

**1.2.3. Генерація коду на основі proto файлів**

Найважливішим компонентом Protobuf є компілятор protoc. У приведеному прикладі (рис. 1.5) буде відтворена генерація коду мови Java на основі proto файлу.

Тепер, при наявності proto-файлу, наступне, що потрібно зробити, це згенерувати класи, які знадобляться для читання і запису повідомлень адресної книги (а отже, Person і PhoneNumber). Для цього потрібно запустити утиліту protoc на proto-файлі: (рис. 1.6)

Якщо потрібно згенерувати класи Java, то варто використати опцію — java_out. Аналогічні опції передбачено для інших мов, які підтримуються Protobuf.

```
syntax = "proto2";

package tutorial;

option java_multiple_files = true;
option java_package = "com.example.tutorial.protos";
option java_outer_classname = "AddressBookProtos";

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

Рисунок 1.5 - Proto файл (addressbook.proto)

```
protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/addressbook.proto
```

Рисунок 1.6 - Компіляція proto-файлу

### 1.2.4. The Protocol Buffer API

При перегляді згенерованого Java коду можна побачити, що кожному повідомленню з proto-файлу, був згенерований відповідний клас. Кожен клас має власний клас Builder, який використовується для створення екземплярів цього класу.

І повідомлення, і builders мають автоматично згенеровані методи доступу для кожного поля повідомлення; повідомлення мають лише getters, а builders мають і getters, і setters. Нижче наведено деякі з методів доступу для класу Person (для стислості реалізацію не показано). (рис. 1.7)

```
// required string name = 1;
public boolean hasName();
public String getName();

// required int32 id = 2;
public boolean hasId();
public int getId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();

// repeated .tutorial.Person.PhoneNumber
phones = 4;
public List<PhoneNumber> getPhonesList();
public int getPhonesCount();
public PhoneNumber getPhones(int index);
```

Рисунок 1.7 - Методи екземпляра (Person) повідомлення

Person.Builder має ті ж getters та додатково setters (рис. 1.8). Згенеровані класи мають стиль JavaBeans (getters, setters and builder).

```
// required string name = 1;
public boolean hasName();
public java.lang.String getName();
public Builder setName(String value);
public Builder clearName();

// required int32 id = 2;
public boolean hasId();
public int getId();
public Builder setId(int value);
public Builder clearId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();
public Builder setEmail(String value);
public Builder clearEmail();

// repeated .tutorial.Person.PhoneNumber phones =
4;
public List<PhoneNumber> getPhonesList();
public int getPhonesCount();
public PhoneNumber getPhones(int index);
public Builder setPhones(int index, PhoneNumber
value);
public Builder addPhones(PhoneNumber value);
public Builder addAllPhones(Iterable<PhoneNumber>
value);
public Builder clearPhones();
```

Рисунок 1.8 - Методи Person.Builder

### 1.3. Протокол HTTP/2

HTTP/2 забезпечує оптимізований транспорт для семантики HTTP. HTTP/2 підтримує всі основні функції HTTP/1.1, але прагне бути більш ефективним в декількох аспектах. [5]

Основною одиницею протоколу HTTP/2 є кадр. Кожен тип фрейму служить для різних цілей. Наприклад, кадри HEADERS і DATA складають основу HTTP-запиту і відповіді, в той час як інші типи кадрів, такі як SETTINGS, WINDOW_UPDATE і PUSH_PROMISE, використовуються для підтримки інших функцій HTTP/2. [6]

Мультиплексування запитів досягається шляхом асоціювання кожного HTTP-запиту-відповіді з власним потоком. Оскільки потоки значною мірою незалежні один від одного, заблоковані або призупинені запити і відповіді не заважають виконанню інших потоків. [6]

Керування потоками та визначення пріоритетів забезпечують ефективне використання мультиплексованих потоків. Керування потоками допомагає гарантувати, що надсилаються лише ті дані, які можуть бути використані приймаючою стороною. Пріоритезація гарантує, що обмежені ресурси можуть бути виділені в першу чергу найбільш важливим потокам.

HTTP/2 додає новий режим взаємодії, який дозволяє серверу підштовхувати відповіді клієнту (Server push). Серверні підштовхування можуть компенсувати використання мережі та збільшити затримку, дозволяючи серверам спекулятивно надсилати клієнту дані, які вони вважають необхідними. Сервери досягають цього, складаючи запити і надсилаючи їх як кадри PUSH_PROMISE. Потім сервер може відправити відповідь на складений запит в окремому потоці. [6]

Оскільки поля заголовка HTTP, що використовуються для з'єднання, можуть містити багато надлишкових даних, кадри, що їх містять, стискаються. Це особливо сприятливо впливає на розмір типового запиту, оскільки багато запитів можна стиснути в один TCP-пакет.

### 1.3.1. HTTP протокол

HTTP 1.1 перетворився на протокол, який використовується практично для всього в Інтернеті. Величезні інвестиції були зроблені в протоколи та інфраструктуру, які використовують його переваги, до такої міри, що сьогодні часто простіше зробити так, щоб щось працювало на основі HTTP, ніж будувати щось нове самостійно.

#### 1.3.1.1. HTTP протокол є надто великим

Коли HTTP був створений і випущений у світ, він, ймовірно, сприймався як досить простий і зрозумілий протокол, але час довів, що це не так. HTTP 1.0 в RFC 1945 [7] - це 60-сторінкова специфікація, випущена в 1996 році. RFC 2616 [8], що описує HTTP 1.1, був випущений лише через три роки, в 1999 році, і значно збільшився до 176 сторінок. Проте, при формуванні у IETF над оновленням цієї специфікації, вона була розділена і перетворена в шість документів з набагато більшою кількістю сторінок (в результаті чого з'явився RFC 7230 [9]). За будь-якими підрахунками, HTTP 1.1 великий і включає в себе безліч деталей, тонкощів і, що не менш важливо, багато необов'язкових частин.

**1.3.1.2. Кількість функціоналу**

Характер HTTP 1.1, що містив безліч дрібних деталей і опцій, доступних для пізніших розширень, створив програмну екосистему, в якій майже жодна реалізація не реалізує все - і навіть неможливо точно сказати, що таке "все". Це призвело до того, що функції, які спочатку мало використовувалися, мали дуже мало реалізацій, а ті, що були реалізовані, використовувалися дуже мало.

Пізніше це спричинило проблему інтероперабельності, коли клієнти і сервери почали активніше використовувати такі функції. Конвеєризація HTTP є основним прикладом такої функції.

**1.3.1.3. Проблеми TCP**

HTTP 1.1 не може повною мірою скористатися всіма перевагами потужності та продуктивності, які пропонує TCP. Клієнтам і браузерам HTTP доводиться бути дуже винахідливими, щоб знайти рішення, які зменшують час завантаження сторінок.

Інші спроби, які відбувалися паралельно протягом багатьох років, також підтвердили, що TCP не так просто замінити, і тому робота над вдосконаленням як TCP, так і протоколів, що на ньому базуються, продовжується.

Простіше кажучи, TCP можна використовувати краще, щоб уникнути пауз або даремно витрачених інтервалів, які можна було б використати для надсилання або отримання більшої кількості даних.

**1.3.1.4. Head-of-line blocking**

HTTP pipelining - це спосіб надсилання чергового запиту під час очікування відповіді на попередній запит. Це дуже схоже на чергу в банку або в супермаркеті: ви просто не знаєте, чи людина перед вами швидкий клієнт, чи настирливий, який буде чекати цілу вічність, перш ніж він/вона закінчить. Це називається блокуванням у голові черги.

Звичайно, ви можете спробувати вибрати репліку, яка, на вашу думку, є правильною, а іноді навіть почати нову репліку. Але, врешті-решт, ви не можете уникнути прийняття рішення. І коли воно прийняте, ви не можете змінити лінію.

Створення нового рядка також пов'язане з втратою продуктивності та ресурсів, тому його не можна масштабувати до меншої кількості рядків. Ідеального рішення просто не існує.

Навіть сьогодні більшість desktop веб-браузерів за замовчуванням вимикають конвеєризацію HTTP.

**1.3.2. HTTP/2 протокол**

**1.3.2.1. Двійковий протокол**

HTTP/2 - це бінарний протокол, що може здивувати тих, хто має досвід роботи з текстовими інтернет-протоколами. Можна інстинктивно віддати перевагу текстовим протоколам, де можна вручну створювати запити за допомогою telnet або інших подібних засобів. Однак HTTP/2 був розроблений як двійковий протокол з певною метою: спростити процес генерації фреймів.

Однією з головних проблем HTTP 1.1 та інших текстових протоколів є визначення початку і кінця фреймів. Необов'язкові пробіли і відмова від написання одного і того ж декількома різними способами спрощують

реалізацію протоколу. Крім того, в HTTP 1.1 кадри і власне частина протоколу можуть бути змішані, в той час як в двійковому протоколі дуже легко розділити кадри.

Включення стиснення в протокол і часте використання TLS ще більше зменшує потребу в тексті. Дійсно, в багатьох випадках текст може навіть не бути видимим у мережі. Тому користувачам потрібно звикнути до ідеї використання таких інструментів, як Wireshark, для перевірки рівня протоколу HTTP/2. Для налагодження може знадобитися використання таких інструментів, як curl або аналіз мережевих потоків за допомогою дисектора HTTP/2 Wireshark або подібних інструментів.

#### 1.3.2.2. Двійковий формат

HTTP/2 надсилає двійкові кадри. Існують різні типи кадрів, які можна надсилати, і всі вони мають однакові налаштування: Довжина, Тип, Прапори, Ідентифікатор потоку та корисне навантаження кадру.

У специфікації http2 [9] визначено десять різних типів кадрів, і, можливо, два найбільш фундаментальних з них, які відповідають функціям HTTP 1.1 - це DATA і HEADERS. Далі деякі з них будуть розглянуті більш детально.

#### 1.3.2.3. Мультиплексовані потоки

Ідентифікатор потоку пов'язує кожен кадр, надісланий через http2, з "потоком". Потік - це незалежна, двонапрямна послідовність кадрів, якими обмінюються клієнт і сервер в рамках http2-з'єднання.

Одне http2-з'єднання може містити декілька одночасно відкритих потоків, причому будь-яка кінцева точка може чергувати кадри з декількох потоків. Потоки можуть створюватися і використовуватися в односторонньому порядку або спільно клієнтом і сервером, і вони можуть бути закриті будь-якою кінцевою точкою. Порядок, в якому кадри надсилаються в потоці, має важливе значення. Одержувачі обробляють кадри в порядку їх отримання.

Мультиплексування потоків означає, що пакети з багатьох потоків змішуються через одне з'єднання. Два (або більше) окремих потоки даних об'єднуються в один, а потім знову розділяються на іншій стороні.

#### 1.3.2.4. Пріоритети та залежності

Кожен потік також має пріоритет (також відомий як "вага"), який використовується для того, щоб повідомити одноранговому серверу, які потоки вважати найважливішими, якщо існують обмеження ресурсів, що змушують сервер вибирати, які потоки надсилати першими.

За допомогою фрейму PRIORITY клієнт також може повідомити серверу, від якого іншого потоку залежить цей потік. Це дозволяє клієнту побудувати "дерево" пріоритетів, де кілька "дочірніх потоків" можуть залежати від завершення "батьківських потоків".

Ваги пріоритетів і залежності можна динамічно змінювати під час виконання, що має дозволити браузерам зробити так, щоб коли користувачі прокручують сторінку, повну зображень, браузер міг вказати, які зображення є найбільш важливими, або при перемиканні вкладок він міг надати пріоритет новому набору потоків, які раптово опинилися у фокусі уваги.

#### 1.3.2.5. Стиснення заголовків

HTTP - це протокол без стану. Коротко кажучи, це означає, що кожен запит повинен містити стільки деталей, скільки потрібно серверу, щоб обробити цей запит, при цьому серверу не потрібно зберігати багато інформації та мета-даних з попередніх запитів. Оскільки http2 не змінює цю парадигму, він повинен працювати так само.

Це робить HTTP повторюваним. Коли клієнт запитує багато ресурсів з одного і того ж сервера, наприклад, зображення з веб-сторінки, буде велика серія запитів, які виглядатимуть майже однаково. Серія майже однакових речей може бути стиснена.

У той час як кількість об'єктів на веб-сторінці збільшилася (як згадувалося раніше), використання файлів cookie і розмір запитів також зростали з плином часу. Файли cookie також повинні бути включені в усі запити, часто одні й ті ж самі в декількох запитах.

Розміри запитів HTTP 1.1 стали настільки великими, що іноді вони стають більшими за початкове вікно TCP, що робить їх надсилання дуже повільним, оскільки вони потребують повного зворотного шляху, щоб отримати від сервера відповідь ACK до того, як буде надіслано повний запит. Це ще один аргумент на користь стиснення.

**1.3.2.5.1. Стиснення заголовків**

Стиснення HTTPS і SPDY [12] виявилися вразливими до атак BREACH [10] і CRIME [11]. Вставивши в потік відомий текст і з'ясувавши, як це змінює вихідні дані, зловмисник може з'ясувати,  які корисні дані надсилаються.

Виконання стиснення динамічного контенту для протоколу - без того, щоб стати вразливим до однієї з цих атак - вимагає певних роздумів і ретельного розгляду. Це те, що команда HTTPbis спробувала зробити.

Введіть HPACK [13], Header Compression for HTTP/2, який, як випливає з назви, є форматом стиснення, спеціально розробленим для заголовків http2. Основна мета цього формату є ускладнення зловживання стисненням.

За словами Роберто Пеона (одного з творців HPACK [9]):

"HPACK був розроблений для того, щоб ускладнити витік інформації для відповідної реалізації, зробити кодування і декодування дуже швидким/дешевим, забезпечити контроль одержувача над розміром контексту стиснення, дозволити переіндексацію проксі (тобто, спільний стан між фронтендом і бекендом в межах проксі), а також для швидкого порівняння рядків, закодованих Хаффманом".

**1.3.2.6. Скасування запиту**

Одним з недоліків HTTP 1.1 є те, що коли HTTP-повідомлення було відправлено з довжиною контенту певного розміру, то його не можна зупинити. Звичайно, часто (але не завжди) можливо розірвати TCP-з'єднання, але це призводить до того, що доведеться знову здійснювати нове TCP-рукостискання.

Кращим рішенням буде просто зупинити повідомлення і почати заново. Це можна зробити за допомогою кадру RST_STREAM у http2, який допоможе запобігти марному використанню пропускної здатності і необхідності розривати з'єднання.

#### 1.3.2.7. Server push

Ця функція також відома як "виштовхування кешу". Ідея полягає в тому, що якщо клієнт запитує ресурс X, сервер може знати, що клієнт, ймовірно, захоче ресурс Z, і надсилає його клієнту без запиту. Це допомагає клієнту, поміщаючи Z у свій кеш, щоб він був там, коли йому буде потрібно.

Виштовхування сервером - це те, що клієнт повинен явно дозволити серверу. Варто зазначити, що клієнт у будь-який момент може швидко завершити потік, що виштовхується, за допомогою RST_STREAM, якщо йому не потрібен певний ресурс.

#### 1.3.2.8. Контроль потоку

Кожен окремий потік http2 має власне оголошене вікно потоку, в яке інший кінець може надсилати дані. Це дуже схоже на логіку утиліти SSH.

Для кожного потоку обидва кінці повинні повідомити одноранговому вузлу, що у нього достатньо місця для обробки вхідних даних, а інший кінець має право надсилати стільки даних, скільки потрібно, доки вікно не буде розширено. Потоком керують лише кадри DATA.

### 1.4. Висновки до першого розділу.

Розділ охоплює три ключові аспекти протоколу gRPC: Віддалений виклик процедур (RPC), буфери протоколу (Protobuf) та HTTP/2. Кожна з цих

функцій відіграє вирішальну роль у забезпеченні ефективного та масштабованого зв'язку між клієнтськими та серверними додатками.

Першою функцією є RPC, яка є основою протоколу gRPC. RPC дозволяє клієнтам викликати методи або процедури на віддалених серверах, абстрагуючись від складнощів мережевого зв'язку. gRPC використовує високопродуктивну двонаправлену модель потокової передачі даних для RPC, що забезпечує асинхронний зв'язок між клієнтами і серверами в режимі реального часу.

Другою функцією, яку було розглянуто, є буфери протоколу (Protobuf). Protobuf - це ефективний і розширюваний механізм для серіалізації структурованих даних, що залежить від мови. Він забезпечує стислий і незалежний від платформи спосіб визначення схеми даних і генерування коду для різних мов програмування. gRPC використовує Protobuf для визначення сервісних інтерфейсів і типів повідомлень, забезпечуючи легку функціональну сумісність між різними мовами програмування.

Нарешті, розділ заглиблюється в інтеграцію gRPC з HTTP/2, сучасним мультиплексованим протоколом прикладного рівня. HTTP/2 підвищує продуктивність веб-комунікації, дозволяючи надсилати декілька запитів і відповідей через одне постійне з'єднання. gRPC використовує можливості HTTP/2, такі як мультиплексування запитів, стиснення заголовків і підштовхування сервера, для ефективного транспортування RPC-повідомлень, що призводить до зменшення затримок і поліпшення використання мережі.

## 2. Практика використання gRPC у розподілених системах

### 2.1. gRPC: Основні концепції, архітектура та життєвий цикл

gRPC — це платформа, яка надає велику кількість готових рішень для вирішення проблем, які виникають при розробці розподілених систем. Для побудови API, за допомогою якого буде відбуватись комунікація, у gRPC використовується протокол віддаленого виклику процедур (RPC). [14]

Ця платформа має відкриту кодову базою, таким чином розробники можуть досліджувати платформу, виявляти недоліки, обговорювати нові ідеї для покращення системи та змінювати її. [15]

Ліцензія на основі якої регулюється використання цієї платформи має назву “Apache License 2.0”. Ця ліцензія дозволяє використовувати технології для комерційного та приватного використання. [16]

gRPC не був написаний з нуля. Ця технологія була створена на основі іншої, яка мала назву Stubby. У Google цю технологію називали єдиною універсальною інфраструктурою для з'єднання великої кількості мікросервісів. Але у 2015 році Google наважився виділити усі найкращі особливості Stubby у нову платформу та зробити її код відкритим. [17] Це сталося у 23 серпня 2016 року, саме у цей день світ побачила перша версія gRPC. [18]

У складі gRPC можна знайти велику кількість інструментів, які можна комбінувати між собою. Таким чином можна створювати застосунки незалежно від мови програмування, таким чином у вас може бути написана серверна частина додатку на одній мові програмування, а клієнтська частина на іншій. Це дуже розширює можливість для розвитку системи та не зв’язує стек проекту лише з однією мовою програмування. [19]

Також варто виокремити підтримку протоколу HTTP/2. Цей протокол надає можливість реалізовувати двонапрямлений стрімінг даних з мультиплексуванням у рамках лише одного TCP з’єднання. Це TCP з’єднання

може бути перевикористаним для подальшої комунікації, на відміну від минулої версії HTTP/1.1. [20]

Ще однією особливістю gRPC є технологія Protocol Buffers, яка визначає новий формат для серіалізація та десеріалізація даних. Цей формат є кращим, якщо порівнювати його з іншими форматами як: JSON та XML. Новий формат надає можливість  версійності та зворотної сумісності, яким чином зміна API не буде порушувати роботу сервісів, які використовують стару версію. [21]

gRPC є платформою, яка входить до Cloud Native Computing Foundation. Cloud Native Computing Foundation (CNCF) - це фонд програмного забезпечення з відкритими кодовими базами, який сприяє впровадженню хмарних обчислень. Він був заснований у 2015 році як проєкт Linux Foundation для просування контейнерних технологій і спрямування технологічної індустрії на їхній розвиток, CNCF прагне зробити хмарні обчислення універсальними та стійкими. У складі CNCF наявні такі відомі проекти, як: Kubernetes, Prometheus та інші. [22] [23]

### 2.1.1. Загальна інформація

#### 2.1.1.1. Визначення сервісу

Як і багато RPC-систем, gRPC базується на ідеї визначення сервісу, вказуючи методи, які можуть бути викликані віддалено, з їх параметрами та типами повернення. За замовчуванням gRPC використовує буфери протоколу [24] як мову опису інтерфейсу (IDL) для опису як інтерфейсу сервісу, так і структури повідомлень корисного навантаження. За бажанням можна використовувати інші альтернативи.

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

Рисунок 2.1 - Приклад опису сервісу та повідомлення за допомогою Protobuf

gRPC дозволяє визначити чотири типи методів обслуговування:

- Унарні RPC, де клієнт надсилає один запит на сервер і отримує одну відповідь, так само як звичайний виклик функції. (рис. 2.2)

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

Рисунок 2.2 - Унарна RPC

- Серверні потокові RPC, де клієнт надсилає запит на сервер і отримує потік для читання послідовності повідомлень назад. Клієнт читає з повернутого потоку, доки не залишиться більше повідомлень. gRPC гарантує впорядкування повідомлень в межах окремого RPC-виклику. (рис. 2.3)

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

Рисунок 2.3 - Серверна потокова RPC

- Клієнтські потокові RPC, де клієнт пише послідовність повідомлень і надсилає їх на сервер, знову ж таки використовуючи наданий потік. Після того, як клієнт закінчив писати повідомлення, він чекає, поки сервер прочитає їх і поверне свою відповідь. Знову ж таки, gRPC гарантує впорядкування повідомлень в межах окремого RPC-виклику. (рис. 2.4)

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

Рисунок 2.4 - Клієнтська потокова RPC

- Двонапрямні потокові RPC, де обидві сторони надсилають послідовність повідомлень, використовуючи потік читання-запису. Обидва потоки працюють незалежно, тому клієнти і сервери можуть читати і писати в довільному порядку: наприклад, сервер може чекати на отримання всіх повідомлень клієнта, перш ніж писати свої відповіді, або по черзі читати повідомлення, а потім писати повідомлення, або будь-яка інша комбінація читання і запису. Порядок повідомлень у кожному потоці зберігається. (рис. 2.5)

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

Рисунок 2.5 - Двонапрямлена потокова RPC

### 2.1.1.2. Використання API

Починаючи з визначення сервісу в .proto-файлі, gRPC надає плагіни компілятора буфера протоколу, які генерують код на стороні клієнта і сервера. Користувачі gRPC зазвичай викликають ці API на стороні клієнта і реалізують відповідний API на стороні сервера.

- На стороні сервера сервер реалізує методи, оголошені сервісом, і запускає gRPC-сервер для обробки клієнтських викликів. Інфраструктура gRPC декодує вхідні запити, виконує методи сервісу і кодує відповіді сервісу.
- На стороні клієнта клієнт має локальний об'єкт, відомий як заглушка (для деяких мов перевага надається терміну клієнт), який реалізує ті ж методи, що і сервіс. Клієнт може просто викликати ці методи на локальному об'єкті, а методи обертають параметри виклику у відповідний тип повідомлення буфера протоколу, надсилають запити до сервера і повертають відповіді з буфера протоколу сервера.

### 2.1.1.3. Синхронність та асинхронність

Синхронні виклики RPC, які блокуються до отримання відповіді від сервера, є найближчим наближенням до абстракції виклику процедури, до якої прагне RPC. З іншого боку, мережі за своєю суттю є асинхронними, і в багатьох сценаріях корисно мати можливість запускати RPC без блокування поточного потоку.

**2.1.2. Життєвий цикл RPC**

У цьому розділі буде докладно розглянуто, що відбувається, коли gRPC-клієнт викликає метод gRPC-сервера.

**2.1.2.1. Унарна RPC**

Спочатку розглянемо найпростіший тип RPC, де клієнт надсилає один запит і отримує одну відповідь:

1. Коли клієнт викликає метод-заглушку, сервер отримує повідомлення про виклик RPC з метаданими клієнта для цього виклику, ім'ям методу та зазначеним dedline, якщо такий наявний.
2. Після цього сервер може або негайно відправити назад свої власні початкові метадані (які мають бути надіслані перед будь-якою відповіддю), або зачекати на повідомлення із запитом від клієнта. Що станеться раніше, залежить від конкретної програми.
3. Після того, як сервер отримав повідомлення із запитом від клієнта, він виконує всю роботу, необхідну для створення відповіді. Потім відповідь повертається (у разі успіху) клієнту разом з інформацією про стан (код стану і необов'язкове повідомлення про стан) і необов'язковими супутніми метаданими.
4. Якщо статус відповіді є успішним, клієнт отримує відповідь, яка завершує виклик на стороні клієнта.

**2.1.2.2. Серверна потокова RPC**

Серверна потокова RPC схожа на унарну RPC, за винятком того, що сервер повертає потік повідомлень у відповідь на запит клієнта. Після відправлення всіх повідомлень клієнту надсилається інформація про стан сервера (код стану і необов'язкове повідомлення про стан) та необов'язкові додаткові метадані. На цьому обробка на стороні сервера завершується. Клієнт завершує роботу, як тільки отримає всі повідомлення від сервера.

#### 2.1.2.3. Клієнтська потокова RPC

Клієнтська потокова RPC схожа на унарну RPC, за винятком того, що клієнт надсилає серверу потік повідомлень, а не одне повідомлення. Сервер відповідає одним повідомленням (разом з інформацією про стан і необов'язковими додатковими метаданими), зазвичай, але не обов'язково після того, як отримає всі повідомлення клієнта.

#### 2.1.2.4. Двонапрямлена потокова RPC

У двонапрямленій потоковій RPC виклик ініціюється клієнтом, який викликає метод, а сервер отримує метадані клієнта, назву методу та кінцевий термін виконання. Сервер може відправити назад свої початкові метадані або дочекатися, поки клієнт почне передачу повідомлень.

Обробка потоку на стороні клієнта і сервера залежить від конкретної програми. Оскільки ці два потоки є незалежними, клієнт і сервер можуть читати і записувати повідомлення у будь-якому порядку. Наприклад, сервер може чекати, поки не отримає всі повідомлення від клієнта, перш ніж писати свої повідомлення, або сервер і клієнт можуть грати в "пінг-понг" - сервер отримує запит, потім надсилає відповідь, потім клієнт надсилає ще один запит на основі відповіді і так далі.

#### 2.1.2.5. Контроль часу виконання

gRPC дозволяє клієнтам вказувати, як довго вони готові чекати на завершення RPC, перш ніж RPC буде завершено з помилкою DEADLINE_EXCEEDED. На стороні сервера сервер може зробити запит, щоб

дізнатися, чи закінчився час виконання певного RPC, або скільки часу залишилося до завершення RPC.

Вказівка deadline або timeout залежить від мови: деякі мовні API працюють з timeouts (проміжками часу), а деякі API працюють з deadline (фіксованим моментом у часі) і можуть мати або не мати deadline за замовчуванням.

**2.1.2.6. Припинення дії RPC**

У gRPC і клієнт, і сервер роблять незалежні та локальні визначення успішності виклику, і їхні висновки можуть не збігатися. Це означає, що, наприклад, ви можете мати RPC, який успішно завершується на стороні сервера ("Я відправив всі свої відповіді!"), але не завершується на стороні клієнта ("Відповіді надійшли після встановленого мною терміну!"). Також можливо, що сервер вирішить завершити роботу до того, як клієнт надішле всі свої запити.

**2.1.2.7. Скасування RPC**

Клієнт або сервер можуть скасувати RPC в будь-який час. Скасування негайно припиняє RPC, так що ніяка подальша робота не виконується.

**2.1.2.8. Метадані**

Метадані - це інформація про конкретний виклик RPC (наприклад, дані автентифікації) у вигляді списку пар ключ-значення, де ключі - це рядки, а значення - зазвичай рядки, але можуть бути і двійковими даними.

Ключі не чутливі до регістру і складаються з літер ASCII, цифр і спеціальних символів ‘-’, ‘_’, ‘.’ і не повинні починатися з ‘grpc-’ (який зарезервовано для самого gRPC). Двійкові ключі закінчуються на ‘-bin’, а ключі у форматі ASCII - ні.

Метадані, визначені користувачем, не використовуються gRPC, що дозволяє клієнту надавати інформацію, пов'язану з викликом до сервера, і навпаки.

Доступ до метаданих залежить від мови.

#### 2.1.2.9. Канали

Канал gRPC забезпечує з'єднання з gRPC-сервером на вказаному хості та порту. Він використовується при створенні клієнтської заглушки. Клієнти можуть вказувати аргументи каналу, щоб змінити поведінку gRPC за замовчуванням, наприклад, увімкнути або вимкнути стиснення повідомлень. Канал має стан, який може мати значення: connected та idle.

Те, як gRPC поводиться із закриттям каналу, залежить від мови. Деякі мови також дозволяють запитувати стан каналу.

## 2.2. Приклади використання gRPC у реальних проектах

gRPC - це високопродуктивний фреймворк віддаленого виклику процедур (RPC) з відкритим вихідним кодом, розроблений компанією Google. gRPC набув значної популярності в останні роки завдяки своїй ефективності, простоті використання та універсальності. gRPC дозволяє розробникам створювати швидкі, надійні та масштабовані розподілені системи та мікросервіси.

Перша версія gRPC побачила світ у 2016 році. На даний момент цій платформі вже 7 років. Багато компаній успішно інтегрували gRPC у свої системи, досягнувши значного покращення продуктивності та масштабованості. Ось перелік відомих компаній які використовують цю платформу: Google, Spotify, Uber, TikTok, Netflix.

У цьому розділі буде розглянуто, як gRPC використовується в різних проектах на реальних прикладах. Приклади будуть включати у себе створення високопродуктивних API, мікросервісів і розподілених систем в різних доменах. Також будуть розглянуті деякі проблеми, з якими стикаються розробники при використанні gRPC, і як вони їх долають.

### 2.2.1. Використання gRPC в Kubernetes CRI

Архітектура Kubernetes вимагає величезного обсягу внутрішньої активності обміну повідомленнями. gRPC використовується в одній ключовій

області: підтримка управління контейнерами для кожного вузла в кластері Kubernetes. [24]

Kubernetes використовує додаток під назвою kubelet для створення контейнерів на кластері комп'ютерів. Екземпляри kubelet знаходяться на кожній робочій машині в кластері Kubernetes. Коли це потрібно, головний контролер надсилає повідомлення екземпляру kubelet на ідентифікованому комп'ютері з проханням створити певний контейнер в абстрактній організаційній одиниці, яка називається pod.

Саме тут Kubernetes цікавий тим, що підтримує декілька середовищ виконання контейнерів (середовища виконання контейнерів виконують роботу з керування станом та активністю контейнерів, запущених на певній машині) Kubelet не має прямого доступу до середовища виконання контейнерів. Замість цього він звертається до механізму, який називається інтерфейс виконання контейнерів (CRI), який, у свою чергу, звертається до середовища виконання контейнерів. Ця взаємодія між kubelet та CRI відбувається за допомогою gRPC.

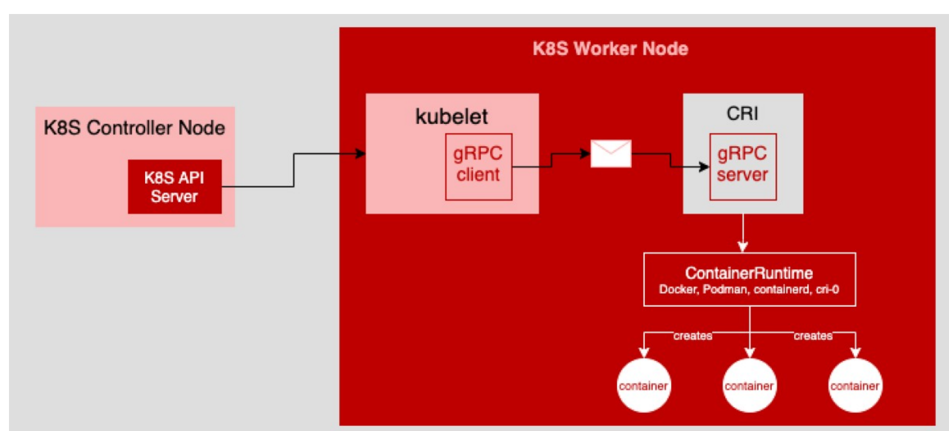


Рисунок 2.6 - Kubernetes використовує gRPC для зв'язку між kubelet та CRI

Кластер Kubernetes зазвичай складається з сотень або тисяч обчислювальних вузлів. Кожен з цих вузлів може динамічно підтримувати сотні, якщо не тисячі, контейнерів (у багатьох випадках контейнери створюються і знищуються на основі вимог додатків або умов несправності). Контейнери постійно створюються і видаляються з дуже високим рівнем

активності. Вся ця активність здійснюється за допомогою повідомлень, що надсилаються та отримуються gRPC. [25]

gRPC швидкий і ефективний: за словами розробників платформи управління API DreamFactory, з'єднання gRPC в 7-10 разів швидше, ніж з'єднання REST API. Така продуктивність робить gRPC придатним для зв'язку в веб-масштабі від kubelet до CRI. Коротше кажучи, gRPC є невід'ємною частиною внутрішньої структури Kubernetes; Kubernetes не може працювати без нього. [25]

### 2.2.2. Уніфікація робочих процесів, написаних кількома мовами програмування

Temporal - це платформа з відкритим вихідним кодом, яка дозволяє архітекторам підприємств підтримувати відмовостійкість виконання бізнес-процесів у разі збою інфраструктури. Наприклад, архітектура ланцюжка поставок повинна гарантувати, що всі запити на закупівлю будуть виконані незалежно від миттєвого збою. Уявіть, що компанія CoolCo регулярно купує товари в іншої компанії AcmeWidgets. Припустимо, що служба виконання замовлень AcmeWidgets переходить в офлайн. Використовуючи технологію Temporal, розробники CoolCo можуть зробити так, що будь-який запит на покупку до AcmeWidgets буде виконано, незважаючи ні на що. Таким чином CoolCo завжди отримує свої товари. [26]

Перевага Temporal полягає в тому, що розробники AcmeWidgets можуть легко реалізовувати закупівельну діяльність різними мовами. Temporal виконує важку роботу, забезпечуючи виконання в разі небезпеки, наприклад, короткочасного відключення мережі або несподіваного сповільнення доступу до диска.

Тепер, де стає цікаво, gRPC є комунікаційною технологією, яка об'єднує всі визначені розробником дії робочого процесу, написані на різних мовах програмування, в загальну архітектуру обміну даними.

Максим Фатєєв, генеральний директор і засновник Temporal, пояснив деталі технології під час інтерв'ю на KubeCon 2021. У прикладі Фатєєва, один розробник може писати код активності на PHP, а інший - на Go. Кожна активність надсилає свої критичні дані gRPC-клієнту, який пересилає їх на gRPC-сервер. Потім сервер gRPC передає всі ці дані з мовних завдань до внутрішньої служби Temporal. Внутрішня служба Temporal виконує необхідну роботу. [26]

Те, що починалося як дані, згенеровані в PHP або Go, перетворюється в загальний формат даних, який є буферами протоколу, форматом серіалізації за замовчуванням для gRPC. Сервер gRPC виконує роботу по перетворенню даних у формат, придатний для використання внутрішніми тимчасовими службами. Використання gRPC у веб-масштабі для підтримки передачі даних, створених різними мовами, є переконливим варіантом використання gRPC.

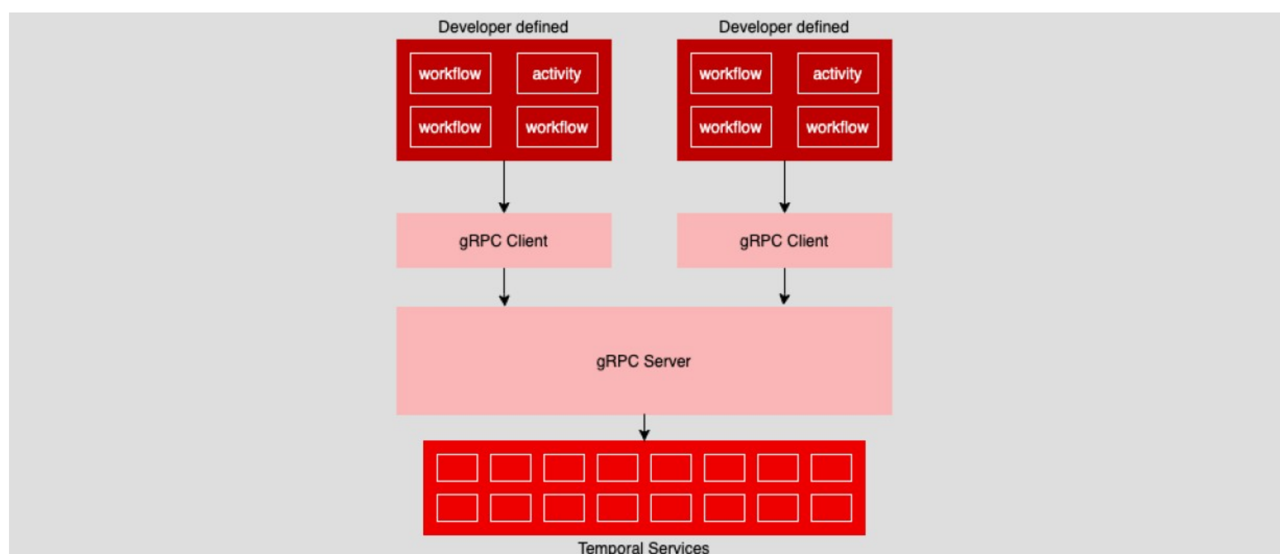


Рисунок 2.7 - Temporal використовує gRPC, щоб уніфікувати робочий процес і діяльність, написані різними мовами, до єдиного формату обміну даними

### 2.2.3. Реалізація потокового gRPC для надання інформації про місцезнаходження транспортного засобу

Однією з цікавих особливостей gRPC є підтримка безперервного двонапрямного потокового передавання даних через HTTP/2. Це робить gRPC придатним для додатків спільного використання автомобілів.

Однією з проблем, з якою стикається каршерінг, є те, як інформувати користувачів про місцезнаходження транспортного засобу на дорозі. На початку розвитку каршерінгу деякі мобільні додатки регулярно (наприклад, щосекунди) зверталися до серверів, щоб перевірити місцезнаходження автомобіля. Це означало встановлення мережевого з'єднання з сервером, запит про місцезнаходження автомобіля, отримання відповіді та закриття мережевого з'єднання. Така мережева взаємодія без статусу лежить в основі HTTP/1.1. Цей підхід є фундаментальним для HTTP/1.1 і виконує свою роботу, але він явно неефективний.

Lyft вирішив застосувати інший підхід, який Майкл Ребелло, iOS-розробник Lyft, пояснив у презентації на Swift User Group у 2018 році: замість того, щоб використовувати опитування для отримання місцезнаходження транспортного засобу, Lyft використовує gRPC щоб надсилати дані про місцезнаходження транспортного засобу в безперервному потоці gRPC-повідомлень. [27]

Це працює завдяки тому, що мобільний додаток підключається до gRPC-сервера і тримає з'єднання відкритим. gRPC-сервери безперервно надсилають повідомлення мобільному додатку через відкрите з'єднання. Кожне повідомлення описує місцезнаходження транспортного засобу, що наближається до клієнта із запитом на поїздку. Мобільний додаток декодує серійні gRPC-повідомлення в буфері протоколу і відображає інформацію на екрані мобільного додатку.

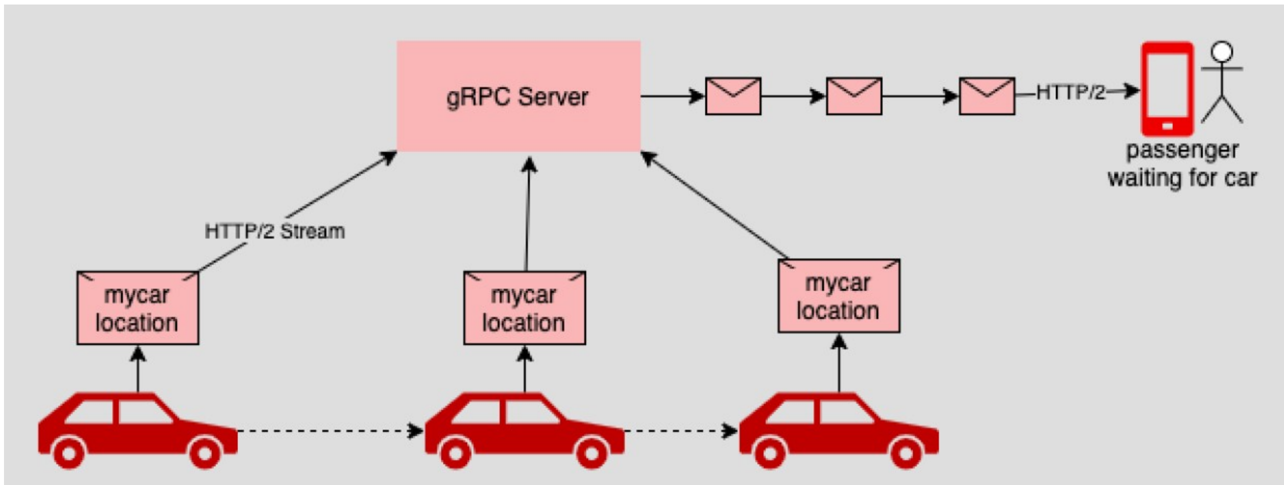


Рисунок 2.8 - Lyft використовує gRPC, щоб забезпечити безперервну передачу інформації про місцезнаходження автомобіля на мобільний телефон водія

Використання потокового gRPC є розумним підходом для повідомлення про місцезнаходження транспортного засобу, але мобільний додаток повинен підтримувати HTTP/2 і вміти декодувати серіалізовані повідомлення у форматі буфера протоколу. Кілька років тому це було б інноваційним рішенням. Однак сьогодні підтримка HTTP/2 широко розповсюджена, а інструменти для кодування та декодування повідомлень у буферах протоколів стають все більш досконалими. Тим не менш, на той час реалізація gRPC в Lyft була інноваційним підходом для підтримки зв'язку в реальному часі між мобільними пристроями. [28]

### 2.2.4. Використання gRPC із зовнішніми веб-клієнтами через проксі-сервер

Використання gRPC в інтерфейсі браузерних веб-додатків може бути складним. Всі поширені браузери стверджують, що підтримують HTTP/2, протокол, з яким працює gRPC, але вони не підтримують gRPC в повному обсязі. Наприклад, gRPC використовує серіалізовані двійкові дані для зв'язку між клієнтом і сервером, тоді як браузерні веб-клієнти надають перевагу текстовим даним. Крім того, реалізація потокової передачі даних, яка є

фундаментальною для gRPC, може бути складною в браузерах. Ці проблеми не є тривіальними.

Тим не менш, використання gRPC з браузера є привабливою ідеєю для багатьох архітекторів, які розробляють високопродуктивні потокові додатки, що вимагають надійної продуктивності. Однак, це не станеться на рівному місці. Потрібне якесь альтернативне рішення. На щастя, воно існує. Рішення полягає у використанні веб-проксі, який діє як посередник між браузером і gRPC-сервером.

Torq використовує цей метод, і за словами Джошуа Торнгрена, віце-президента Torq, компанія пропонує безкодовий підхід до запобігання веб-загрозам для фахівців з безпеки. Конфігурація та автоматизація здійснюється у веб-браузері в поєднанні з gRPC на стороні сервера.

Щоб зробити веб-сторінки сумісними з бекендом gRPC, Torq використовує фреймворк grpc-web з відкритим вихідним кодом. grpc-web дозволяє браузеру підключатися до проксі-сервера в Інтернеті. Проксі-сервер зв'язується з gRPC-сервером; Envoy є найкращим проксі-сервером, який використовується у багатьох реалізаціях grpc-web. [29]

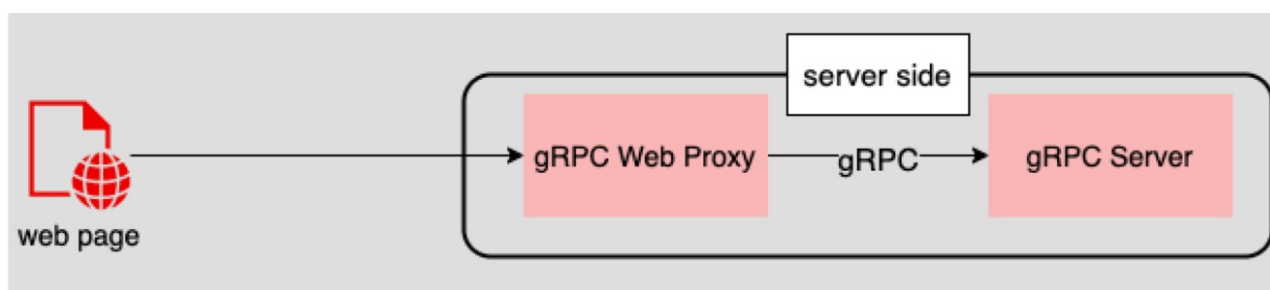


Рисунок 2.9 - Torq підтримує свої браузерні інтерфейси, використовуючи gRPC проксі для зв'язку з бекенд-серверами

Інженери інфраструктури налаштовують проксі Envoy на прослуховування трафіку HTTP/1.1, що надходить від веб-браузерів через певний порт, наприклад, 8080. Потім він перетворює цей трафік у gRPC-сумісну

версію і перенаправляє його на gRPC-сервер, який прослуховує інший порт, наприклад, 9090. Відповіді gRPC-сервера перенаправляються в браузер через проксі-сервер Envoy. [27]

Важливо розуміти, що браузеру не потрібно взаємодіяти з gRPC-сервером. Браузер взаємодіє з проксі, проксі взаємодіє з gRPC-сервером, і навпаки - використання проксі для зв'язку з gRPC-сервером є розумним рішенням дуже складної проблеми для front-end розробників.

### 2.2.5. Проектування граничного gateway, платформи управління життєвим циклом API Uber

#### 2.2.5.1. Еволюція API-Gateway Uber

У жовтні 2014 року Uber розпочав свій шлях масштабування, що призвело до одного з найбільш вражаючих етапів зростання компанії. Відтоді компанія щомісяця нелінійно масштабує свою інженерну команду, залучаючи мільйони користувачів по всьому світу. У цій частині будуть висвітлені різні етапи розвитку API-Gateway Uber, який лежить в основі продуктів Uber. Буде описано еволюцію протягом двох поколінь системи gateway, досліджено її виклики та обов'язки.

#### 2.2.5.2. Перше покоління: органічний розвиток

Дослідження архітектури Uber, проведене у 2014 році, дозволило виокремити два ключові сервіси: диспетчерський та api. Диспетчерська служба відповідала за з'єднання пасажира з водієм, а API-сервіс був довгостроковим сховищем користувачів і поїздок. Крім того, існувала однозначна кількість мікросервісів, які підтримували критичні потоки у користувацькому додатку. [30]

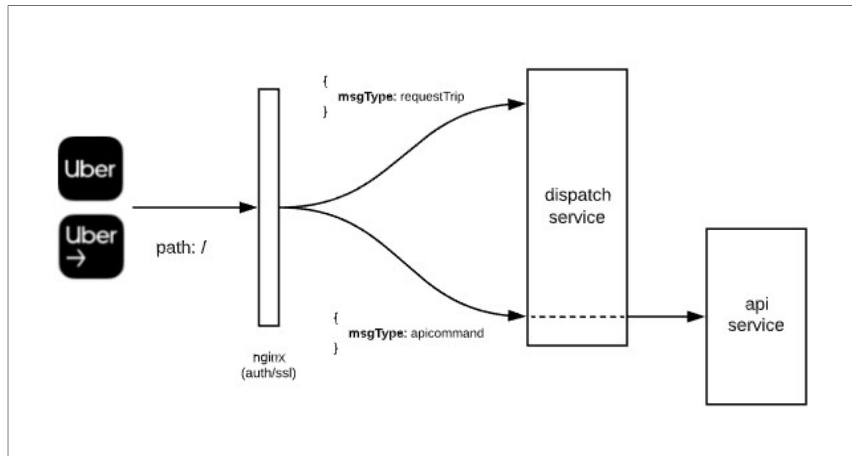


Рисунок 2.10 - Спрощена ілюстрація високого рівня архітектури першого покоління

І додаток пасажира, і додаток водія підключалися до диспетчерської служби за допомогою однієї кінцевої точки, розміщеної на "/". Тіло кінцевої точки мало спеціальне поле "messageType", яке визначало RPC-команду для виклику конкретного обробника. Обробник відправляв відповідь у форматі JSON.

Серед набору RPC-команд 15 були зарезервовані для критично важливих операцій в режимі реального часу, таких як дозвіл партнерам-водіям починати приймати рейси, відхиляти рейси та запитувати рейси для пасажирів. Спеціальний тип повідомлення отримав назву "ApiCommand", який переадресовував усі запити до api-сервісу з деяким додатковим контекстом від диспетчерської служби.

У контексті API-шлюзу це виглядало б так, ніби "ApiCommand" був нашим шлюзом до Uber. Перше покоління - це результат органічної еволюції єдиного монолітного сервісу, який почав обслуговувати реальних користувачів і знайшов спосіб масштабуватися за допомогою додаткових мікросервісів. Диспетчерська служба слугувала мобільним інтерфейсом із загальнодоступними API, але включала в себе диспетчерську систему з відповідною логікою та проксі для перенаправлення всього іншого трафіку на інші мікросервіси всередині Uber. [30]

Після цього дні слави цієї системи першого покоління тривали недовго, оскільки вона вже була у виробництві протягом кількох попередніх років. У січні 2015 року було створено проект абсолютно нового API-gateway, і перший семантично RESTful API, що дозволяв додатку для водіїв Uber шукати місце призначення, був розгорнутий з кількома тисячами запитів в секунду (QPS).

**2.2.5.3. Перше покоління: органічний розвиток**

Uber прийняв мікросервісну архітектуру на самому початку своєї діяльності. Це архітектурне рішення призвело до того, що до 2019 року було створено понад 2200 мікросервісів, які стали основою всіх продуктів Uber.

API Gateway отримав назву RTAPI, скорочення від Real Time-API. Він розпочався з одного RESTful API на початку 2015 року і перетворився на шлюз з багатьма загальнодоступними API, що забезпечують роботу понад 20 зростаючих мобільних додатків і веб-клієнтів. Послуга являла собою єдиний репозиторій, який був розбитий на кілька спеціалізованих груп розгортання, оскільки він продовжував зростати в геометричній прогресії. [30]

Цей API-Gateway був одним з найбільших NodeJS-додатків в Uber з вражаючою статистикою:

- Багато кінцевих точок. Вони згруповані  у 110 логічні групи.
- 40% інженерного коду було присвячено цьому рівню.
- 800 000 запитів на секунду на піку.
- 1,2 мільйона перекладених рядків.
- 50 000 інтеграційних тестів, які у середньому виконуються за 5 хвилин
- Розгортання виконується майже кожен день за довгому проміжку часу (4-9 місяців).
- ~1 млн рядків коду, що обробляє найважливішу бізнес-логіку користувачів
- ~20% мобільної збірки — це автогенерований код, згенерований на основі схем.

- Комунікація з ~400+ додатковими сервісами, які належать 100+ командам в Uber

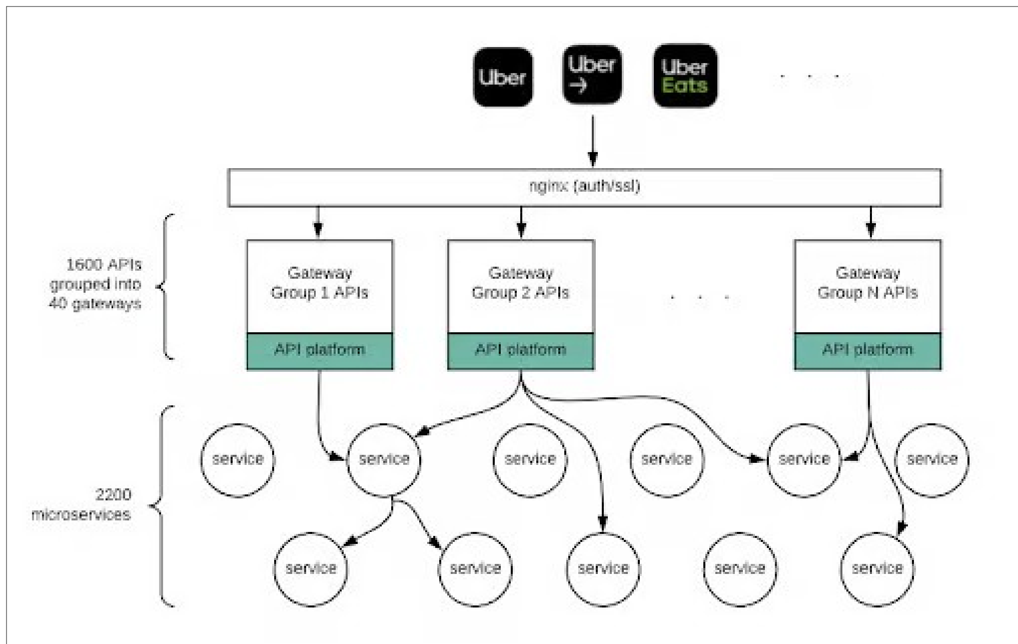


Рисунок 2.11 - Високорівнева ілюстрація сервісу RTAPI як частини загального стеку компанії

### 2.2.5.4. Друге покоління: технічні труднощі

Початкові цілі для gateway були здебільшого пов'язані з input/output, і команда була присвячена дослідженню Node.js. Після низки оглядів, Node.js став ключовою технологією для цього gateway. З часом виникали все більші проблеми з такою динамічною технологією та забезпеченням простору для вільного кодування для 1500 інженерів на такому важливому рівні архітектури Uber. [30]

У певний момент, коли 50 000 тестів запускалися на кожну нову зміну API/коду, стало складно створити надійний фреймворк для інкрементального тестування на основі залежностей з деякими механізмами динамічного завантаження. Оскільки інші підрозділи Uber перейшли на Golang і Java як основні підтримувані мови, залучення нових бекенд-інженерів до роботи над gateway і його асинхронними шаблонами Node.js сповільнило роботу інженерів.

Gateway став досить великим. Він набув статусу моно-репозиторію (gateway був розгорнутий як 40+ незалежних сервісів), а оновлення 2500 npm бібліотек до новішої версії Node.js продовжувало збільшувати зусилля в геометричній прогресії. Це означало, що використання нових версій багатьох бібліотек було неможливим. У цей час Uber почав використовувати gRPC як протокол комунікації. [30]

#### 2.2.6. Міграція до gRPC у компанії Dropbox

У компанії Dropbox працюють сотні сервісів, написаних різними мовами, які обмінюються мільйонами запитів за секунду. В основі сервісно-орієнтованої архітектури лежить Courier, це фреймворк віддаленого виклику процедур (RPC) на основі gRPC. Розробляючи Courier, технічна база була доповнена інформацією про розширення gRPC, оптимізацію продуктивності для масштабування та забезпечення зворотної сумісності з застарілими системами на основі RPC.

У цьому розділі буде розглянуто шлях, який пройшли розробники компанії Dropbox при переході на gRPC.

##### 2.2.6.1. Шлях до gRPC

Перший RPC-фреймворк Dropbox - це не Courier. Перш ніж серйозно розбити свій моноліт, на основі Python, на сервіси, потрібен був міцний фундамент для міжсервісної комунікації. Вибір фреймворку RPC мав глибокі наслідки для надійності.

У минулому Dropbox експериментував з кількома фреймворками RPC. Спочатку використовувався власний протокол для ручної серіалізації та десеріалізації. Apache Thrift [49] використовувався деякими сервісами, такими як конвеєр журналів на основі Scribe. Однак, основний фреймворк RPC, так званий застарілий RPC, використовував протокол на основі HTTP/1.1 з повідомленнями, закодованими у вигляді protobuf.

Для нового фреймворку було розглянуто декілька варіантів. Застарілий фреймворк RPC можна було розвинути, включивши в нього Swagger (тепер OpenAPI [50]), або ж створити абсолютно новий стандарт. Було також взято до уваги можливість побудови на основі Thrift та gRPC.

Було прийнято рішення зупинитися на gRPC в першу чергу через його здатність включати існуючі protobuffs. Крім того, значну роль відіграла привабливість мультиплексування HTTP/2 і двонапрямного потоку для їхніх сценаріїв використання.

Варто зазначити, що якби fbthrift [51] був доступний на той час, можна було б детальніше розглянути рішення на основі Thrift.

**2.2.6.2. Що Courier привносить в gRPC?**

Courier - це не просто інший протокол RPC, це інтеграція gRPC з інфраструктурою Dropbox. Ця інтеграція передбачає роботу з певними версіями автентифікації, авторизації та виявлення сервісів, а також безперешкодну інтеграцію зі статистикою, реєстрацією подій та інструментами відстеження. Кульмінацією цих зусиль є Courier.

Хоча в деяких конкретних випадках використання Bandaid [52] використовується як gRPC-проксі, більшість сервісів Dropbox взаємодіють безпосередньо один з одним, щоб мінімізувати вплив RPC на затримку при обслуговуванні.

Мета полягає в тому, щоб зменшити кількість написаного шаблонного коду. Оскільки Courier слугує загальним фреймворком для розробки сервісів, він включає в себе основні функції, яких потребують усі сервіси. Більшість цих функцій увімкнено за замовчуванням, і ними можна керувати за допомогою аргументів командного рядка. Крім того, деякі функції можна динамічно вмикати і вимикати за допомогою прапорця функції.

**2.2.6.2.1. Безпека: ідентифікація сервісу та взаємна автентифікація TLS**

Стандартний механізм ідентифікації сервісу реалізовано в Courier. Усім серверам і клієнтам призначаються TLS-сертифікати, які видаються внутрішнім центром сертифікації. Кожен сертифікат кодує унікальний ідентифікатор для сервера або клієнта, що полегшує взаємну аутентифікацію між ними.

На стороні TLS, де підтримується контроль над обома кінцями зв'язку, застосовуються суворі налаштування за замовчуванням. Шифрування з ідеальною прямою секретністю (PFS) є обов'язковим для всіх внутрішніх RPC. Версія TLS повинна бути не нижче 1.2. Крім того, симетричні та асиметричні алгоритми обмежені безпечною підмножиною, перевага надається ECDHE-ECDSA-AES128-GCM-SHA256.

Після підтвердження особи і розшифровки запиту сервер переходить до перевірки наявності у клієнта необхідних дозволів. Списки контролю доступу (ACL) та обмеження швидкості можуть бути налаштовані на рівні послуг та методів. Ці конфігурації можна оновлювати через розподілену файлову систему конфігурацій (AFS), що дозволяє власникам сервісів швидко розвантажувати трафік без необхідності перезапуску процесів. Фреймворк Courier обробляє підписку на сповіщення та керує оновленнями конфігурацій.

Сервіс "Ідентифікація" слугує універсальним ідентифікатором для ACL, обмеження швидкості, статистики та інших функцій. Крім того, він пропонує криптографічний захист як додаткову перевагу.

**2.2.6.2.2. Observability: статистика та відстеження**

Використовуючи простий ідентифікатор, можна легко отримати інформації для відлагодження.

Завдяки процесу генерації коду, статистика для кожної послуги та методу автоматично додається як для клієнтів, так і для серверів. Серверна статистика додатково класифікується на основі ідентифікації клієнта, що дозволяє точно визначити навантаження, помилки та затримки для будь-якої кур'єрської служби.

Статистика кур'єрської служби охоплює доступність і затримку на стороні клієнта, а також частоту запитів на стороні сервера і розмір черги. Крім того, доступні такі розбивки, як гістограми затримок для кожного методу або рукостискання TLS для кожного клієнта. Перевагою кастомної генерації коду є можливість статичної ініціалізації цих структур даних, включаючи гістограми та відрізки трасування, тим самим мінімізуючи вплив на продуктивність.

На відміну від застарілого RPC, який лише поширював request_id через межі API, Courier представляє API, заснований на підмножині специфікації OpenTracing. Таким чином були розроблені власні клієнтські бібліотеки, в той час як серверна частина покладається на Cassandra та Jaeger.

Крім того, трасування дозволяє генерувати граф залежностей сервісу під час виконання, що дає змогу інженерам зрозуміти транзитивні залежності сервісу. Вона також може слугувати для перевірки після розгортання, щоб уникнути ненавмисних залежностей.

**2.2.6.2.3. Надійність: deadlines та circuit-breaking**

Courier надає централізоване місце для мовно-специфічних реалізацій функціональності, яка є спільною для всіх клієнтів, наприклад, тайм-аутів. З часом до цього рівня було додано численні можливості, часто в результаті дій, виявлених під час розтину.

Кожен gRPC-запит включає в себе deadline, який вказує на тривалість, протягом якої клієнт готовий чекати на відповідь. Завдяки автоматичному поширенню відомих метаданих за допомогою заглушок Courier, deadline супроводжує запит, навіть коли він перетинає межі API. У процесі deadline перетворюються у власне представлення. Наприклад, у Go вони представлені об'єктом context.Context, отриманим за допомогою методу WithDeadline.

У практичному застосуванні було ефективно вирішено цілі класи проблем надійності шляхом обов'язкового визначення deadline у визначеннях сервісів.

Примітно, що цей контекст може навіть виходити за межі рівня RPC. Наприклад, застаріла MySQL ORM серіалізує контекст RPC і deadline в коментар в SQL запиті. Аналізуючи ці коментарі, SQLProxy може припиняти запити, які перевищують deadline. Цей підхід також забезпечує атрибуцію для кожного запиту при налагодженні запитів до бази даних.

Іншою поширеною проблемою, яку вирішують застарілі RPC-клієнти, є реалізація користувацького експоненціального відступу і Jitter при повторних спробах. Це часто необхідно, щоб запобігти каскадним перевантаженням, які впливають на інші сервіси.

У Courier мета полягала в тому, щоб вирішити проблему обриву ланцюга в більш загальному вигляді. Для цього було запроваджено чергу LIFO (Last-In-First-Out) між слухачем і робочим пулом. (рис. 2.12)

Під час перевантаження сервісу ця черга LIFO функціонує як автоматичний вимикач. Черга обмежена не тільки розміром, але й часом. Отже, запит може перебувати в черзі лише обмежений проміжок часу.

В той час як LIFO має недолік потенційного переупорядковування запитів, CoDel (контрольована затримка) може бути використана для збереження порядку в черзі. CoDel також має властивості розриву ланцюга, але не втручається в порядок запитів. (рис. 2.13)

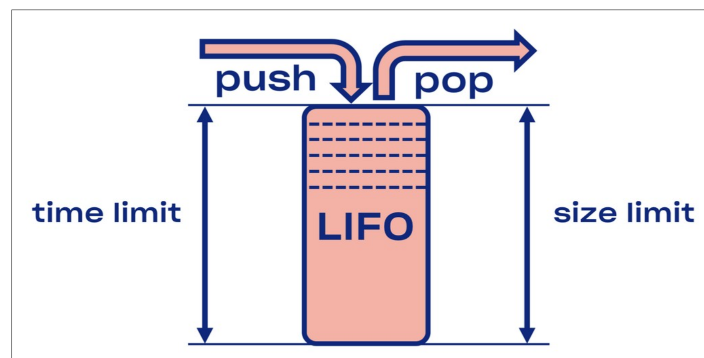


Рисунок 2.12 - Черга LIFO

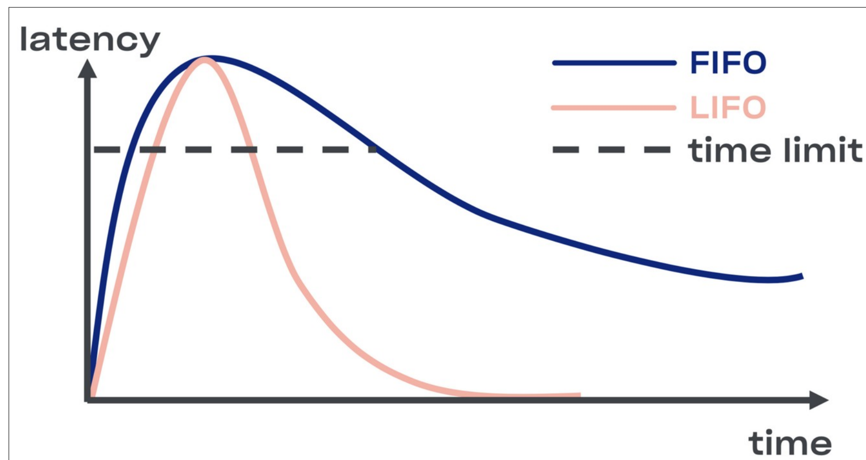


Рисунок 2.13 - Зростання затримки при різних типах черг

### 2.2.6.2.4. Introspection: налагодження кінцевих точок

Хоча налагоджувальні кінцеві точки не є невід'ємною частиною самого Courier, вони отримали широке розповсюдження в Dropbox через їхню значну корисність. Тому важливо визнати їх важливість. Тут наводиться кілька прикладів, що підкреслюють їхню цінність у наданні можливостей для самоаналізу.

З міркувань безпеки бажано виставляти ці кінцеві точки на окремий порт (потенційно обмежений інтерфейсом зі зворотним зв'язком) або навіть на Unix-сокет (з додатковим контролем доступу за допомогою дозволів на файли в Unix). Крім того, слід серйозно розглянути можливість використання взаємної автентифікації TLS, що вимагає від розробників пред'являти свої сертифікати для доступу до кінцевих точок налагодження, особливо тих, що не призначені тільки для читання.

Можливість отримати уявлення про стан виконання є надзвичайно цінною функцією налагодження. Наприклад, виставляння профілів купи і процесора як кінцевих точок HTTP або gRPC може бути надзвичайно корисним. Наразі планується використовувати цю можливість для тестування методом канарки, щоб автоматизувати порівняння різниці між процесором і пам'яттю між старою і новою версіями коду.

Кінцеві точки налагодження також мають потенціал для зміни стану виконання. Наприклад, сервіс на основі мови програмування Go дозволяє динамічно змінювати налаштування GCPercent.

Для авторів бібліотек автоматичний експорт специфічних для бібліотеки даних до кінцевого пункту RPC може бути дуже корисним. Яскравим прикладом є можливість вивантаження внутрішньої статистики бібліотеки malloc. Іншим прикладом є надання налагоджувальної кінцевої точки для читання/запису для модифікації рівня журналювання сервісу на льоту.

Враховуючи, що усунення несправностей у зашифрованих та двійково-кодованих протоколах може бути складним, бажано включити якомога більше інструментів, наскільки це дозволяє продуктивність, у сам рівень RPC. Прикладом такого API для самоаналізу є нещодавня пропозиція щодо каналів у gRPC. [53]

Можливість перегляду параметрів на рівні програми також може виявитися корисною. Яскравим прикладом є узагальнена кінцева точка інформації про програму, яка надає такі деталі, як хеш збірки/джерела та інформацію про командний рядок. Ця кінцева точка може бути використана системою оркестрування для перевірки узгодженості розгортання служби.

**2.3. Проблеми розподілених систем**

Створення розподілених систем - нелегка справа. Навіть "проста" система, яка складається лише з пари вузлів, все все одно породжує проблеми. Ці вузли повинні взаємодіяти через мережу. Однак один з них може вийти з ладу, або сама мережа може стати недоступною, або породжувати велику затримку.

При проектуванні розподіленої системи важливо пам'ятати, що не варто припускати, що все завжди буде йти за планом. Потрібно усвідомлювати, що існують обмеження та перешкоди, які потрібно буде подолати. [31]

Щоб краще зрозуміти виклики, які виникають при розробці надійних розподілених систем, варто звернутися до помилок розподілених обчислень — списку хибних припущень, які можуть зробити архітектори та розробники: [32]

1. Мережа надійна.
2. Затримка дорівнює нулю.
3. Пропускна здатність безмежна.
4. Безпека мережі.
5. Топологія не змінюється.
6. Наявний лише адміністратор.
7. Транспортні витрати дорівнюють нулю.
8. Мережа є однорідною.

Цей список був сформований групою розробників на протязі 7 років, у різні моменти протягом цього інтервалу (1990 — 1997). Розробники які брали участь у формуванні: Bill Joy, Dave Lyon, James Gosling, Peter Deutsch.

У цьому розділі буде детально розглянута кожна з помилок проектування, висвітлено проблеми, до яких помилки можуть призвести, а також те, яких заходів можна вжити, щоб пом'якшити наслідки.

**2.3.1. Мережа надійна**

Для розгляду цієї помилки варто визначити таку сутність, як “надійність”. У цій роботі надійність обґрунтовується як ступінь, до якого продукт або послуга відповідає своїм специфікаціям під час використання, навіть у випадку збоїв. Таким чином, надійність можна уявляти як якість часу безвідмовної роботи - тобто гарантію того, що функціональність і користувацький досвід кінцевого користувача зберігаються максимально ефективно.

Тепер повернемося до помилки. Мережі є складними, динамічними і часто непередбачуваними. Багато причин можуть призвести до збою в роботі мережі або проблем, пов'язаних з мережею: комутатор або відключення

живлення, неправильна конфігурація, недоступність цілого дата-центру, DDoS-атаки і т.д. Через таку складність і загальну непередбачуваність мережі є ненадійними. [33]

На цьому моменті варто відповісти на питання: “Як зробити розподілену систему надійною, гарантуючи, що вона продовжуватиме працювати як очікувалося, навіть в умовах ненадійних мереж?”

Вкрай важливо прийняти збій і ставитися до нього як до чогось само собою зрозумілого. Система має бути спроектована таким чином, щоб вона була здатна пом'якшити збої, які неминуче відбудуться, і продовжувати працювати належним чином, незважаючи на несприятливі умови.

Конкретно, з точки зору інфраструктури, потрібно спроектувати систему так, щоб вона була відмовостійкою і з високим ступенем резервування.

На додаток до інфраструктурних проблем, також потрібно подумати про обриви з'єднань, втрату повідомлень і викликів API через збої в мережі. Для деяких випадків використання (наприклад, чат-додаток в режимі реального часу) цілісність даних має важливе значення, і всі повідомлення повинні доставлятися точно один раз та правильним користувачам. Це має відбуватись постійно (навіть у разі збоїв). Щоб забезпечити цілісність даних, система повинна мати характеристики стану. Крім того, потрібні такі механізми, як автоматичне перепідключення і повторні спроби, дедуплікація (або ідемпотентність), а також способи забезпечення впорядкування повідомлень і гарантії доставки.

### 2.3.2. Затримка дорівнює нулю

Затримка може бути близькою до нуля, коли інфраструктура працюєте у локальному середовищі. Однак у глобальній мережі затримка швидко збільшується. Це пов'язано з тим, що у глобальній мережі даним часто доводиться передаватися далі від одного вузла до іншого, оскільки мережа може

охоплювати великі географічні регіони (це зазвичай відбувається у випадку з великомасштабними розподіленими системами).

Проектуючи систему, варто пам'ятати, що затримка є невід'ємним обмеженням мереж. Ніколи не слід вважати, що між відправленням та отриманням даних не буде затримки або нульової затримки.

Затримка в першу чергу обмежується відстанню та швидкістю світла. Звісно, з останньою нічого не можемо вдіяти. Навіть в теоретично ідеальних умовах мережі пакети не можуть перевищувати швидкість світла. Однак покращення можна зробити, коли мова йде про відстань: наблизити дані до клієнтів за допомогою периферійних обчислень. Якщо у завданні йдеться про побудову хмарової системи, слід ретельно вибирати зони доступності, забезпечуючи їхню близькість до клієнтів, і відповідно маршрутизувати трафік.

При розгляді проблеми затримки, варто поміркувати над наступними речами:

- Кешування. Кешування браузера може допомогти поліпшити затримку і зменшити кількість запитів, що надсилаються на сервер. Інколи можна скористатись CDN для кешування ресурсів у різних місцях по всьому світу. Після кешування вони можуть бути отримані через центр обробки даних або точку присутності, найближчу до клієнта (на відміну від обслуговування сервером-джерелом).
- Використання протоколу, керованого подіями. Залежно від характеру варіанта використання, можливо розглянути можливість використання комунікаційного протоколу, такого як WebSockets. Порівняно з HTTP, WebSockets має значно менший час в обидва боки (після встановлення з'єднання). Крім того, з'єднання WebSocket залишається відкритим, що дозволяє передавати дані між сервером і клієнтом, як тільки вони стають доступними, в режимі реального часу.

- Продуктивність сервера. Існує сильна кореляція між продуктивністю сервера (швидкість обробки, використовуване обладнання, доступна оперативна пам'ять) і затримкою. Щоб запобігти перевантаженню мережі та ваших серверів, вам потрібна можливість (динамічно) збільшувати пропускну здатність вашого серверного рівня та перерозподіляти навантаження.

**2.3.3. Пропускна здатність нескінченна**

У той час як затримка - це швидкість, з якою дані передаються з точки А в точку Б, пропускна здатність - це обсяг даних, який може бути переданий з одного місця в інше за певний проміжок часу.

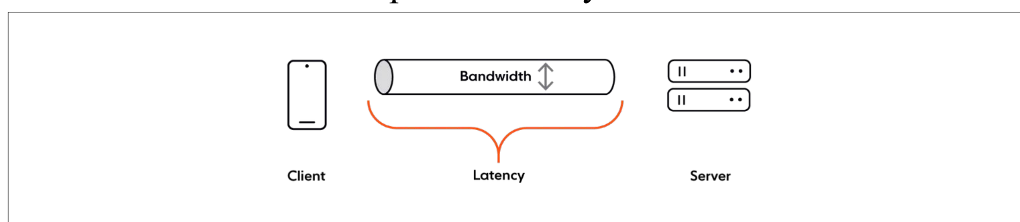


Рисунок 2.14 - Пропускна здатність

Комп'ютерні технології, без сумніву, досягли значного прогресу з 90-х років, коли були винайдені 8 помилок. Пропускна здатність мереж також покращилася, і тепер можливо надсилати більше даних через наші мережі. [32]

Однак, навіть з урахуванням цих удосконалень, пропускна здатність мереж не безмежна (частково тому, що генерування та споживання даних також зросли). Коли великий обсяг даних намагається пройти через мережу, а пропускна здатність недостатня, можуть виникнути різні проблеми:

- Затримки в черзі, вузькі місця та перевантаження мережі.
- Втрата пакетів, що призводить до погіршення гарантій якості обслуговування, тобто повідомлення губляться або доставляються з перебоями.
- Жахлива продуктивність мережі і навіть загальна нестабільність системи.

Існує кілька способів підвищити пропускну здатність вашої мережі. Серед них:

- Комплексний моніторинг. Дуже важливо відстежувати і перевіряти використання мережі, щоб можна було швидко виявити проблеми (наприклад, хто або що займає пропускну здатність) і вжити відповідних заходів для виправлення ситуації.
- Мультиплексування. Такі протоколи, як HTTP/2, HTTP/3 і WebSockets, підтримують мультиплексування - технологію, яка покращує використання пропускної здатності, дозволяючи об'єднувати дані з декількох джерел і відправляти їх по одному каналу/носію зв'язку.
- Полегшені формати даних. Пропускна здатність мережі можна використовувати ефективно за допомогою форматів обміну даними, створені для швидкості та ефективності, такі як JSON. Інший варіант - MessagePack, компактний двійковий формат серіалізації, який створює ще менші повідомлення, ніж JSON.
- Контроль мережевого трафіку. Варто подумати про використання таких механізмів, як дроселювання/обмеження швидкості, контроль перевантажень, експоненціальне відставання тощо.

### 2.3.4. Безпека мережі

Існує багато способів, як мережа може бути атакована або скомпрометована: помилки, вразливості в операційних системах і бібліотеках, незашифрований зв'язок, недогляди, що призводять до доступу до даних неавторизованих осіб, віруси і шкідливе програмне забезпечення, міжсайтовий скриптинг (XSS) і DDoS-атаки, і це лише деякі з них (але цей список нескінченний). [31]

Незважаючи на те, що справжня (абсолютна) безпека в світі розподілених обчислень є помилкою, варто робити все можливе, щоб запобігти порушенням і атакам під час проектування, побудови і тестування системи.

Мета полягає в тому, щоб інциденти, пов'язані з безпекою, траплялися якомога рідше і мали обмежений вплив і наслідки. [32]

Ось кілька речей, які слід врахувати:

- Моделювання загроз. Бажано використовувати структурований процес для виявлення потенційних загроз безпеці та вразливостей, кількісної оцінки серйозності кожної з них і визначення пріоритетів для пом'якшення наслідків атак.
- Глибинний захист. Слід використовувати багаторівневий підхід, з різними перевірками безпеки на рівні мережі, інфраструктури та додатків.
- Мислення безпеки. При розробці системи слід пам'ятати про безпеку і дотримуватися найкращих практик, галузевих рекомендацій та порад.

### 2.3.5. Топологія не змінюється

Топологія мережі - це спосіб, у який зв'язки і вузли мережі розташовані і пов'язані один з одним. У розподіленій системі топологія мережі постійно змінюється. Іноді це відбувається випадково або через проблеми, такі як збій сервера. Інколи це відбувається навмисно - додаються, оновлюються або видавлюються сервери. [33]

Під час проектування розподіленої системи важливо не покладатися на те, що топологія мережі буде незмінною, і не очікувати, що вона завжди буде поводитися певним чином. Існує безліч типів мережевих топологій, які можна використовувати, кожна з яких має свої переваги та недоліки.

### 2.3.6. Наявний лише один адміністратор

У випадку дуже маленької системи може бути лише один адміністратор, або, можливо, в контексті особистого проекту. Крім того, у розподіленій системі, як правило, є більше одного адміністратора майже у всіх реальних сценаріях. Наприклад, варто розглянути сучасні хмарні системи, які складаються з багатьох сервісів, розроблених і керованих різними командами.

При розробці системи, необхідно зробити так, щоб різним адміністраторам було легко (настільки легко, наскільки це можливо) керувати нею. Потрібно подумати про відмовостійкість системи і переконатися, що на неї не вплинуть різні люди, які взаємодіють з нею. Ось кілька речей, які слід врахувати:

- Робота зв’язаних компонентів у роз'єднаному стані. Забезпечення належного роз'єднання дозволяє підвищити відмовостійкість в контексті як запланованих оновлень, що призводять до проблем, так і незапланованих подій, таких як збої. Одним з найпопулярніших варіантів, що полегшує роз'єднання (або послаблення зв'язку), є схема pub/sub.
- Можливість усунення несправностей. Важливо забезпечити видимість системи, щоб адміністратори могли діагностувати та вирішувати проблеми, які можуть виникнути. Повідомлення про помилки і генерування логів у виняткових ситуаціях — ці речі є доволі корисними для формування контексту проблеми. При наявності такого контексту адміністратори з більшою вірогідністю зроблять правильне рішення задля усунення проблем. Крім того, ведення журналів, метрики і трасування (які часто називають трьома стовпами спостережуваності) повинні бути одними з ключових аспектів при проектуванні системи.

**2.3.7. Транспортні витрати дорівнюють нулю**

Так само, як затримка не дорівнює нулю, транспортування даних з однієї точки в іншу має певну вартість, яка зовсім не є незначною.

Перш за все, вартість має мережева інфраструктура. Сервери, мережеві комутатори, балансувальники навантаження, проксі-сервери, брандмауери, експлуатація та обслуговування мережі, забезпечення її безпеки, не кажучи вже про персонал, який забезпечує її безперебійну роботу - все це коштує грошей. Чим більша мережа, тим більші фінансові витрати. [33]

На додаток до фінансів, необхідно враховувати час, зусилля і труднощі, пов'язані з архітектурою розподіленої системи, яка працює через доступну, надійну і відмовостійку мережу. Часто буває менш ризиковано, простіше і економічно вигідніше перекласти ці складнощі на повністю кероване і перевірене рішення, розроблене спеціально для цієї мети.

Окрім інфраструктурних витрат, існують також витрати, пов'язані з транспортуванням даних через мережу. Для переходу від прикладного рівня до транспортного потрібен час і ресурси процесора. Інформація повинна бути оброблена і серіалізована на стороні сервера перед передачею на сторону клієнта, де вона повинна бути десеріалізована. Якщо потрібно зменшити транспортні витрати, варто не використовувати XML або варіантів, заснованих на XML, і використовувати замість них легкі формати серіалізації та десеріалізації, такі як JSON, MessagePack або протокольні буфери (Protobuf).

### 2.3.8. Мережа є однорідною

Інколи навіть домашня мережа не є однорідною. Достатньо лише двох пристроїв з різними конфігураціями (наприклад, ноутбуків або мобільних пристроїв), що використовують різні транспортні протоколи, і у наслідок цього, мережа стає гетерогенною. [32]

Більшість розподілених систем повинні інтегруватися з різними типами пристроїв, адаптуватися до різних операційних систем, працювати з різними браузерами та взаємодіяти з іншими системами. Тому дуже важливо зосередитися на інтероперабельності, гарантуючи, що всі ці компоненти можуть "комунікувати" один з одним, незважаючи на те, що вони відрізняються один від одного.

Необхідно використовувати відкриті, стандартні протоколи, які широко підтримуються, замість пропрієтарних. Наприклад HTTP, WebSockets, SSE або MQTT. Та ж логіка застосовується і до форматів даних, як JSON або MessagePack.

## 2.4. Використання платформи gRPC для вирішення проблем розподілених систем

Для чіткого обґрунтування можливостей платформи gRPC для вирішення проблем розподілених системи розглянути її не як набір інструментів, а як абстрактну специфікацію протоколу RPC. Протокол RPC яка має певні властивості:

Перша властивість - підтримка як одиничних викликів, так і стримінгу. Тобто усі мікросервіси, які реалізовують цю специфікацію, підтримують обидва варіанти.

Друга властивість - наявність додаткової інформації (метаданих), тобто разом з відповіддю на запит  була можливість надати метадані. Під терміном “метадані” зазвичай мають на увазі заголовки (headers).

Третя властивість - підтримка можливості скасування запиту з механізмом timeouts (deadlines).

Четверта властивість - опис повідомлень і самих сервісів здійснюється через Interface Definition Language або IDL.

П’ята властивість - опис wire-протоколу, який взаємодіє з HTTP/2, таким чином gRPC передбачає роботу тільки з HTTP/2.

### 2.4.1. Реалізація специфікації gRPC за замовчуванням

Є типова реалізація gRPC, яка використовується в більшості випадків. Як IDL використовується proto-формат. gRPC плагін для proto-компілятора дозволяє з proto-опису отримувати вихідні коди згенерованих сервісів. І існують runtime-бібліотеки різними мовами - Java, C++, Python. Загалом, практично всі популярні мови підтримуються, тобто для кожної існує runtime-бібліотека. Як повідомлення, якими обмінюються сервіси, використовується proto-повідомлення, стилізовані повідомлення за схемою protobuf [36].

Така конфігурація втілює у собі багато переваг.

#### 2.4.1.1. Сувора типізація

Ця перевага виникає при використанні утиліти Protobuf [36], тобто proto повідомлення завжди має сувору типізацію. При опису будь-якого поля у повідомленні має бути зазначений тип. Типи існують як примітивні, так і строкові, масиви байт. Вони можуть бути скалярні, можуть бути векторні. Також варто відмітити можливість перевикористання вже створених типів. Загалом такий підхід надає можливість описати будь-яку модель.

#### 2.4.1.2. Зворотна сумісність

Варто зауважити, що proto IDL (protobuf) [36] - це формат, у який закладена зворотна сумісність за замовчуванням, тобто protobuf з самого початку його розробки проектувався на основі ідеї зворотної сумісність, і Google випустив версію proto3, яка, порівнюючи з proto2, ще більше покращує зворотну сумісність. На додачу у Google є багато специфікацій, які можна використовувати при проектуванні proto messages на різні випадки не тривіальних архітектурних рішень. Основна мета цих додаткових специфікацій є збереження зворотної сумісності.

#### 2.4.1.3. Значення за замовчуванням

У версії специфікації protbuf proto3 [35] є можливість зазначити значень за замовчуванням. Також усі поля у цій специфікації є опціональними ця саме властивість задовольняє зворотну сумісність, тобто можна додавати нові поля у proto повідомлення і у споживача нічого не потрібно змінювати. Додатковою властивістю є те, що деякі поля у повідомленні можна залишити не заповненими і звернення до віддаленого поля не викликає помилок на клієнті при зверненні до нього. (Хоча це не гарантує того, що відсутність поля не спричинить помилки при роботі мікросервіса, яка наприклад поле яке відсутнє є критичним для виконання бізнес-логіки цього мікросервіса)

#### 2.4.1.4. Автоматична генерація коду

Ще одна перевага реалізації gRPC за замовчуванням — це генерація заглушок для клієнта і сервера за допомогою proto-компілятора і gRPC-плагіну

на основі proto-опису. Таким чином, це полегшує розробку мікросервісів. При автогенерації можна зазначити тип клієнта, який буде згенерований: асинхронний або синхронний, залежно від того, якого роду код ви пишете. Наприклад, при створенні реактивного додатку варто обрати асинхронний клієнт. Така можливість наявна для будь-якої мови. Таким чином, розробникам не потрібно витрачати час на розробку цих клієнтів.

Це також полегшує інтеграцію додатків з окремих систем. Потрібно лише надіслати специфікацію розробникам з іншої системи і вони самі можуть згенерувати клієнти.

#### 2.4.1.5. Документація

Оскільки в IDL для gRPC використовується proto-формат, це звичайний код. При проектуванні proto файлу можна писати коментарі, зокрема дуже розгорнуті. Користувачам мікросервісу буде потрібен proto файл для того, щоб зробити інтеграцію. Proto файл потрапить до користувачів разом з коментарями. Це дуже доволі зручний підхід, коли документація йде поруч із кодом. Це доволі схоже на підхід, який відтворюється у Java, а саме javadocs. [34]

### 2.4.2. Скасування запиту та deadlines

Про скасування запиту і deadline варто зазначити, що запит можна скасувати на сервері і на клієнті. Якщо на якомусь моменті виконання додаток розуміє, що подальша обробка запиту не потрібна, то існує можливість скасувати його.

Також є можливість виставити timeout на запит. У gRPC у більшості runtime-бібліотек як поняття timeout використовується deadline. Але за фактом це те ж саме. Тобто це час, коли запит має завершитися.

Важливою деталлю є те, що сервер може дізнатися, як про скасування запиту, так і про закінчення timeout і перестати виконувати запит на своєму боці.

### 2.4.3. Потокова передача

Цей функціонал є надто передовий, він дозволяє пришвидшити обмін повідомленнями по мережі, але для цього треба мати специфічну архітектуру додатку. Під специфічним мається на увазі не таку на якій прийнято писати додатки. Але іноді є випадки, коли така річ варто реалізовувати навіть в умовах звичайної архітектури. Одним з таких випадків є скачування і завантаження великих обсягів даних. Сервер або клієнт може видавати дані порціями. Групування цих порцій виконується на клієнті чи на сервері. Логіка групування може бути змінена.

Наступна перевага стрімінгу, це прив'язка до однієї машини. При ініціалізації стрімінгу буде встановлено лише одне наскрізне з'єднання (UDP з’єднання) на всі повідомлення всередині потокової передачі (на відміну від TCP з’єднання), і це з'єднання буде прив'язане до однієї машини. З’єднання гарантовано не зміниться.

Якщо з’єднання не зміниться, то існує можливість:

- повторити запит як в одну так і в іншу сторону
- зробити міжсерверну синхронізацію
- реалізувати транзакціональність (реалізувати операції початку, завершення, відкату транзакції)

**2.4.4. Типові завдання**

Існує ряд типових завдань, які доволі легко вирішити за допомогою gRPC.

**2.4.4.1. Обробка помилок**

Існує проблема стандартизації помилок. У gRPC є підхід до уніфікації. Насамперед, коди відповідей, які наявні у runtime-бібліотеках, стандартні.

Наприклад, для Java в разі помилкового статусу викидається виняток. Для C++ статус є просто результатом виконання виклику функції, його можна перевірити і далі вже діяти залежно від нього. Всередині класу google.rpc.Status є 3 поля: код відповіді, повідомлення і деталі. Є стандартний набір кодів

відповіді, які можна використовувати. У поле повідомлення можна просто записати нелокалізоване повідомлення, щоб розібратися з проблемою. Деталі - це вектор, у якому можна передавати кастомні об'єкти, зокрема бінарні. Також варто відзначити набір готових error details, які можна використовувати. [42]

Тут варто поставити запитання: “Чим відрізняється HTTP код від RPC коду?”. Насправді вони не дуже відрізняються. Так само є BadRequest та інші. Але варто притримуватись деталей специфікації. Тобто, якщо сервіс повертає код помилки, то відповідні деталі повинні бути додані до цього коду. Яле дослідження цієї логіки можна звернутись до відповідної таблиці у специфікації. [42]

**2.4.4.2. Трасування запитів**

Трасування теж є типовим завданням. У мікросервісній архітектурі постійно можна побачити ситуацію, коли один виклик породжує 10 інших викликів у 5 інших сервісів. У такій ситуації трапляються помилки і доволі складно зрозуміти на якому сервісі виникла помилка. Для вирішення цих питань є трасування. Існує багато рішень з відкритим кодом, одне з них це застосунок Zipkin [41]. При протоколі HTTP ця бібліотека реалізує ідею трасування через заголовки, а у gRPC через metadata.

Атрибути метаданих можуть бути як рядкові, так і бінарні. У випадку з трасуванням простіше використовувати строкове, тому що якщо стаэться помилка у якомусь клієнті, то легше читати рядок, ніж потім додатково десервізувати бінарні дані.

Модифікацію та додавання метаданих можливо реалізувати через interceptors у runtime-бібліотеках. Для Java це ClientInterceptor і ServerInterceptor. Еквівалентні класи також наявні для інших мов програмування. Крім операцій над мета даними, interceptors застосовують для аутентифікації. Наявна вбудована аутентифікація в gRPC, але вона далеко не

всім підходить, але написання домашнього механізму з подальшим його перевикористанням не є проблемою.

**2.4.4.3. Відлагодження**

Ніби як RPC — бінарний протокол який використовує HTTP/2. Доволі складно відлаголити ці технології окремо, але при їх комбінації все доволі сильно ускладнюється. Але gRPC надає декілька готових інструментів, які полегшують це завдання. А саме це інструмент grpc_cli (аналог curl).

Інструмент є дуже простий та інтуїтивно зрозумілий. Він дає можливість повністю проглядати виклики та створювати їх та відтворювати стримінг.

Також існує інша утиліта — evans. [38] Це інтерактивний CLI: він видає підказки, при заповненні proto повідомлення. Для початку роботи з gRPC він підійде, але для використання у скриптах краще використовувати grpc_cli. [37] У світі HTTP є доволі корисна утиліта Postman [40], для RPC існує аналог — є gRPCox. [39] Він дуже схожий на Postman візуально. Але Postman надає більшу кількість функціоналу. Проте gRPCox надає базовий функціонал і зазвичай цього достатньо.

**2.5. Висновки до другого розділу.**

У розділі було досліджено основні концепції, архітектуру та життєвий цикл gRPC, а також наведено приклади його використання. Крім того, у розділі розглядаються проблеми, що виникають при побудові розподілених систем, і обговорюється, як gRPC може бути застосований для ефективного вирішення цих проблем.

У перший розділі було пояснено фундаментальні поняттями gRPC. Пояснено модель клієнт-сервер і використання буферів протоколу для визначення сервісних інтерфейсів і типів повідомлень. Розділ також охоплює життєвий цикл gRPC-запиту, від клієнта, який робить запит, до сервера, який обробляє його і відповідає на нього. Розуміння цих основних концепцій має вирішальне значення для використання gRPC в розподілених системах.

Другий розділ, представляє практичні приклади використання gRPC в розподілених системах. Він демонструє такі сценарії, як архітектура мікросервісів, міжсервісний зв'язок і потокова передача даних в реальному часі. Приклади підкреслюють переваги gRPC, такі як продуктивність, масштабованість і сумісність мов. Ілюструючи реальні додатки, цей розділ підкреслює практичність використання gRPC в розподілених системах.

Третій розділ, визнає проблеми, з якими стикаються при розробці розподілених систем. У ньому обговорюються такі питання, як виявлення сервісів, балансування навантаження, відмовостійкість і узгодженість даних. Ці проблеми виникають через розподілену природу системи та необхідність обробляти збої і забезпечувати надійність. Цей розділ створює основу для розуміння того, як gRPC може вирішити ці проблеми.

Останній розділ, досліджує, як gRPC може бути застосований для вирішення вищезгаданих проблем. У ньому обговорюються вбудовані функції gRPC, такі як балансування навантаження і перевірка працездатності, які полегшують розробку відмовостійких розподілених систем. Також висвітлюється, як підтримка gRPC двонаправленої потокової передачі даних і асинхронного зв'язку може бути використана для вирішення проблем, пов'язаних з обробкою даних в реальному часі і сценаріями високої пропускної здатності. Розглядаючи застосування gRPC для вирішення проблем розподілених систем, цей розділ демонструє його корисність і універсальність.

## 3. Порівняння gRPC з іншими технологіями

У цьому розділі платформа gRPC буде порівняна з іншими технологіями на основі функції побудови API та реалізації протоколу RPC.

### 3.1. Реалізація протоколу RPC

У цій категорії не так багато технологій, які можуть конкурувати з gRPC (2016), а саме:

- CORBA (Common Object Request Broker Architecture) (1991)
- Java RMI (Remote Method Invocation) (1993)
- Apache Thrift (2012)

#### 3.1.1. CORBA

CORBA (Common Object Request Broker Architecture) — це технологія, яка дозволяє програмним компонентам або об'єктам спілкуватися та взаємодіяти один з одним через мережу. Це технологія проміжного програмного забезпечення, яка забезпечує безперешкодну інтеграцію та сумісність між програмними системами, розробленими на різних мовах програмування і працюючими на різних операційних системах.

Важливо розуміти основні принципи та компоненти CORBA:

- *Object Request Broker (ORB)*. В основі CORBA лежить брокер об'єктних запитів (Object Request Broker), який виступає в ролі посередника між розподіленими об'єктами. Він обробляє зв'язок і передачу повідомлень між об'єктами незалежно від їх фізичного розташування, мови програмування або апаратної платформи.
- *Мова визначення інтерфейсу (IDL)*. Для визначення інтерфейсів об'єктів CORBA використовує нейтральну до мови мову визначення інтерфейсів (Interface Definition Language). IDL дозволяє розробникам описувати методи, властивості та структури даних, які розкривають об'єкти, незалежно від мови програмування, що використовується для їх

реалізації. Інтерфейси IDL слугують контрактом між клієнтами та серверами, визначаючи, як вони можуть взаємодіяти один з одним.

- *Адаптер об'єктів.* Адаптер об'єктів відповідає за відображення комунікаційних запитів між ORB та реальними об'єктами. Він забезпечує рівень абстракції, що дозволяє об'єктам, написаним різними мовами програмування, безперешкодно взаємодіяти.
- *Об'єктні служби*. CORBA надає набір загальних служб, таких як іменування, безпека, сповіщення про події та персистентність, які можуть бути використані розподіленими додатками. Ці сервіси полегшують розробку розподілених систем і сприяють повторному використанню.

CORBA спрощує розробку розподілених додатків, надаючи стандартну і незалежну від платформи комунікаційну інфраструктуру. Вона дозволяє розробникам створювати складні системи, збираючи багаторазові компоненти, розроблені на різних мовах і платформах. CORBA широко використовується в різних галузях, включаючи телекомунікації, фінанси, оборону та корпоративне програмне забезпечення, де інтероперабельність та інтеграція мають вирішальне значення.

Технології CORBA та gRPC варто порівняти на різних рівнях: архітектура, реалізація протоколу, мова опису інтерфейсів, підтримка мов, незалежність від платформи, експертиза та підтримка у спільноті, екосистема та інструментарій.

1. Архітектура:
   - CORBA: CORBA використовує розподілену об'єктно-орієнтовану архітектуру, де доступ до об'єктів здійснюється віддалено за допомогою механізму запитів-відповідей, що забезпечується брокером об'єктних запитів (ORB).

   - gRPC: gRPC використовує клієнт-серверну архітектуру, засновану на віддалених викликах процедур (RPC), де клієнти викликають методи на віддалених серверах і отримують відповіді.
2. Комунікаційний протокол:
   - CORBA: CORBA підтримує декілька комунікаційних протоколів, включаючи IIOP (Internet Inter-ORB Protocol), який є стандартизованим протоколом, заснованим на стандартах Object Management Group (OMG).
   - gRPC: gRPC використовує протокол HTTP/2 як протокол зв'язку за замовчуванням, забезпечуючи такі функції, як двонаправлений потік, мультиплексування та стиснення заголовків.
3. Мова опису інтерфейсів (IDL):
   - CORBA: CORBA використовує мову визначення інтерфейсів (IDL) для визначення інтерфейсів об'єктів і створення специфічних для мови заглушок і скелетів для зв'язку.
   - gRPC: gRPC використовує буфери протоколів (protobuf) як IDL, що дозволяє визначати сервісні методи та структури даних. Інструменти генерації коду створюють специфічні для мови клієнтські та серверні заглушки.
4. Підтримка мов:
   - CORBA: CORBA підтримує широкий спектр мов програмування, включаючи Java, C++, Python та інші. Це сприяє інтероперабельності між компонентами, розробленими різними мовами.
   - gRPC: gRPC пропонує мовну підтримку декількох популярних мов програмування, таких як Java, C++, Python, Go, Ruby та інших. Однак, мовна підтримка може бути не такою широкою, як у CORBA.
5. Незалежність від платформи:

- CORBA: CORBA прагне бути незалежною від платформи, забезпечуючи зв'язок між об'єктами, що працюють на різних операційних системах і апаратних платформах.
- gRPC: gRPC розроблений як крос-платформенний, що дозволяє взаємодіяти між різними системами, незалежно від їх базової платформи.

6. Впровадження та спільнота:
   - CORBA: CORBA існує вже давно і широко використовується в різних галузях. Однак з роками її використання зменшилося, а підтримка спільноти може бути не такою активною, як раніше.
   - gRPC: gRPC набув значної популярності в останні роки, особливо в контексті мікросервісів та хмарних додатків. Він має активну спільноту і підтримується великими технологічними компаніями, такими як Google.
7. Екосистема та інструментарій:
   - CORBA: CORBA має розвинену екосистему з різними інструментами та фреймворками, такими як ORB, компілятори IDL та сервіси проміжного програмного забезпечення. Однак, доступність та підтримка цих інструментів може відрізнятися.
   - gRPC: gRPC користується перевагами зростаючої екосистеми з широким набором інструментів та фреймворків. Він надає багатий набір функцій, включаючи балансування навантаження, автентифікацію та перехоплювачі, що підключаються.

Отже, обидві технології CORBA та gRPC полегшують розподілену комунікацію та інтероперабельність, але gRPC набув більшої популярності в останні роки завдяки своїй простоті, продуктивності та потужній підтримці з боку великих технологічних компаній. У той час як CORBA широко використовується в застарілих системах, gRPC часто надають перевагу

сучасним архітектурам мікросервісів та хмарним додаткам. Однак вибір між ними залежить від конкретних вимог, інфраструктури, а також наявності підтримки мови та платформи.

### 3.1.2. Java RMI

Java RMI (віддалений виклик методів) - це технологія, яка дозволяє об'єктам у віртуальній машині Java (JVM) викликати методи об'єктів, що знаходяться в іншій JVM, можливо, на віддаленій машині. Вона забезпечує розподілену комунікацію та віддалений виклик об'єктів у мові програмування Java.

- *Віддалений доступ до об'єктів.* Java RMI забезпечує віддалений зв'язок між об'єктами Java. Він дозволяє клієнтському об'єкту Java викликати методи на серверному об'єкті Java так, ніби це локальні виклики методів, навіть якщо об'єкти можуть знаходитись у різних JVM.
- *Інтерфейси Java*. RMI використовує інтерфейси Java для визначення віддалених інтерфейсів. Ці інтерфейси визначають методи, які можна викликати віддалено. І клієнт, і сервер повинні мати доступ до цих визначень інтерфейсів.
- *Віддалені об'єкти.* У Java RMI об'єкти, до яких можна отримати віддалений доступ, називаються віддаленими об'єктами. Віддалений об'єкт реалізує віддалений інтерфейс і забезпечує фактичну реалізацію методів, визначених в інтерфейсі.
- *Заглушка та скелет.* RMI використовує заглушки та скелети для полегшення віддаленого виклику методів. Заглушка на стороні клієнта діє як локальний представник віддаленого об'єкта. Коли метод викликається на заглушці, вона обробляє зв'язок з віддаленим об'єктом шляхом впорядкування параметрів, надсилання запиту через мережу та розбиття відповіді. Скелет на стороні сервера отримує запит від заглушки,

розбиває параметри і делегує виклик методу фактичному віддаленому об'єкту.

- *Реєстр RMI.* Реєстр RMI - це простий сервіс іменування, що надається Java RMI. Він дозволяє клієнтам знаходити віддалені об'єкти, прив'язуючи їх до унікальних імен. Клієнти можуть шукати об'єкти в реєстрі за їхніми іменами та отримувати посилання на віддалені об'єкти.
- *Серіалізація.* Java RMI використовує вбудований в Java механізм серіалізації для об'єднання і роз'єднання об'єктів і аргументів їх методів по мережі. Об'єкти, що надсилаються мережею, повинні реалізовувати інтерфейс Serializable або використовувати власні механізми серіалізації.

Java RMI спрощує розробку розподілених Java-додатків, абстрагуючись від складнощів віддаленого зв'язку. Він забезпечує прозорий віддалений виклик методів, де розробник може зосередитися на логіці програми, не маючи справи з низькорівневими деталями мережевої комунікації.

Однак, варто зазначити, що Java RMI в першу чергу використовується в комунікації Java-to-Java і вимагає, щоб і клієнт, і сервер були реалізовані на Java. Для більш нейтральної до мови або незалежної від платформи комунікації, інші технології, такі як CORBA або gRPC, можуть бути більш придатними.

Java RMI широко використовується в різних сферах, таких як розподілені системи, клієнт-серверні додатки та віддалений виклик сервісів. Він забезпечує надійну основу для створення розподілених додатків на Java і є частиною платформи Java Standard Edition (SE).

### 3.1.3. Apache Thrift

Apache Thrift — це фреймворк RPC (спочатку розроблений у Facebook, а потім переданий Apache), схожий на gRPC. Він використовує власну мову визначення інтерфейсу і пропонує підтримку широкого спектру мов програмування. Thrift дозволяє визначати типи даних та інтерфейси сервісів у файлі визначення. Беручи на вхід визначення сервісу, компілятор Thrift генерує

код для клієнтської та серверної частин. Транспортний рівень Thrift надає абстракції для мережевого вводу/виводу і відокремлює Thrift від решти системи, що означає, що він може працювати на будь-якій транспортній реалізації, такій як TCP, HTTP і так далі.

**3.1.3.1. Порівняння gRPC та Apache Thrift**

Якщо порівняти Thrift з gRPC, то можна побачити, що обидва мають майже однакові цілі розробки та використання. Однак, між ними є кілька важливих відмінностей.

*Передача даних.* gRPC є більш незалежним, ніж Thrift, і пропонує першокласну підтримку HTTP/2. Його реалізації на HTTP/2 використовують можливості протоколу для досягнення ефективності та підтримки шаблонів обміну повідомленнями, таких як потокова передача.

*Потокова передача.* Визначення сервісів gRPC за замовчуванням підтримують двонапрямний потік (клієнт і сервером) як частина самого визначення сервісу.

*Інтеграція.* Коли справа доходить до інтеграції у проект, то gRPC має багато ресурсів для дослідження технології. Ці ресурси включають у себе: документацію, зовнішні презентації, приклади використання, приклади інтеграції з іншими корисними додатками. Наявність цих ресурсів робить процес адаптації набагато легшим у порівнянні з Thrift.

*Оптимізація.* Хоча офіційних результатів порівняння gRPC і Thrift немає, є кілька онлайн-ресурсів, які порівнюють продуктивність цих двох технологій і показують кращі показники для Thrift. Однак, gRPC також інтенсивно тестується на продуктивність майже у всіх версіях. Тому продуктивність навряд чи буде вирішальним фактором, коли мова йде про вибір Thrift над gRPC. Крім того, існують інші фреймворки RPC, які пропонують подібні можливості, але gRPC в даний час лідирує як найбільш стандартизована, сумісна і широко прийнята технологія RPC.

### 3.2. Побудова API

Розробляючи API, постає багато задач, які треба вирішувати, включаючи вибір відповідного протоколу зв'язку для міжсервісної взаємодії. Це рішення стає особливо важливим в контексті сучасних архітектур, які використовують безліч сервісів.

Традиційно, в епоху монолітних архітектур, API призначалися в першу чергу для front-end або мобільних додатків. Однак поява мікросервісів призвела до необхідності створення внутрішніх каналів зв'язку між сервісами.

Існує кілька широко використовуваних протоколів, які можна розглянути для міжсервісної взаємодії, таких як REST, gRPC, GraphQL. Кожен протокол має свої сильні сторони, обмеження та варіанти використання.

Зрештою, вибір протоколу повинен відповідати конкретним вимогам і цілям API і загальної архітектури системи. Важливо ретельно оцінити компроміси і наслідки кожного протоколу, враховуючи такі фактори, як продуктивність, сумісність з існуючими системами, простота реалізації, а також майбутня масштабованість і розширюваність дизайну API.

#### 3.2.1. SOAP

Через обмеження традиційних реалізацій RPC, таких як CORBA, був розроблений і активно просувається великими компаніями, такими як Microsoft, IBM тощо, простий протокол доступу до об'єктів (Simple Object Access Protocol, SOAP). SOAP — це стандартна комунікаційна техніка в сервісно-орієнтованій архітектурі (SOA) для обміну структурованими даними на основі XML між сервісами (які в контексті SOA зазвичай називаються веб-сервісами), що передаються за допомогою будь-якого базового протоколу зв'язку, такого як HTTP (найбільш часто використовуваний). За допомогою SOAP ви можете визначити інтерфейс сервісу, операції цього сервісу та відповідний формат XML-повідомлень, який буде використовуватися для виклику цих операцій. SOAP був досить популярною технологією, але складність формату

повідомлень, а також складність специфікацій, побудованих на основі SOAP, перешкоджає гнучкості побудови розподілених додатків. Тому в контексті сучасної розробки розподілених додатків веб-сервіси SOAP вважаються застарілою технологією. Замість того, щоб використовувати SOAP, більшість існуючих розподілених додатків зараз розробляються з використанням архітектури REST.

### 3.2.2. REST

Representational State Transfer (REST) - це архітектурний стиль, який виник з докторської дисертації Роя Філдінга. Філдінг є одним з головних авторів специфікації HTTP і засновником архітектурного стилю REST. REST є основою ресурсно-орієнтованої архітектури (ROA), де розподілені додатки моделюються як набір ресурсів, а клієнти, які отримують доступ до цих ресурсів, можуть змінювати стан (створювати, читати, оновлювати або видаляти) цих ресурсів. Де-факто реалізацією REST є HTTP, і HTTP дозволяє моделювати REST веб-додаток як набір ресурсів, доступних за допомогою унікального ідентифікатора (URL). Над цими ресурсами застосовуються операції зміни стану у вигляді дієслів HTTP (GET, POST, PUT, DELETE, PATCH і так далі). Стан ресурсу представляється в текстових форматах, таких як JSON, XML, HTML, YAML і так далі.

Створення додатків з використанням архітектурного стилю REST за допомогою HTTP і JSON стало де-факто методом побудови мікросервісів. Однак, зі збільшенням кількості мікросервісів та їх мережевих взаємодій, REST-сервіси не змогли задовольнити очікувані сучасні вимоги. Існує кілька ключових обмежень RESTful сервісів, які перешкоджають використанню їх в якості протоколу обміну повідомленнями для сучасних додатків, заснованих на мікросервісах.

#### 3.2.2.1. Неефективні протоколи текстових повідомлень

За своєю суттю, RESTful сервіси побудовані на основі текстових транспортних протоколів, таких як HTTP 1.x, і використовують текстові формати, що читаються людиною, такі як JSON. Коли мова йде про комунікацію між сервісами, використання текстових форматів, таких як JSON, є досить неефективним, оскільки обидві сторони комунікації не потребують використання таких текстових форматів, придатних для читання людиною.

Клієнтська програма (джерело) створює двійковий вміст для відправки на сервер, потім перетворює двійкову структуру в текст (тому що HTTP 1.x дозволяє відправляти тільки текстові повідомлення) і відправляє його по мережі в текстовому вигляді (через HTTP) на машину, яка розбирає і перетворює його назад в двійкову структуру на стороні сервісу. Натомість більш ефективно було б надсилати двійковий формат, який можна зіставити з бізнес-логікою сервісу та споживача. Один з популярних аргументів на користь використання JSON полягає в тому, що його легше використовувати, оскільки він "читається людиною". Це більше проблема інструментарію, ніж проблема двійкових протоколів.

#### 3.2.2.2. Брак суворої типізації інтерфейсів між додатками

Зі збільшенням кількості сервісів, що взаємодіють через мережу, побудованих за допомогою різнорідних мов програмування, відсутність чітко визначених і суворо типізованих визначень сервісів стала серйозною перешкодою. Більшість технологій визначення сервісів, які наявні у RESTful сервісах, таких як OpenAPI/Swagger, не інтегровані з основним архітектурним стилем або протоколами обміну повідомленнями.

Це призводить до багатьох несумісностей, помилок під час виконання та проблем сумісності при створенні таких децентралізованих додатків. Наприклад, при розробці RESTful сервісів, не потрібно мати визначення сервісу і визначення типу інформації, якою обмінюються додатки. Замість цього розробляються RESTful-додатки, дивлячись на текстовий формат, який

подається на вхід, або на сторонні технології визначення API, як OpenAPI. Тому наявність сучасної суворо типізованої технології визначення сервісів і фреймворку, який генерує ядро серверного і клієнтського коду є ключовою необхідністю.

#### 3.2.2.3. Архітектурний стиль REST важко реалізувати

Як архітектурний стиль, REST має багато "хороших практик", яких потрібно дотримуватися, щоб створити справжній REST-сервіс. Але вони не є частиною протоколів реалізації (таких як HTTP), що ускладнює їх дотримання на етапі реалізації. Тому на практиці більшість сервісів, які претендують на статус RESTful, не дотримуються належним чином основ стилю REST. Таким чином, більшість так званих RESTful сервісів — це просто HTTP-сервіси, доступні через мережу. Тому командам розробників доводиться витрачати багато часу на підтримку узгодженості та чистоти RESTful сервісу.

### 3.2.3. GraphQL

GraphQL — це ще одна технологія (винайдена Facebook і стандартизована як відкрита), яка стає досить популярною для побудови міжпроцесної взаємодії. Це мова запитів для API та середовище виконання для виконання цих запитів з наявними даними. GraphQL пропонує принципово інший підхід до звичайної взаємодії між клієнтом і сервером, дозволяючи клієнтам визначати, які дані вони хочуть, як вони хочуть їх отримати і в якому форматі. gRPC, з іншого боку, має фіксований контракт на віддалені методи, які забезпечують зв'язок між клієнтом і сервером.

GraphQL більше підходить для зовнішніх сервісів або API, які доступні споживачам безпосередньо, де клієнти потребують більшого контролю над даними, які споживають з сервера. Наприклад, у сценарії додатку для інтернет-магазину, припустимо, що користувачам сервісу ProductInfo потрібна лише конкретна інформація про товари, а не весь набір атрибутів товару, і їм також потрібен спосіб вказати інформацію, яку вони хочуть отримати. За допомогою

GraphQL можливо змоделювати сервіс таким чином, щоб користувачі могли звертатися до нього, використовуючи мову запитів GraphQL, і отримувати необхідну інформацію.

### 3.3. Висновки до третього розділу

Розділ присвячений порівнянню gRPC з різними технологіями у двох ключових сферах: реалізація RPC та розробка API. Зокрема, обговорюється реалізація RPC в таких технологіях, як CORBA, Java RMI і Apache Thrift, а також порівнюється gRPC з REST і GraphQL з точки зору розробки API.

У першому розділі, розглядається, як gRPC порівнюється з іншими технологіями в реалізації механізмів віддаленого виклику процедур (RPC). Досліджуються такі технології, як CORBA (Common Object Request Broker Architecture), Java RMI (Remote Method Invocation) та Apache Thrift. У розділі обговорюються такі фактори, як продуктивність, мовна підтримка, простота використання та інтероперабельність. Аналізуючи ці аспекти, він дає уявлення про сильні та слабкі сторони gRPC у порівнянні з цими альтернативами.

Другий розділ, заглиблюється в порівняння gRPC з REST (Representational State Transfer) і GraphQL з точки зору розробки API. Він підкреслює відмінності в архітектурних стилях, моделюванні даних, шаблонах запитів/відповідей та клієнт-серверних взаємодіях. Розділ оцінює такі фактори, як гнучкість, ефективність, масштабованість і легкість інтеграції. Розглядаючи ці аспекти, він пропонує комплексне уявлення про те, чим gRPC відрізняється від REST і GraphQL в контексті розробки API.

В цілому, розділ, присвячений порівнянню gRPC та інших технологій, надає ретельний аналіз gRPC по відношенню до різних реалізацій RPC і підходів до розробки API. Це дозволяє зрозуміти переваги та обмеження gRPC у порівнянні з CORBA, Java RMI, Apache Thrift, REST та GraphQL. Це порівняння допомагає приймати обґрунтовані рішення щодо вибору найбільш підходящої технології для конкретних випадків використання та вимог.

# 4. Реалізація мікросервісної архітектури за допомогою платформи gRPC

У цьому розділі буде розроблена проста інфраструктура, яка буде складатись з 3 мікросервісів на різних мовах програмування. Комунікація у цій інфраструктурі буде реалізована на основі платформи gRPC. На основі такої інфраструктури будуть продемонстровано розглянутий раніше функціонал gRPC. А саме:

- Проектування API на основі protobuf.
  - Документація у proto файлах
  - Сувора типізація
  - Автогенерація
- Стримінг

## 4.1. Унарна комунікація

### 4.1.1. Бізнес логіка

Для початку треба сформувати бізнес-логіку, яку буде виконувати наш додаток:

- додаток буде надавати метеорологічні дані
- метеорологічні дані формуються на основі наданих даних позиції
- дані про позицію включаю у себе:
  - широту
  - довготу
- метеорологічні дані включають у себе:
  - дані про ультрафіолетовий індекс
  - дані про повітря
  - звіт
  - часові дані
- метеорологічні дані будуть надаватись за запитом користувача
- метеорологічні дані будуть надаватись за запитом сторонніх систем

### 4.1.2. Проектування API

На основі бізнес логіки можна спроектувати proto файл. Proto-файл [43] буде містити преамбулу. (рис. 4.1)

```
syntax = "proto3";
import "google/protobuf/timestamp.proto";
package com.ua.mlgmag.grpc;
option java_package = "com.ua.mlgmag.grpc";
```

Рисунок 4.1 - Преамбула weather.proto

Далі потрібно визначити такі сутності як «вологість», «температура», «координати», «часові дані». (рис. 4.2)

```
message Coordinates {
  fixed64 latitude = 1;
  fixed64 longitude = 2;
}

message Temperature {
  float degrees = 1;
  Units units = 2;

  enum Units {
    FAHRENHEIT = 0;
    CELSIUS = 1;
  }
}

message Humidity {
  float value = 1;
}

message Timestamp {
  int64 seconds = 1;
  int32 nanos = 2;
}

message Summary {
  string value = 1;
}
```

Рисунок 4.2 - Повідомлення у weather.proto

Наступним кроком буде поєднання усіх наведених даних у одну єдину відповідь. (рис. 4.3)

```
message WeatherResponse {
  AirData airData = 1;
  UltraVioletData ultraVioletData = 2;
  Summary summary = 3;
  google.protobuf.Timestamp dateTimeStamp = 4;
}
```

Рисунок 4.3 - Повідомлення WeatherResponse у weather.proto

На цьому етапі можна формувати API для сервісів. (рис. 4.4)

```
service WeatherService {
  rpc getWeather(Coordinates) returns (WeatherResponse);
  rpc getWeatherStream(stream Coordinates) returns (stream WeatherResponse);
}
service AirService {
  rpc getAirData(Coordinates) returns (AirData);
  rpc getAirDataStream(stream Coordinates) returns (stream AirData);
}

service UltraVioletService {
  rpc getUltraVioletData(Coordinates) returns (UltraVioletData);
  rpc getUltraVioletDataStream(stream Coordinates) returns (stream
UltraVioletData);
}

service SummaryService {
  rpc getSummary(Coordinates) returns (Summary);
  rpc getSummaryStream(stream Coordinates) returns (stream Summary);
}
```

Рисунок 4.4 - API у weather.proto

Наданий proto-файл представляє набір сервісів та визначень повідомлень, пов'язаних з погодними даними. У файлі визначено декілька сервісів, таких як WeatherService, AirService, UltraVioletService та SummaryService, кожен з яких пропонує певну функціональність, пов'язану з пошуком погодної інформації. Розглянемо компоненти proto-файлу.

Сервіси:

- WeatherService: Надає методи для отримання інформації про погоду. Включає методи getWeather та getWeatherStream, які повертають повідомлення WeatherResponse.
- AirService: Надає методи для отримання даних про повітря. Включає методи getAirData і getAirDataStream, які повертають повідомлення Temperature.
- UltraVioletService: Надає методи для отримання даних про ультрафіолетовий індекс. Включає методи getUltraVioletData і getUltraVioletDataStream, які повертають повідомлення UltraVioletData.
- SummaryService: Надає методи для отримання підсумкової інформації. Включає методи getSummary і getSummaryStream, які повертають повідомлення Summary.

Повідомлення:

- WeatherResponse: Являє собою відповідь на запити, пов'язані з погодою. Містить наступні поля:
  - AirData: Відображає стан повітря в певному місці, з полем з назвою airData типу AirData.
  - UltraVioletData: Відображає дані про ультрафіолетовий індекс у певному місці з полем з назвою ultraVioletData типу UltraVioletData.
  - DateTimeStamp: Відображає дату та час, коли було записано погодні дані. Використовує тип google.protobuf.Timestamp.
  - Summary: Надає підсумок погодних умов у вигляді рядка.
- Координати: Відображає широту і довготу певного місця. Містить наступні поля:
  - Широта: Представляє координату широти як значення у форматі fixed64 (представлення з фіксованим розміром у 64-біт).
  - Довгота: Відображає координату довготи як значення у форматі fixed64 (представлення з фіксованим розміром у 64-біт).

- AirData: Відображає дані про стан повітря. Містить наступні поля:
  - QualityIndex: Відображає індекс забрудненості повітря у вигляді цілого числа.
  - PollutionLevel: Відображає рівень забрудненості повітря. Це тип зі сталою кількістю значень, який може мати значення GREEN або RED.
- UltraVioletData: Відображає дані про ультрафіолетове випромінювання. Містить одне поле:
  - Значення: Представляє значення ультрафіолетового випромінювання як ціле число.
- Підсумок: Відображає підсумок погодних умов у вигляді рядка. Містить одне поле:
  - Значення: Відображає текст зведення.

Prot-файл [43] визначає як унарні RPC-методи (наприклад, getWeather, getAirData тощо), так і потокові RPC-методи (наприклад, getWeatherStream, getAirDataStream тощо) для отримання метеорологічних даних. Потокові методи дозволяють безперервно оновлювати або отримувати дані, приймаючи на вхід потік координат і повертаючи потік відповідних повідомлень про погоду.

Цей proto-файл можу бути використаний для створення коду на мові програмування на якій буде написаний додаток за допомогою компілятора protobuf. Згенерований код надасть суворо типізовані API для взаємодії з визначеними сервісами і типами повідомлень, що дозволить реалізувати пов'язану з погодою функціональність у програмному додатку.

### 4.1.3. Логічна схема архітектури додатку.

При наявному API можна спроектувати логічну схему архітектури додатку.

На схемі (рис. 4.5) зображені протоколи комунікації, які будуть використані. У всій інфраструктурі використовується протокол RPC, але при взаємодії на пряму з користувачем протокол має бути змінено на HTTP. Це варто

зробити, бо більшість користувачів системи буде користуватись веб-браузером для взаємодії з нашою системою. Браузери використовуються протоколи на основі HTTP для взаємодії з серверами.

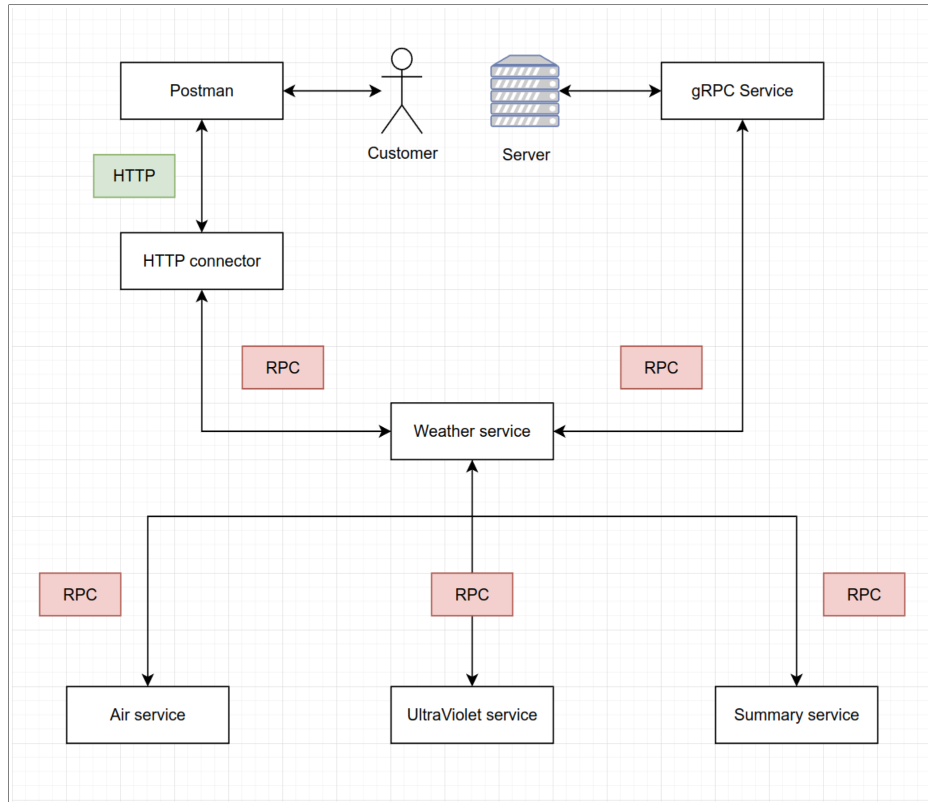


Рисунок 4.5 - Логічна схема архітектури додатку

### 4.1.4. Фізична схема архітектури додатку

Фізична схема буде відрізнятись від логічної: (рис. 4.6)

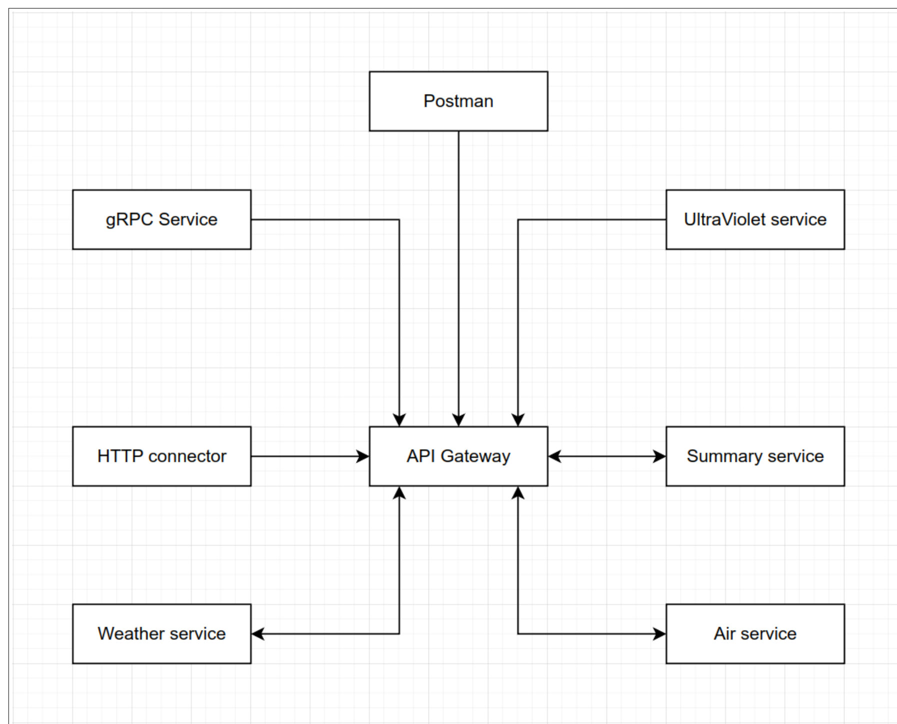


Рисунок 4.6 - Фізична схема архітектури додатку

Фізична схема містить у собі додатковий сервіс API Gateway. Наявність API Gateway має вирішальне значення в архітектурі мікросервісів з кількох причин:

- Абстракція сервісу. API Gateway діє як єдина точка входу для клієнтів для доступу до декількох мікросервісів. Він абстрагується від складнощів, що лежать в основі окремих сервісів, надаючи уніфікований інтерфейс, з яким можуть взаємодіяти клієнти. Ця абстракція спрощує інтеграцію на стороні клієнта, оскільки йому не потрібно знати про конкретні мікросервіси та їхні кінцеві точки.
- Маршрутизація запитів і балансування навантаження: Шлюз API може керувати маршрутизацією запитів і балансуванням навантаження між декількома екземплярами одного мікросервісу. Він інтелектуально спрямовує клієнтські запити до відповідного екземпляру сервісу на основі попередньо визначених правил або алгоритмів. Це забезпечує

ефективне використання ресурсів та покращує загальну продуктивність і масштабованість системи.

Загалом, API Gateway відіграє важливу роль у підвищенні керованості, масштабованості, безпеки та продуктивності архітектури мікросервісів. Він забезпечує уніфіковану та контрольовану точку доступу для клієнтів, абстрагує базові сервіси та дозволяє централізовано реалізовувати різні наскрізні проблеми.

#### 4.1.5. Розробка сервісів

У цьому розділі будуть розглянуть деталі реалізації кожного сервісу.

##### 4.1.5.1. Air service

Цей сервіс буде розроблено за допомогою мови JavaScript на основі платформи NodeJS [44]. (рис. 4.7)

```javascript
const grpc = require("@grpc/grpc-js");
const PROTO_PATH = "weather.proto";
var protoLoader = require("@grpc/proto-loader");

var packageDefinition = protoLoader.loadSync(
        PROTO_PATH,
        {
                keepCase: true,
                longs: String,
                enums: String,
                defaults: true,
                oneofs: true
        }
);

const protoDescriptor = grpc.loadPackageDefinition(packageDefinition);
const airService = protoDescriptor.com.ua.mlgmag.grpc.AirService;

function getRandomArbitrary(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function createResponse() {
  let qualityIndex = getRandomArbitrary(150, 400);

  let pollutionLevel = 1;
  if (qualityIndex > 250) {
    pollutionLevel = 2;
  }

  return { qualityIndex: qualityIndex, pollutionLevel: pollutionLevel };
}

function getAirData(coordinates) {
  console.log("Call getAirData");
  console.log(
    `Coordinates: latitude '${coordinates.latitude}' longitude '${coordinates.longitude}'`
  );
  console.log("Response is sent successfully");
  return createResponse();
}

const server = new grpc.Server();

const server = new grpc.Server();
server.addService(airService.service, {
  getAirData: (call, callback) => {
    callback(null, getAirData(call.request));
  }
});

server.bindAsync("127.0.0.1:8082", grpc.ServerCredentials.createInsecure(), () => {
        console.log("Server running at 127.0.0.1:8082");
        server.start();
});
```

Рисунок 4.7 - Реалізація air service на JavaScript

Наведений код (рис. 4.7) демонструє реалізацію сервісу gRPC за допомогою Node.js. Розберемо код та опишемо його функціональність:

1. Залежності:
   - Код імпортує пакет grpc, який є Node.js реалізацією gRPC, високопродуктивного фреймворку з відкритим вихідним кодом для віддаленого виклику процедур.
   - Він також вимагає пакет @grpc/proto-loader і призначає його змінній protoLoader. Цей пакет дозволяє завантажувати буферні файли протоколу (.proto) і генерувати відповідний JavaScript код.
2. Визначення буфера протоколу:
   - Код визначає шлях до буферного файлу протоколу за допомогою константи PROTO_PATH.
   - Він використовує protoLoader для завантаження буферного файлу протоколу і генерує відповідне визначення пакета за допомогою loadSync(). Надані опції (keepCase, longs, enums, defaults, oneofs) налаштовують поведінку процесу завантаження.
3. Завантаження та доступ до сервісу:
   - Код завантажує визначення пакунка за допомогою grpc.loadPackageDefinition() і присвоює його proto-дескриптору.
   - Він отримує доступ до TemperatureService із завантаженого визначення пакета за допомогою protoDescriptor.com.ua.mlgmag.grpc.AirDataService.
4. Реалізація сервісу:
   - У коді визначено функцію getAirData(coordinates), яка імітує логіку отримання даних про якість повітря на основі наданого параметра координат. Вона реєструє координати і повертає об'єкт AirData.
5. Налаштування та прив'язка сервера:
   - Код створює новий екземпляр сервера gRPC за допомогою grpc.Server().

   - Він додає temperatureService як сервіс до сервера за допомогою server.addService(). Реалізований метод getAirData зіставляється з відповідним методом сервісу у proto-файлі.
6. Конфігурація та запуск сервера:
   - Код прив'язує сервер до адреси "127.0.0.1:8082" за допомогою server.bindAsync(). Він використовує grpc.ServerCredentials.createInsecure() для створення незахищених облікових даних сервера (без шифрування).
   - Після успішного зв'язування сервера виконується функція зворотного виклику, яка реєструє повідомлення про те, що сервер працює за вказаною адресою.
   - Нарешті, сервер запускається за допомогою server.start().

Загалом, цей код налаштовує сервер gRPC в Node.js, завантажує визначення буфера протоколу, реалізує метод getTemperature і запускає сервер для прослуховування вхідних запитів. Він демонструє базову реалізацію для отримання даних про температуру, яка може бути розширена для включення додаткових сервісних методів і логіки, заснованої на визначеному proto-файлі.

#### 4.1.5.2. UltraVioletService

Цей сервіс буде розроблено за допомогою мови Python [45]. (рис. 4.8)

```python
import grpc
from concurrent import futures
import weather_pb2 as pb2
import weather_pb2_grpc as pb2_grpc
import random

def createResponse():
    value = random.randint(1, 7)
    return {"value": value}

class UltraVioletService(pb2_grpc.UltraVioletServiceServicer):

    def __init__(self, *args, **kwargs):
        pass

    def getUltraVioletData(self, request, context):
        print("Call getUltraVioletData")
        latitude = request.latitude
        longitude = request.longitude
        print(f"Coordinates: latitude '{latitude}' longitude '{longitude}'")

        result = createResponse()

        print("Response is sent successfully")

        return pb2.UltraVioletData(**result)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    pb2_grpc.add_UltraVioletServiceServicer_to_server(UltraVioletService(), server)
    server.add_insecure_port("[::]:8081")
    server.start()
    print("Server running at 127.0.0.1:8081")
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

Рисунок 4.8 - Реалізація UltraVioletService на Python

Наведений код (рис. 4.8) демонструє реалізацію сервісу gRPC за допомогою Python. Розберемо код та опишемо його функціональність:

1. Залежності:
   - Код імпортує необхідні модулі, включаючи grpc для роботи gRPC, concurrent.futures для роботи з паралелізмом, time для операцій,

пов'язаних з часом, і два згенеровані Python файли weather_pb2 і weather_pb2_grpc, які містять згенерований код з файлу weather.proto.

2. Реалізація сервісу:
   - У коді визначено клас з назвою UltraVioletService, який слугує реалізацією сервісу gRPC для функціональності, пов'язаної з вологістю.
   - Клас успадковується від pb2_grpc.UltraVioletServiceServicer, який генерується з визначення protobuf і забезпечує необхідну базову функціональність для обслуговування UltraVioletService.
   - Метод __init__ ініціалізує службу, але наразі він порожній.
   - Метод getHumidity є реалізацією методу сервісу getUltraVioletData, визначеного у файлі protobuf. Він отримує параметр запиту, який містить значення широти і довготи місця, для якого запитується вологість. Метод друкує деталі запиту і повертає об'єкт pb2.UltraVioletData з жорстко закодованим значенням вологості 18.9.
3. Налаштування та запуск сервера:
   - Код визначає функцію serve, яка налаштовує сервер gRPC.
   - Сервер створюється за допомогою grpc.server(futures.ThreadPoolExecutor(max_workers=10)), яка створює сервер з максимум 10 паралельними робочими потоками для обробки вхідних запитів.
   - Реалізація HumidityService додається до сервера за допомогою pb2_grpc.add_UltraVioletServiceServicer_to_server(UltraVioletService(), server).
   - Сервер налаштовано на прослуховування незахищеного порту [::]:8081 за допомогою server.add_insecure_port('[::]:8081').
   - Сервер запускається за допомогою server.start(), і виводиться повідомлення про те, що сервер запущено.

- ○ Нарешті, сервер очікує на завершення роботи за допомогою server.wait_for_termination(), що дозволяє йому обробляти вхідні запити, доки його не буде явно завершено.

Загалом, цей код налаштовує gRPC-сервер на Python, реалізує UltraVioletService для обробки запитів, пов'язаних з вологістю, і запускає сервер для прослуховування вхідних запитів на порт 8081. Він демонструє базову реалізацію для отримання даних про вологість, яку можна розширити, включивши додаткові методи обслуговування та логіку, засновану на визначеному proto-файлі.

#### 4.1.5.3. Summary service

Цей сервіс буде розроблено за допомогою мови Golang [46]. Наведений код (рис. 4.9) демонструє реалізацію сервісу gRPC на мові Go (Golang). Розберемо код та опишемо його функціональність:

1. Залежності:
   - ○ Код імпортує необхідні пакунки, зокрема "context" для керування контекстом у gRPC-операціях, "log" для ведення журналу, "net" для роботи у мережі, "pb" для імпортованого коду, згенерованого у protobuf, та "google.golang.org/grpc" для функціональності gRPC.
2. Реалізація сервісу:
   - ○ У коді визначено структуру з іменем summaryServer, яка представляє реалізацію сервісу gRPC для функцій, пов'язаних зі зведеннями. Структура містить pb.SummaryServiceServer, який генерується з імпортованого визначення protobuf і надає необхідну базову функціональність для обслуговування SummaryService.
   - ○ Реалізація включає метод з іменем GetSummary, який відповідає підпису, визначеному у файлі protobuf. Він приймає об'єкт context.Context та об'єкт *pb.Coordinates як параметри. Він реєструє

деталі запиту і повертає об'єкт *pb.Summary з жорстко закодованим підсумковим повідомленням: "Сьогодні сонячно!".

3. Налаштування та запуск сервера:
   - Код визначає головну функцію, яка є точкою входу в програму.
   - Вона створює TCP-слухач за допомогою net.Listen() на порту 8085. Якщо під час створення слухача виникає помилка, вона реєструє помилку і завершує роботу програми.
   - Новий gRPC-сервер створюється за допомогою grpc.NewServer().
   - Реалізація summaryServer реєструється на gRPC-сервері за допомогою pb.RegisterSummaryServiceServer(grpcServer, &summaryServer{}), що дозволяє серверу обробляти вхідні запити до SummaryService.
   - Він реєструє адресу прослуховування сервера за допомогою lis.Addr().
   - Сервер gRPC починає обслуговувати вхідні запити викликом grpcServer.Serve(lis). Якщо під час запуску сервера виникає помилка, він записує її до журналу і завершує роботу програми.

Загалом, цей код налаштовує gRPC-сервер у Go, реалізує summaryServer для обробки запитів, пов'язаних зі зведеннями, і запускає сервер для прослуховування вхідних запитів на порт 8085. Він демонструє базову реалізацію для надання зведених даних, яку можна розширити, включивши додаткові методи обслуговування і логіку, засновану на визначеному proto-файлі.

```
package main
import (
        "context"
        "log"
        "net"
        pb "github.com/mlgmag/summary-service/proto"
        "google.golang.org/grpc"
)

type summaryServer struct {
        pb.SummaryServiceServer
}

func (m *summaryServer) GetSummary(ctx context.Context, coordinates *pb.Coordinates)
(*pb.Summary, error) {
    log.Println("Call getSummary")
     log.Printf("Coordinates: latitude '%d' longitude '%d'", coordinates.Latitude,
coordinates.Longitude)
    summary := "It's sunny today!"
    return &pb.Summary{Value: summary}, nil
}

func main() {
    lis, err := net.Listen("tcp", ":8085")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    grpcServer := grpc.NewServer()

    pb.RegisterSummaryServiceServer(grpcServer, &summaryServer{})
    log.Printf("server listening at %v", lis.Addr())
    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

Рисунок 4.9 - Реалізація summary service на Golang

#### 4.1.5.4. Weather service

Цей сервіс буде розроблено за допомогою мови Java [47]. Сервіс включає у себе декілька файлів:

- Клієнти:
    - UltraVioletClient
    - SummaryClient

- AirClient
- Інше:
  - DefaultWeatherService
  - Main

Основна задача клієнтів (рис. 4.10) полягає у тому щоб встановити з'єднання з іншими сервісами. Для з'єднання з кожним сервісом виділений окремий файл.

```
public class UltraVioletClient {

    private static final int PORT = 8081;
    private static final String HOST = "127.0.0.1";

    private final UltraVioletServiceGrpc.UltraVioletServiceBlockingStub stub;

    public UltraVioletClient() {
                ManagedChannel  channel  =  NettyChannelBuilder.forAddress(HOST,
PORT).usePlaintext().build();
        this.stub = UltraVioletServiceGrpc.newBlockingStub(channel);
    }

         public   Weather.UltraVioletData   getUltravioletData(Weather.Coordinates
coordinates) {
        return stub.getUltraVioletData(coordinates);
    }
}
```

Рисунок 4.10 - Реалізація UltraVioletClient на Java

Цей код створює gRPC-клієнт на Java для сервісу UltraViolet. Клас UltraVioletClient встановлює з'єднання з gRPC-сервером за допомогою керованого каналу і надає метод getUltraVioletData для виклику методу getUltraVioletData RPC на сервері. Він демонструє базову реалізацію для блокування викликів gRPC, яка може бути розширена для включення додаткових сервісних методів і логіки на основі визначеного proto-файлу.

DefaultWeatherService (рис. 4.11) відповідає за об'єднання даних, які були отримані з інших сервісів. Основна задача Main класу це створення gRPC серверу для надання API іншим сервісам.

```java
public class DefaultWeatherService extends WeatherServiceGrpc.WeatherServiceImplBase {

    private static final Logger LOG = LoggerFactory.getLogger(DefaultWeatherService.class);

    private static final AirClient AIR_DATA = new AirClient();
    private static final UltraVioletClient ULTRAVIOLET_CLIENT = new UltraVioletClient();
    private static final SummaryClient SUMMARY_CLIENT = new SummaryClient();

    @Override
    public void getWeather(Weather.Coordinates coordinates, StreamObserver<Weather.WeatherResponse>
responseObserver) {

        LOG.info("Coordinates: latitude '{}' longitude '{}'", coordinates.getLatitude(),
coordinates.getLongitude());

        Timestamp currentTimestamp = WeatherUtils.getCurrentTimestamp();
        Weather.AirData airData = AIR_DATA.getAirData(coordinates);
        Weather.UltraVioletData ultraVioletData = ULTRAVIOLET_CLIENT.getUltravioletData(coordinates);
        Weather.Summary summary = SUMMARY_CLIENT.getSummary(coordinates);

        Weather.WeatherResponse weatherResponse = Weather.WeatherResponse.newBuilder()
                .setDateTimeStamp(currentTimestamp)
                .setAirData(airData)
                .setUltraVioletData(ultraVioletData)
                .setSummary(summary)
                .build();

        responseObserver.onNext(weatherResponse);
        responseObserver.onCompleted();
    }
}
```

Рисунок 4.11 - Реалізація Default Weather Service на Java

#### 4.1.5.5. HTTP connector

Завдання цього сервісу полягає у наданні доступу до функціоналу додатку для протоколів, які базуються на HTTP. Цей метод надає єдиний endpoint: /weather. Для правильного використання потрібно надати значення координат.

```
/weather?latitude=987&longitude=1111
```

Рисунок 4.12 - URL шлях для виклику HTTP adapter

```
@Component
public class WeatherClient {
    private static final int PORT = 8080;
    private static final String HOST = "127.0.0.1";

    private final WeatherServiceGrpc.WeatherServiceBlockingStub stub;

    public WeatherClient() {
            ManagedChannel channel = NettyChannelBuilder.forAddress(HOST,
PORT).usePlaintext().build();
        this.stub = WeatherServiceGrpc.newBlockingStub(channel);
    }

     public Weather.WeatherResponse getWeather(Weather.Coordinates
coordinates) {
        return stub.getWeather(coordinates);
    }
}
```

Рисунок 4.13 - Реалізація Weather Client на Java

```
@GetMapping("/weather")
private WeatherResponseDto getWeather(@RequestParam int latitude, @RequestParam int
longitude) {
    LOG.info("Coordinates: latitude '{}' longitude '{}'", latitude, longitude);

    Weather.Coordinates coordinates = Weather.Coordinates.newBuilder()
            .setLatitude(latitude)
            .setLongitude(longitude)
            .build();

    Weather.WeatherResponse weatherResponse = weatherClient.getWeather(coordinates);

    return weatherConverter.convert(weatherResponse);
}
```

Рисунок 4.14 - Реалізація Weather Client на Java

### 4.1.6. Результат виконання

Для перевірки правильності виконання додатку треба перевірити правильність виконання усіх його окремих компонентів. Це можна здійснити за допомогою утиліти grpc_cli. [48]

#### 4.1.6.1. Перевірка temperature service

Надішлемо RPC запит за допомогою відповідної CLI команди через grpc_cli. (рис. 4.15)

```
grpc_cli call airService getAirData \
--noremotedb \
--protofiles=weather.proto \
--json_input \
--infile=coordinates.json
```

Рисунок 4.15 - Команда для надсилання RPC запиту через grpc_cli за допомогою CLI

Інформація про файли:

- “weather.proto” містить інформацію про API та RPC повідомлення.
- “coordinates.json“ містить інформацію про розташування.

```
{
    "latitude": "123",
    "longitude": "234"
}
```

Рисунок 4.16 - Вміст файлу coordinates.json

```
connecting to 127.0.0.1:8082
Received initial metadata from server:
date : Sun, 21 May 2023 16:44:41 GMT
qualityIndex: 273
pollutionLevel: RED
Rpc succeeded with OK status
```

Рисунок 4.17 - Відповідь від temperature service

```
Call getAirData
Coordinates: latitude '123' longitude '234'
Response is sent successfully
```

Рисунок 4.18 - Логи зі сторони temperature service

**4.1.6.2. Перевірка додатка**

Для перевірки роботи додатку був використаний інструмент Postman Таким чином, отримані метрологічні дані (рис. 4.19) можна використовувати у інших додатках для відображення даних про погоду.

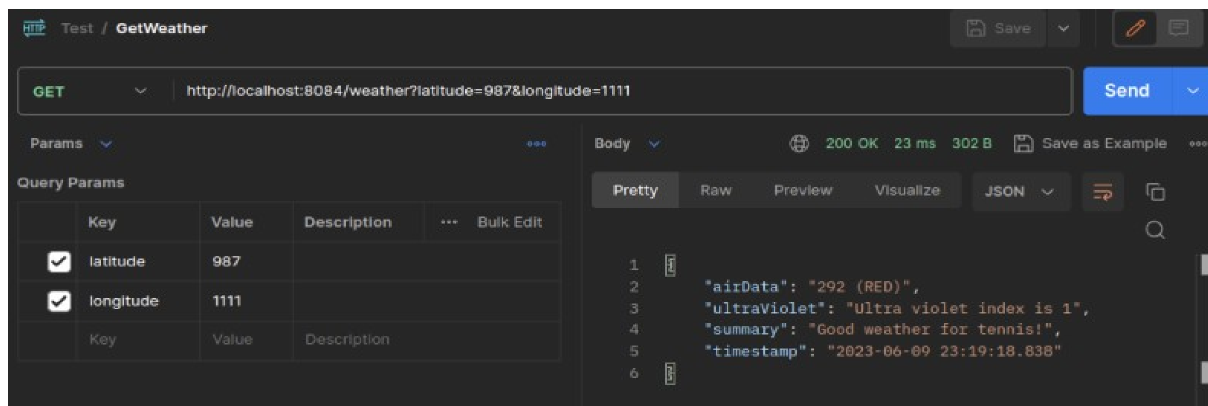


Рисунок 4.19 - Запит у Postman

## 4.2. Двостороння потокова комунікація

Щоб користуватись функціональністю двосторонньої потокової реалізації потрібно реалізувати додаткові методи на кожному з сервісів. (рис. 4.20)

```
rpc getWeatherStream(stream Coordinates) returns (stream WeatherResponse);
rpc getAirDataStream(stream Coordinates) returns (stream AirData);
rpc getUltraVioletDataStream(stream Coordinates) returns (stream UltraVioletData);
rpc getSummaryStream(stream Coordinates) returns (stream Summary);
```

Рисунок 4.20 - Методи для реалізації streaming

### 4.2.1. Air service

Реалізуємо метод *getAirDataStream* для сервісу який надає дані про температуру. (рис. 4.21)

```
function getAirDataStream(call) {
  call.on("data", function (coordinates) {
    console.log("Call getAirDataStream");
    console.log(
      `Coordinates: latitude '${coordinates.latitude}' longitude '${coordinates.longitude}'`
    );
    call.write(createResponse());
    console.log("Response is sent successfully");
  });

  call.on("error", function (error) {
    console.log(`${error.code} → ${error.details}`);
  });

  call.on("end", function () {
    call.end();
  });
}
```

Рисунок 4.21 - Реалізація методу getAirDataStream

Реалізацію (рис. 4.21) можна описати наступним чином:

1. Визначено функцію getAirDataStream, яка отримує об'єкт виклику як параметр. Цей об'єкт представляє потоковий виклик gRPC, зроблений клієнтом.
2. Усередині функції налаштовується приймач подій за допомогою call.on("дані", функція (координати) {...}). Цей приймач подій прослуховує вхідні події даних від клієнта. Щоразу, коли відбувається подія даних, виконується надана функція зворотного виклику.
3. Усередині функції зворотного виклику отриманий об'єкт координат виводиться на консоль, відображаючи значення широти та довготи.
4. Функція createResponse() викликається для генерації об'єкта відповіді, який буде надіслано клієнту. Потім відповідь надсилається за допомогою call.write().
5. Виводиться повідомлення в лог, яке вказує на те, що відповідь було успішно надіслано.
6. Ще один приймач подій налаштовується за допомогою call.on("error", function (error) {...}). Цей обробник подій прослуховує будь-які помилки,

які можуть виникнути під час потокового передавання. Якщо виникає помилка, виконується передбачена функція зворотного виклику, яка записує код помилки та деталі.

7. Останній обробник подій налаштовується за допомогою call.on("end", function () {...}). Цей обробник подій спрацьовує, коли клієнт завершує потоковий запит. Коли ця подія відбувається, виконується функція зворотного виклику, яка завершує потік викликів з боку сервера викликом call.end().

### 4.2.2. UltraViolet Service

Реалізуємо метод *getUltraVioletDataStream* для сервісу який надає дані про вологість. (рис. 4.22)

```
def getUltraVioletDataStream(self, request_iterator, context):
    for request in request_iterator:
        print("Call getUltraVioletDataStream")
        latitude = request.latitude
        longitude = request.longitude
        print(f"Coordinates: latitude '{latitude}' longitude '{longitude}'")

        result = createResponse()

        yield pb2.UltraVioletData(**result)

        print("Response is sent successfully")
```

Рисунок 4.22 - Реалізація методу getUltraVioletDataStream

Реалізацію (рис. 4.22) можна описати наступним чином:

1. Метод getUltraVioletDataStream визначено в класі, який зазвичай асоціюється з реалізацією сервісу gRPC. Він приймає три параметри: self, request_iterator і context.
2. Всередині методу використовується цикл for для перебору ітератора request_iterator. Цей ітератор представляє потік вхідних запитів від клієнта.

3. У кожній ітерації циклу виконується блок коду. Відбуваються наступні дії:
    1. Виводиться повідомлення в журнал про те, що був викликаний метод getUltraVioletDataStream.
    2. Значення широти та довготи витягуються з поточного об'єкту запиту та присвоюються змінним latitude та longitude відповідно.
    3. Виводиться повідомлення в журнал, яке відображає витягнуті значення широти та довготи.
    4. Викликається функція createResponse() для створення об'єкта відповіді. Конкретна реалізація цієї функції у фрагменті коду не надається.
    5. Об'єкт відгуку видається з використанням ключового слова yield. Це вказує на те, що метод є функцією-генератором, здатною виробляти послідовність значень. У цьому випадку значенням, що повертається, є екземпляр pb2.UltraVioletData, який представляє тип повідомлення-відповіді.
    6. f. Друкується повідомлення журналу, яке вказує на те, що відповідь було успішно надіслано.
4. Цикл продовжується до тих пір, поки в ітераторі request_iterator не залишиться більше запитів. У цей момент метод завершується.

### 4.2.3. Summary service

Реалізуємо метод *getSummaryStream* для сервісу який надає дані про вологість. (рис. 4.23)

```
func (s *SummaryServer) GetSummaryStream(stream proto.SummaryService_GetSummaryStreamServer) error {
	for {
		in, err := stream.Recv()

		if err == io.EOF {
			return nil
		}

		if err != nil {
			return err
		}

		log.Println("Call GetSummaryStream")
		log.Printf("Coordinates: latitude '%d' longitude '%d'", in.Latitude, in.Longitude)

		summary := summaryService.GetSummary()
		response := &proto.Summary{Value: summary}
		if err := stream.Send(response); err != nil {
			return err
		}

		log.Println("Response is sent successfully")
	}
}
```

Рисунок 4.23 - Реалізація методу getSummaryStream

Реалізацію (рис. 4.23) можна описати наступним чином:

1. Функцію GetSummaryStream визначено як метод структури SummaryServer у контексті реалізації сервісу gRPC. Вона отримує два параметри: s, який представляє собою вказівник на екземпляр SummaryServer, і stream, який є об'єктом потоку gRPC-сервера.
2. Всередині функції створено нескінченний цикл за допомогою конструкції циклу for. Цей цикл безперервно обробляє вхідні запити від клієнта до тих пір, поки не буде виконано умову завершення.
3. У кожній ітерації циклу виконуються наступні дії:
    1. Метод Recv() викликається на об'єкті потоку для отримання повідомлення від клієнта. Він повертає отримане повідомлення та будь-які можливі помилки. Отримане повідомлення присвоюється змінній in.
    2. Якщо отримане повідомлення вказує на кінець потоку (err == io.EOF), то вихід з циклу здійснюється шляхом повернення nil.

3. Якщо під час операції отримання виникає будь-яка інша помилка, вона повертається з функції, що вказує на проблему з процесом потокового передавання.
4. Виводиться повідомлення в журнал про те, що викликано метод GetSummaryStream.
5. З отриманого повідомлення витягуються значення широти та довготи (in.Latitude та in.Longitude відповідно).
6. Виконується додаткова логіка для отримання підсумкових даних за допомогою методу summaryService.GetSummary(). Специфіка цієї реалізації не наведена у фрагменті коду.
7. Створюється повідомлення-відповідь з використанням отриманих зведених даних. Повідомлення-відповідь має тип proto.Summary і присвоюється змінній response.
8. Метод Send() використовується в об'єкті потоку для відправки повідомлення-відповіді назад клієнту. Якщо під час операції відправки виникає помилка, вона повертається з функції, що вказує на проблему з відправкою відповіді.
9. Виводиться повідомлення в log, яке вказує на те, що відповідь було успішно надіслано.

4. Цикл продовжується, обробляючи наступні запити по мірі їх надходження, доки не буде виконано умову завершення (кінець потоку або помилка).

Таким чином, ця реалізація обробляє потоковий метод gRPC на мові Golang, безперервно отримуючи повідомлення від клієнта, обробляючи кожне повідомлення, генеруючи відповідь і відправляючи її назад клієнту. Цикл гарантує, що сервер може обробляти декілька запитів протягом певного часу і завершується, коли потік закінчується або виникає помилка.

#### 4.2.4. Weather service

Для реалізації відповідного методу потрібно створити додаткову утиліту. Основне її призначення — створення сутності Observer яка необхідна для реалізації потокового методу. (рис. 4.24)

```
public static <T> StreamObserver<T> newStreamObserver(
        StreamObserver<WeatherResponse> responseStreamObserver,
        Lock lock,
        BiFunction<Builder, T, Builder> converter) {
    return new StreamObserver<T>() {
        @Override
        public void onNext(T value) {
            WeatherResponse response =
converter.apply(WeatherResponse.newBuilder(), value).build();
            lock.lock();
            responseStreamObserver.onNext(response);
            lock.unlock();
        }

        @Override
        public void onError(Throwable t) {
            lock.lock();
            responseStreamObserver.onError(t);
            lock.unlock();
        }

        @Override
        public void onCompleted() {
            lock.lock();
            responseStreamObserver.onCompleted();
            lock.unlock();
        }
    };
}
```

Рисунок 4.24 - Реалізація методу getSummaryStream

Реалізуємо метод *getWeatherStream* для сервісу який надає дані про вологість. (рис. 4.24)

```java
@Override
public StreamObserver<Coordinates> getWeatherStream(StreamObserver<WeatherResponse> responseObserver)
{
    LOG.info("Call 'getWeatherStream'");

    ReentrantLock lock = new ReentrantLock();
    StreamObserver<Coordinates> airDataStream =
            airServiceStub.getAirDataStream(newStreamObserver(responseObserver, lock,
                    WeatherResponse.Builder::setAirData));

    StreamObserver<Coordinates> ultraVioletDataStream =
            ultraVioletServiceStub.getUltraVioletDataStream(newStreamObserver(responseObserver, lock,
                    WeatherResponse.Builder::setUltraVioletData));

    StreamObserver<Coordinates> summaryStream =
            summaryClient.getSummaryStream(newStreamObserver(responseObserver, lock,
                    WeatherResponse.Builder::setSummary));

    List<StreamObserver<Coordinates>> streamObservers =
            ImmutableList.copyOf(List.of(airDataStream, ultraVioletDataStream, summaryStream));

    return new StreamObserver<>() {
        @Override
        public void onNext(Coordinates value) {
            streamObservers.forEach(s -> s.onNext(value));
        }

        @Override
        public void onError(Throwable t) {
            streamObservers.forEach(s -> s.onError(t));
        }

        @Override
        public void onCompleted() {
            streamObservers.forEach(StreamObserver::onCompleted);
        }
    };
}
```

Рисунок 4.25 - Реалізація методу getSummaryStream

Реалізацію (рис. 4.25) можна описати наступним чином:

1. Метод getWeatherStream перевизначає метод, визначений в інтерфейсі або суперкласі. Він отримує StreamObserver<Coordinates> як параметр, що представляє потік координат від клієнта. Він повертає StreamObserver<Coordinates>, який використовується для надсилання координат на сервер.
2. Метод починається з реєстрації інформаційного повідомлення про те, що було викликано метод getWeatherStream.
3. Створюється об'єкт ReentrantLock з іменем lock для забезпечення безпеки потоку під час одночасного доступу.

4. Метод отримує три окремі екземпляри StreamObserver<Coordinates>, airStream, ultraVioletStream та summaryStream, викликаючи відповідні потокові методи gRPC на airClient, ultraVioletClient та summaryClient відповідно. Ці потокові спостерігачі отримуються за допомогою методу newStreamObserver, передаючи responseObserver, lock та відповідний метод встановлювача WeatherResponse.Builder як аргументи.
5. Отримані спостерігачі потоку зберігаються у списку з назвою streamObservers за допомогою методу ImmutableList.copyOf.
6. Метод повертає новий екземпляр StreamObserver<Coordinates> як анонімний внутрішній клас.
   1. Перевизначено метод onNext для ітераційного перебору всіх спостерігачів потоку в streamObservers і виклику onNext(value) для кожного з них при отриманні нового значення координат.
   2. Перевизначено метод onError для перебору всіх спостерігачів потоку в streamObservers та виклику onError(t) для кожного з них при виникненні помилки.
   3. Перевизначено метод onCompleted для ітераційного перебору всіх спостерігачів потоку в streamObservers і виклику onCompleted() для кожного з них, коли потік завершено.

Таким чином, ця реалізація обробляє потоковий метод gRPC на Java, отримуючи окремі спостерігачі потоку від різних клієнтських сервісів (airClient, ultravioletClient і summaryClient), пересилаючи отримані координати до кожного з цих спостерігачів потоку і забезпечуючи безпеку потоку за допомогою ReentrantLock. Метод повертає новий екземпляр StreamObserver<Coordinates> як анонімний внутрішній клас для обробки потоку координат від клієнта.

#### 4.2.5. Використання двосторонньої потокової реалізації

При наявності двосторонньої потокової реалізації можна відтворити різні типи комунікації. (табл. 4.1)

| Тип комунікації | Клієнт | Сервер |
|---|---|---|
| Унарна (звичайна) | Унарна | Унарна |
| Потокова на стороні клієнта | Потокова | Унарна |
| Потокова на стороні сервера | Унарна | Потокова |
| Двостороння потокова | Потокова | Потокова |

Табл. 4.1 - Можливі види комунікації

#### 4.2.6. Перевірка потокової реалізації

У минулому прикладі (розділ 4.1) було перевірено звичайну унарну комунікацію. Тепер при наявності двосторонньої потокової реалізації буде перевірена потокова комунікація на стороні клієнта.

Для перевірки потокової реалізації використаємо утиліту grpc_cli. [48]

##### 4.2.6.1. Перевірка Summary Service

Надішлемо потоковий RPC запит за допомогою відповідної CLI команди через grpc_cli. (рис. 4.26)

```
grpc_cli call summaryService getSummaryStream \
--noremotedb \
--protofiles=google/protobuf/timestamp.proto,weather.proto \
--json_input
```

Рисунок 4.26 - Команда для надсилання потокового RPC запиту через grpc_cli за допомогою CLI

```
mlgmag@mlgmag-box:~/coding2/go/summaryService$ go run main.go
2023/05/19 07:03:34 Server running at 127.0.0.1:8085
2023/05/19 07:04:15 Call GetSummaryStream
2023/05/19 07:04:15 Coordinates: latitude '123' longitude '234'
2023/05/19 07:04:15 Response is sent successfully
2023/05/19 07:04:17 Call GetSummaryStream
2023/05/19 07:04:17 Coordinates: latitude '123' longitude '234'
2023/05/19 07:04:17 Response is sent successfully
2023/05/19 07:04:18 Call GetSummaryStream
2023/05/19 07:04:18 Coordinates: latitude '123' longitude '234'
2023/05/19 07:04:18 Response is sent successfully
2023/05/19 07:04:19 Call GetSummaryStream
2023/05/19 07:04:19 Coordinates: latitude '123' longitude '234'
2023/05/19 07:04:19 Response is sent successfully
2023/05/19 07:04:20 Call GetSummaryStream
2023/05/19 07:04:20 Coordinates: latitude '123' longitude '234'
2023/05/19 07:04:20 Response is sent successfully
```

Рисунок 4.27 - Логи зі сторони summary service

```
mlgmag@mlgmag-box:~/coding2/resources/grpc$ ./getSummaryStream.bash
reading streaming request message from stdin...
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
value: "Sadness and rain ;("
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
value: "Sadness and rain ;("
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
value: "It\'s sunny today!"
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
value: "+27 but feels like +100..."
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
value: "Sadness and rain ;("
Stream RPC succeeded with OK status
mlgmag@mlgmag-box:~/coding2/resources/grpc$
```

Рисунок 4.28 - Відповіді від summary service

#### 4.2.6.2. Перевірка Summary Service

Надішлемо потоковий RPC запит за допомогою відповідної CLI команди через grpc_cli. (рис. 4.29)

```
grpc_cli call summaryService getSummaryStream \
--noremotedb \
--protofiles=google/protobuf/timestamp.proto,weather.proto \
--json_input
```

Рисунок 4.29 - Команда для надсилання потокового RPC запиту через grpc_cli за допомогою CLI

```
mlgmag@mlgmag-box:~/coding2/resources/grpc$ ./getWeatherStreaming.bash
reading streaming request message from stdin...
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
humidity {
  value: 1
}
got response.
summary {
  value: "+27 but feels like +100..."
}
got response.
temperature {
  degrees: -16
  units: FAHRENHEIT
}
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
summary {
  value: "+27 but feels like +100..."
}
got response.
humidity {
  value: 81
}
got response.
temperature {
  degrees: -38
  units: CELSIUS
}
{
        "latitude": "123",
        "longitude": "234"
}

Request sent.
got response.
summary {
  value: "+27 but feels like +100..."
}
got response.
humidity {
  value: 52
}
got response.
temperature {
  degrees: 35
  units: CELSIUS
}
Stream RPC succeeded with OK status
mlgmag@mlgmag-box:~/coding2/resources/grpc$
```

Рисунок 4.30 - Відповідь від weather service

Таким чином, отримані метрологічні дані (рис. 4.30) можна використовувати у інших додатках для відображення метеорологічних даних.

### 4.3. Порівняння gRPC та REST сервісів

У цьому розділі буде порівняно затримку на рівні мережі, структуру повідомлень та обсяг передачі даних при використанні gRPC та REST сервісів. Для цього Summary service буде переписаний на звичайний REST API.

#### 4.3.1. Архітектура

Архітектура буде складатись з 3 сервісів: (рис. 4.31)

- Клієнт. Він буде відповідати за надсилання запитів до сервісів.
- gRPC Summary Service
- REST Summaary Service.

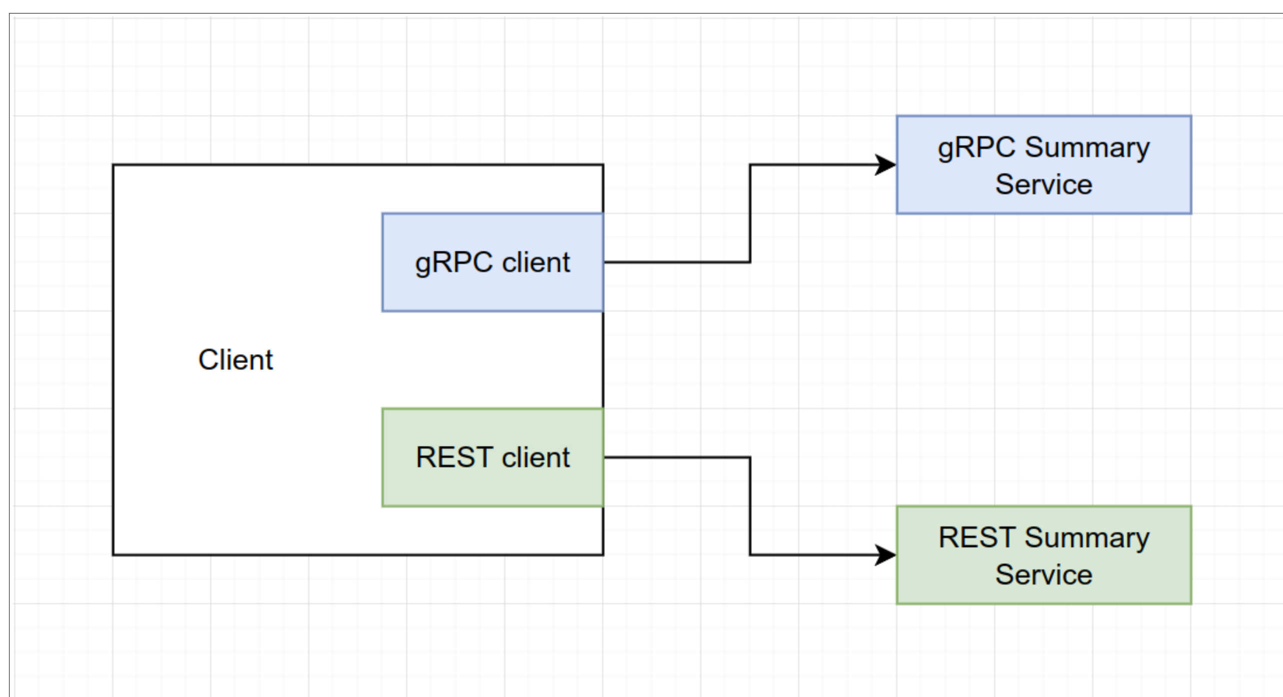


Рисунок 4.3.1 - Архітектура

#### 4.3.2. Client

На стороні клієнт а буде зроблено чотири кінцеві точки:

- /rest/summary
- /rest/summary/stream
- /grpc/getSummary
- /grpc/getSummaryStream

Звичайні кінцеві точки ‘/rest/summary’ та ‘/grpc/getSummary’ будуть повертати 1 відповідь. А потокові кінцеві точки ‘/rest/summary/stream’ та ‘/grpc/getSummaryStream’ будуть повертати n відповідей. Це зроблено для тестування потокової передачі.

При виконанні потокових запитів, клієнтська частина буде надсилати n-запитів на відповідний сервіс.

Для передачі даних у REST частині використовується XML, gRPC — Protobuf.

Приклад повідомлень:


```
<SummaryDto>
   <value>It's sunny today!</value>
</SummaryDto>
```


Рисунок 4.3.2 - Rest повідомлення


```
{
 "value": "It's sunny today!"
}
```


Рисунок 4.3.3 - gRPC повідомлення (розкодоване)

ChFJdCdzIHN1bm55IHRvZGF5IQ==

Рисунок 4.3.4 - gRPC повідомлення (закодоване base64)

### 4.3.3. Порівняння швидкодії

При порівнянні швидкодії, можна отримати такі результати

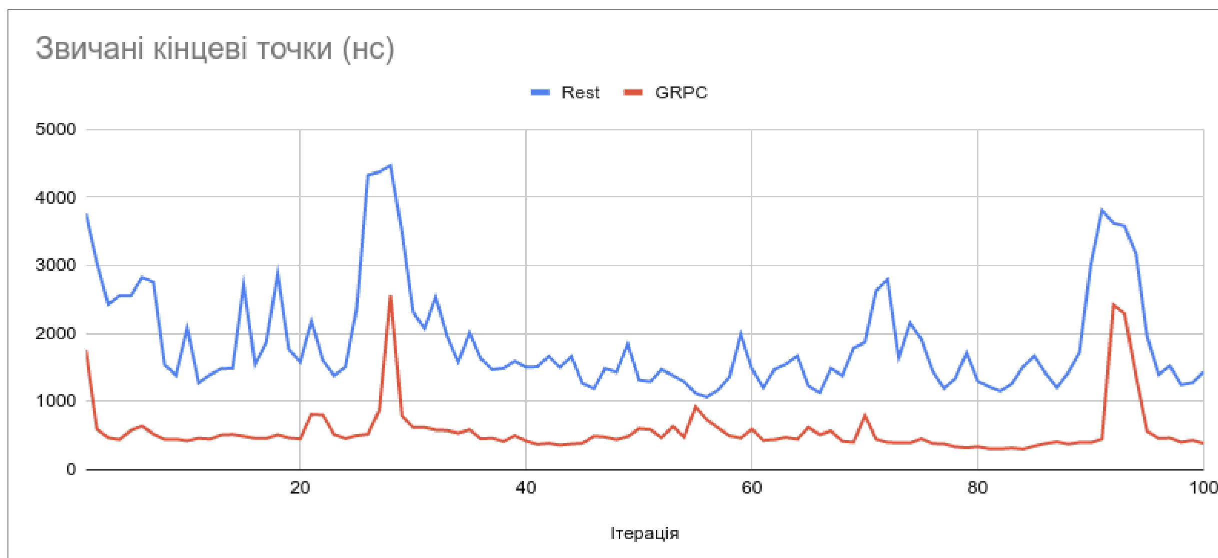


Рисунок 4.3.5 - Звичайні кінцеві точки. 100 ітерацій. Час у наносекундах

Рисунок 4.3.6 - Потокові кінцеві точки. 10 ітерацій. Час у мілісекундах. 100 запитів

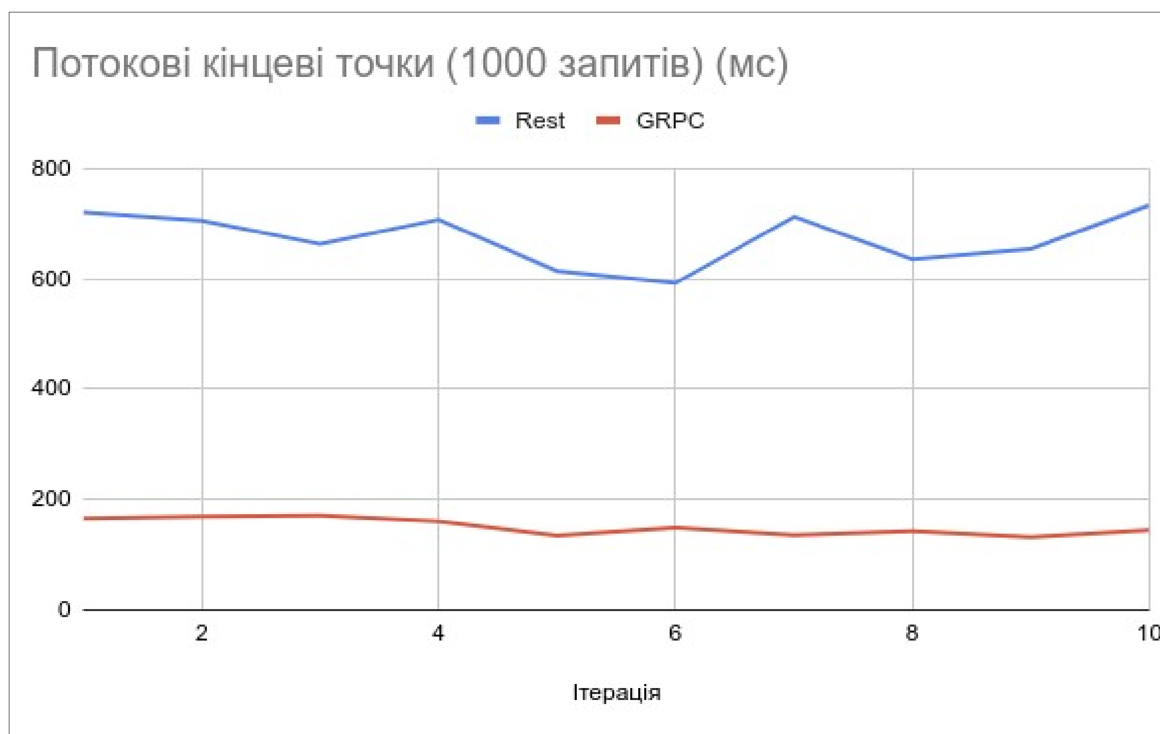


Рисунок 4.3.7 - Потокові кінцеві точки. 10 ітерацій. Час у мілісекундах. 1000 запитів

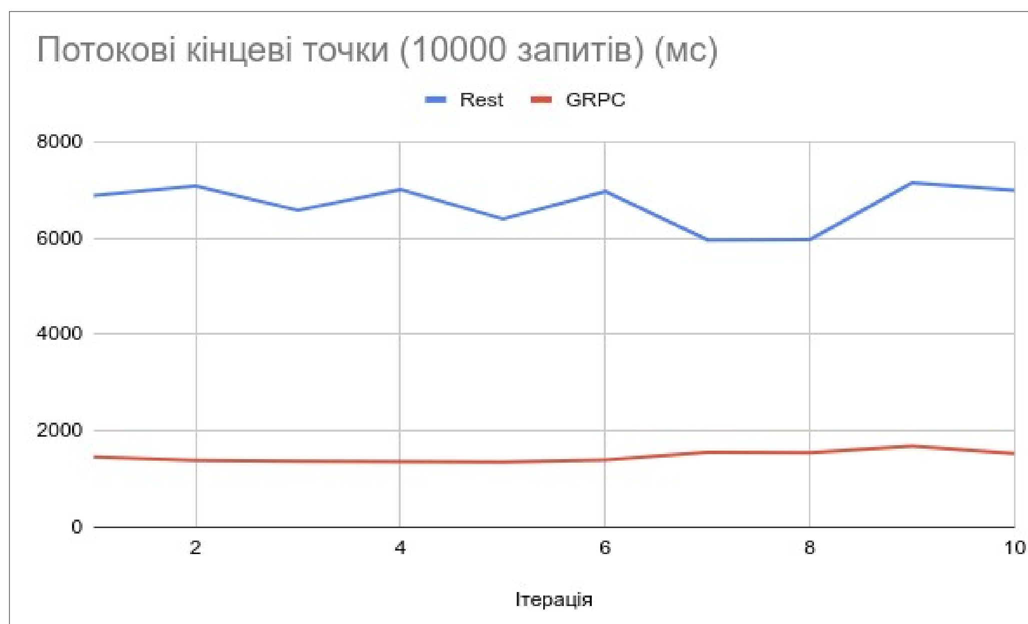


Рисунок 4.3.8 - Потокові кінцеві точки. 10 ітерацій. Час у мілісекундах. 10_000 запитів

На всіх графіках gRPC має перевагу над REST. На потокових кінцевих точках це пов’язано з використанням gRPC протоколу HTTP/2, у той час як REST використовує HTTP/1.1.

Також варто відзначити, що у gRPC вся серіалізація виконана на основі Protobuf, а REST використовує XML. Далі буде порівняно обсяг передачі даних.

### 4.3.3. Порівняння обсягу передачі даних

Для першого порівняння буде використано повідомлення з рис. 4.3.2 / рис. 4.3.3. (Табл. 4.3.1)

Табл. 4.3.1

| **Прокол** | **К-сть запитів** | **1** | **100** | **1000** | **10000** |
|---|---|---|---|---|---|
| REST | | 54 байт | 5.27 кб | 52.75 кб | 527.51 кб |
| gRPC | | 29 байт | 2.83 кб | 28.32 кб | 283.27 кб |

Для другого порівняння буде використано повідомлення Weather Response з рис. 4.3.9 / рис. 4.3.10. (Табл. 4.3.2)

| Табл. 4.3.2 | | | | | |
|---|---|---|---|---|---|
| **Прокол** | **К-сть запитів** | **1** | **100** | **1000** | **10000** |
| REST | | 246 байт | 24.02 кб | 240.23 кб | 2.34 мб |
| gRPC | | 45 байт | 4.39 кб | 43.94 кб | 439.45 кб |


```
<WeatherResponse>
        <airData>
                <qualityIndex>7</qualityIndex>
                <pollutionLevel>2</pollutionLevel>
        </airData>
        <ultraVioletData>
                <value>9</value>
        </ultraVioletData>
        <summary>
                <value>It's sunny today!</value>
        </summary>
</WeatherResponse>
```


Рисунок 4.3.9 - REST повідомлення WeatherResponse


```
{
 "airData": {
  "qualityIndex": 7,
  "pollutionLevel": 2
 },
 "ultraVioletData": {
  "value": 9
 },
 "summary": {
  "value": "It's sunny today!"
 }
}
```


Рисунок 4.3.10 - gRPC повідомлення WeatherResponse (розкодоване)


```
CgQIBxACEgIICRoTChFJdCdzIHN1bm55IHRvZGF5IQ==
```


Рисунок 4.3.11 - gRPC повідомлення WeatherResponse (закодоване base64)

На простих запитах, які містять 1 або 2 поля, повідомлення gRPC менше ніж повідомлення XML у 1.8 разів. На складних об’єктах різниця майже у 6 разів на користь gRPC.

## 4.4. Висновки до четвертого розділу

Розділ присвячений практичним аспектам використання gRPC для розробки мікросервісної архітектури.

У першому розділі, обговорюється реалізація унарних RPC-викликів за допомогою gRPC. Пояснюється, як визначати сервіси gRPC і типи повідомлень за допомогою буферів протоколу, а також демонструється, як реалізувати та інтегрувати ці сервіси в архітектуру мікросервісів. Цей розділ підкреслює простоту та ефективність унарних викликів RPC, коли робиться один запит і отримується одна відповідь, що робить його придатним для багатьох випадків використання.

Другий розділ, досліджує реалізацію двонаправленого потоку з використанням gRPC. Він демонструє можливості gRPC у підтримці зв'язку в реальному часі та ефективного потокового передавання даних між мікросервісами. Двонаправлений потік дозволяє як клієнту, так і серверу надсилати декілька повідомлень асинхронно, що дозволяє створювати інтерактивні та адаптивні додатки.

У третьому розділі, порівнюється продуктивність gRPC і REST з точки зору затримок і розміру повідомлень. У ньому представлено аналіз профілювання, який вимірює затримку запитів і розмір повідомлень, якими обмінюються мікросервіси, що використовують gRPC і REST. Цей аналіз дає уявлення про відмінності в продуктивності між двома протоколами, підкреслюючи ефективність і потенційні переваги використання gRPC з точки зору зменшення затримок і оптимізації розміру повідомлень.

## 5. Функціонально вартісний аналіз програмного продукту

В заданому розділі буде проведено оцінювання основних характеристик для майбутнього програмного продукту, що спеціалізується на дослідженні демографічного стану.

Дана реалізація буде сприяти проведенню усіх необхідних досліджень, що дасть змогу якісно дослідити питання не лише в Україні, проте у всьому світі.

Також в даному дослідженні показано різні варіанти реалізації для забезпечення найбільш коректної та оптимальної стратегії вибору, що має вплив на економічні фактори та сумісність з майбутнім програмним продуктом. Для цього застосовувався апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) передбачає собою технологію, що дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. ФВА проводиться з метою виявлення резервів зниження витрат за рахунок ефективніших варіантів виробництва, кращого співвідношення між споживчою вартістю виробу та витратами на його виготовлення. Для проведення аналізу використовується економічна, технічна та конструкторська інформація.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення повних витрат (річних) та кількості робочих часів, визначення джерел витрат та кінцевий розрахунок вартості програмного продукту.

### 5.1 Постановка задачі проектування

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки системи прогнозу стійкості фінансових показників. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляється, впливають на всю систему, кожна окрема підсистема має її задовольняти. Тому фактичний аналіз представляє собою аналіз функцій

програмного продукту, призначеного для збору, обробки та проведення аналізу даних по компанії.

Технічні вимоги до програмного продукту є наступні:

- функціонування на персональних комп’ютерах із стандартним набором компонентів;
- зручність та зрозумілість для користувача;
- швидкість обробки даних та доступ до інформації в реальному часі;
- можливість зручного масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

**5.2 Обґрунтування функцій програмного продукту**

Головна функція – розробка можливого програмного продукту, яка дозволяє аналізувати різні характеристики, що безпосередньо впливають на стійкість підприємства. Беручи за основу цю функцію, можна виділити наступні:

- $F_1$ – вибір самої програми.
- $F_2$ – якісний аналіз даних.
- $F_3$ – графічні показники.

Кожна з цих функцій має декілька варіантів реалізації:

1. Функція $F_1$:
    1. Eviews.
    2. R.
2. Функція $F_2$.
    1. Застосування вбудованих функцій.
    2. Створення своїх обчислень значень.
3. Функція $F_3$:
    1. Використання шаблонних графіків.
    2. Створення своїх.

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 5.1).

Рисунок 5.1 - Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіантів основних функцій. Позитивно-негативна матриця показана в таблиці 5.1.

Таблиця 5.1 - Позитивно-негативна матриця

| Функції | Варіанти реалізації | Переваги | Недоліки |
|---|---|---|---|
| $F_1$ | *А* | Загальнодоступна програма, доступність багатьох бібліотек | Необхідність повної реалізації алгоритму |
| | *Б* | Доступна в реалізації програма для різних обчислень | На написання коду необхідно мати базові навички та вміння |
| $F_2$ | *А* | Доступність та легкість при написанні | Іноді не відповідає задачі яку треба розв'язати |
| | *Б* | Ідеально описують усі необхідні | Достатньо затратно реалізовувати свої |

| | | | |
|---|---|---|---|
| | | характеристики | алгоритми для подальшої реалізації |
| $F_3$ | *А* | Загально прийнята реалізація | Іноді не відповідає очікуваним значенням |
| | *Б* | При виконанні власних досліджень краще може передавати висновки | Необхідно достатньо багато часу для написання програми для побудови та знаходження всього необхідного в задачі. |

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція $F_1$:

Перевагу даємо загальнодоступності. Для спрощення роботи по написанню коду варіант Б має бути відкинутий.

Функція $F_2$:

Програма допускає обрання обох варіантів. Можливо використати варіанти А чи Б.

Функція $F_3$:

Реалізація першого варіанту є сприйнятливою для програми. Це варіант А.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

$F_1$а – $F_2$а – $F_3$а

$F_1$а – $F_2$*б* – $F_3$а

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

## 5.3 Обґрунтування системи параметрів програмного продукту

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X1 – швидкодія мови програмування;
- X2 – об'єм пам'яті для обчислень та збереження даних;
- X3 – час навчання даних;
- X4 – потенційний об'єм програмного коду.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту, як показано у таблиці 5.2.

Таблиця 5.2 - Основні параметри програмного продукту

| Назва Параметра | Умовні позначення | Одиниці виміру | Значення параметра | | |
|---|---|---|---|---|---|
| | | | гірші | середні | кращі |
| Швидкодія мови програмування | X1 | оп/мс | 60 | 80 | 110 |
| Об'єм пам'яті | X2 | Мб | 60 | 50 | 30 |
| Час попередньої обробки даних | X3 | мс | 80 | 70 | 60 |
| Потенційний об'єм програмного коду | X4 | кількість рядків коду | 35 | 25 | 20 |

За даними таблиці 5.2 будуються графічні характеристики параметрів – рис. 5.2 – рис. 5.5.

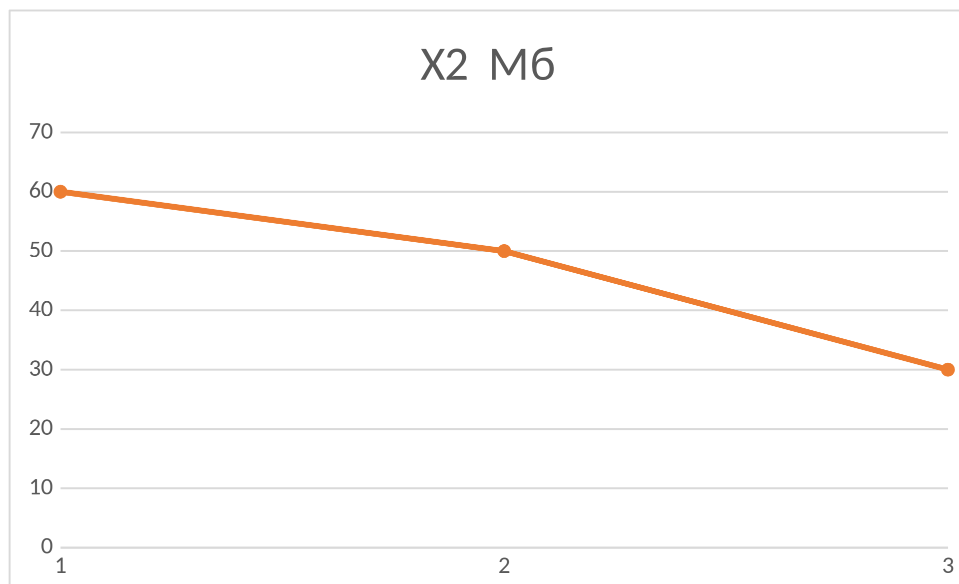


Рисунок 5.2 - X1, швидкодія мови програмування

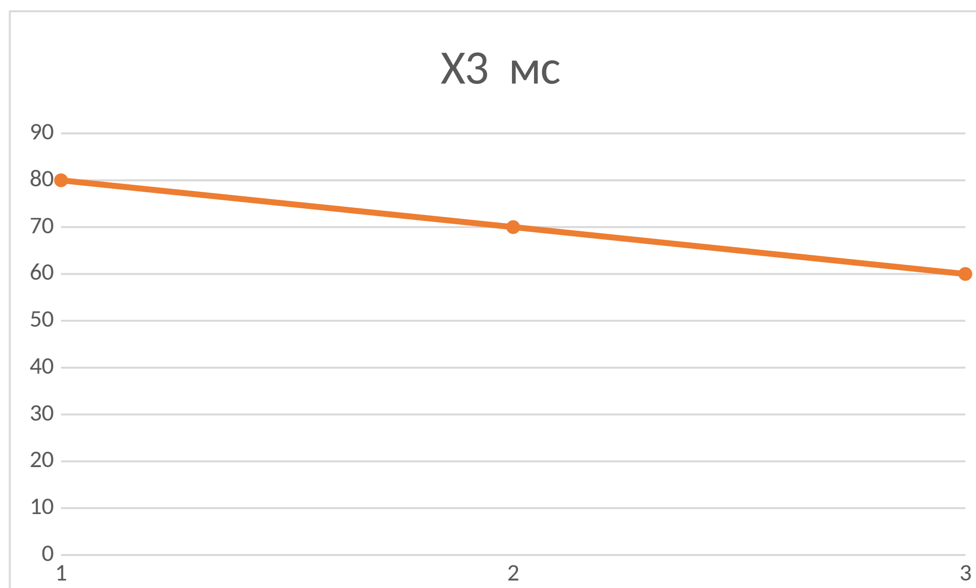


Рисунок 5.3 - X2, об’єм пам’яті

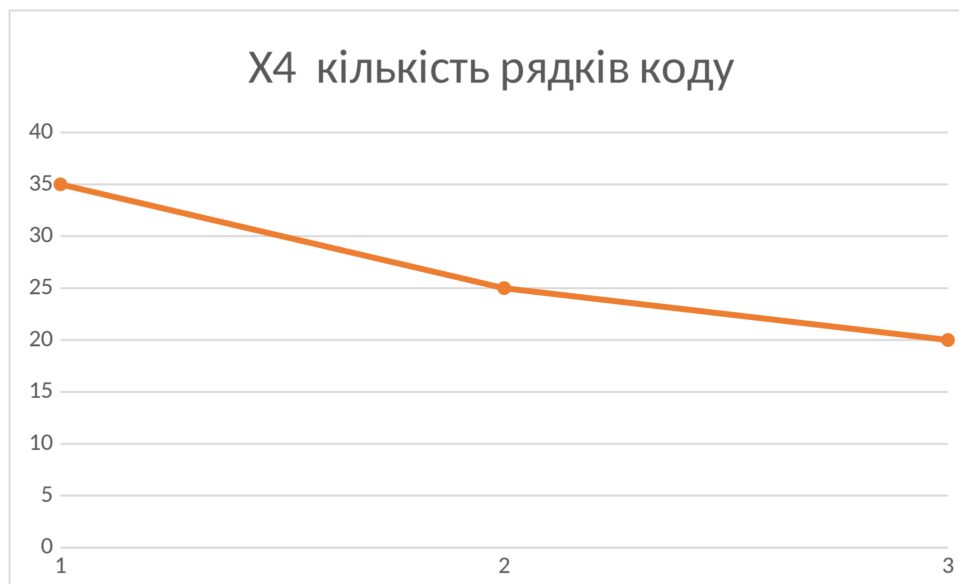


Рисунок 5.4 - Х3, час попередньої обробки даних

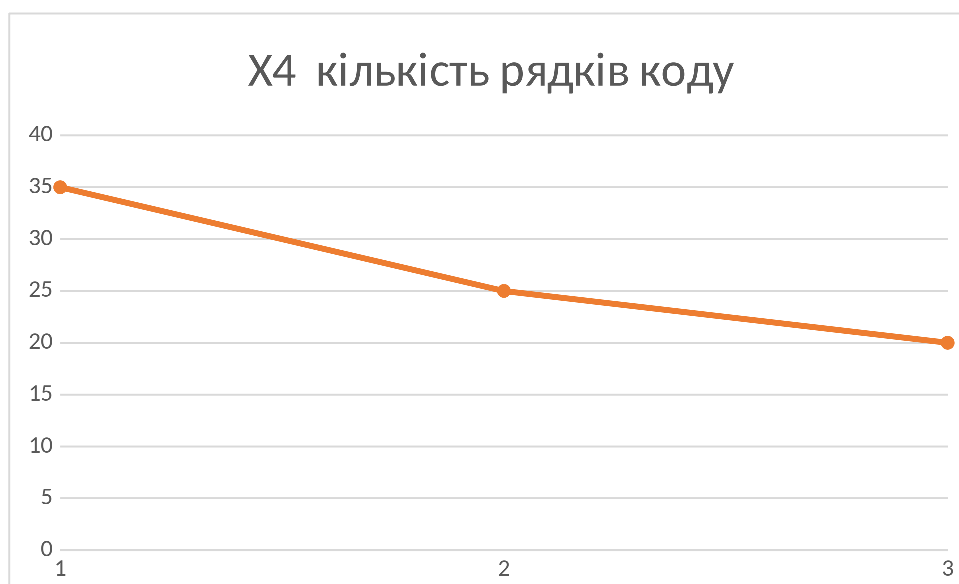


Рисунок 5.5 - Х4, потенційний об’єм програмного коду

### 5.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.

Таблиця 5.3 - Результати ранжування параметрів

| Позначення параметра | Назва параметра | Одиниці виміру | Ранг параметра за оцінкою експерта | | | | | | | Сума рангів $R_i$ | Відхилення $\Delta_i$ | $\Delta_i^2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | |
| *X1* | Швидкодія мови програмування | Оп/мс | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 10 | -7,5 | 56,25 |
| *X2* | Об'єм пам'яті | Mb | 3 | 4 | 3 | 3 | 4 | 3 | 4 | 24 | 6,5 | 42,25 |
| *X3* | Час попередньої обробки даних | мс | 2 | 1 | 1 | 2 | 2 | 2 | 1 | 11 | -6,5 | 42,25 |
| *X4* | Потенційний об'єм програмного коду | Кількість рядків коду | 4 | 3 | 4 | 4 | 3 | 4 | 3 | 25 | 7,5 | 56,25 |
| | Разом | | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 70 | 0 | 197 |

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70, \quad (4.1)$$

де $N$ – число експертів,

$n$ – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17{,}5 \quad (4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \quad (4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 197. \quad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12\,S}{N^2(n^3 - n)} = \frac{12 \cdot 197}{7^2(4^3 - 4)} = 0{,}754 > W_k = 0{,}67. \quad (4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 5.

Таблиця 5.4 — Попарне порівняння параметрів.

| Параметри | Експерти | | | | | | | Кінцева оцінка | Числове значення |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | |
| X1 і X2 | < | < | < | < | < | < | < | < | 0,5 |

| X1 і X3 | < | > | > | < | < | < | > | < | 0,5 |
|---|---|---|---|---|---|---|---|---|---|
| X1 і X4 | < | < | < | < | < | < | < | < | 0,5 |
| X2 і X3 | > | > | > | > | > | > | > | > | 1,5 |
| X2 і X4 | < | > | < | < | > | < | < | < | 0,5 |
| X3 і X4 | < | < | < | < | < | < | < | < | 0,5 |

Числове значення, що визначає ступінь переваги i–го параметра над j–тим, aij визначається по формулі:

$$a_{ij}=\begin{cases}1.5\,при\,X_i>X_j\\1.0\,при\,X_i=X_j\\0.5\,при\,X_i<X_j\end{cases}. \quad (4.6)$$

З отриманих числових оцінок переваги складемо матрицю A=‖aij‖.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{ві}=\frac{b_i}{\sum_{i=1}^{n} b_i} \quad (4.7)$$

$$b_i=\sum_{i=1}^{N} a_{ij} \quad (4.8)$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{ві}=\frac{b_i'}{\sum_{i=1}^{n} b_i'}, \quad (4.9)$$

$$b_i'=\sum_{i=1}^{N} a_{ij} b_j \quad (4.10)$$

Як видно з таблиці 5.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 5.5 - Розрахунок вагомості параметрів

| Параметри $x_i$ | Параметри $x_j$ | | | | Перша ітер. | | Друга ітер. | | Третя ітер | |
|---|---|---|---|---|---|---|---|---|---|---|
| | X1 | X2 | X3 | X4 | $b_i$ | $K_{вi}$ | $b_i^1$ | $K_{вi}^1$ | $b_i^2$ | $K_{вi}^2$ |
| X1 | 1 | 0,5 | 0,5 | 0,5 | 2,5 | 0,16 | 9,25 | 0,16 | 34,125 | 0,16 |
| X2 | 1,5 | 1 | 1,5 | 0,5 | 4,5 | 0,28 | 16,25 | 0,28 | 59,125 | 0,28 |
| X3 | 1,5 | 0,5 | 1 | 0,5 | 3,5 | 0,22 | 12,25 | 0,21 | 41,875 | 0,2 |
| X4 | 1,5 | 1,5 | 1,5 | 1 | 5,5 | 0,34 | 21,25 | 0,35 | 77,875 | 0,36 |
| Всього: | | | | | 16 | 1 | 59 | 1 | 213 | 1 |

### 5.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів X2 (Об’єм пам’яті), X3 (час попередньої обробки даних) та X4 (потенційний об’єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X1 (швидкість роботи мови програмування) обрано не найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 5.6):

$$K_K(j)=\sum_{i=1}^{n} K_{вi,j} B_{i,j}, \quad (4.11)$$

де $n$ – кількість параметрів;

$K_{вi}$– коефіцієнт вагомості $i$–го параметра;

$B_i$ – оцінка $i$–го параметра в балах.

Таблиця 5.6 - Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

| Основні | Варіант | Параметри | Абсолютне | Бальна | Коефіцієнт | Коефіцієнт рівня |
|---|---|---|---|---|---|---|

| функції | реалізації функції | | значення параметра | оцінка параметра | вагомості параметра | якості |
|---|---|---|---|---|---|---|
| F1 | А | X1 | 100 | 45 | 0,16 | 7.2 |
| F3 | А | X2 | 87 | 82 | 0,28 | 22.96 |
| | Б | X3 | 27 | 10 | 0,2 | 2 |
| F4 | А | X4 | 25 | 29 | 0,36 | 10.44 |

За даними з таблиці 7 за формулою:

$$K_K = K_{TУ}[F_{1k}] + K_{TУ}[F_{2k}] + ... + K_{TУ}[F_{zk}], \quad (4.12)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 7.2 + 22.96 + 10.44 = 40.6;$$

$$K_{K2} = 7.2 + 2 + 10.44 = 19.64.$$

Як видно з розрахунків, кращим є 2 варіант, для якого коефіцієнт технічного рівня має найбільше значення.

**5.6 Економічний аналіз варіантів розробки ПП**

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;
2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як:

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{CT} \cdot K_{CT.M}, \quad (4.13)$$

де $T_P$ – трудомісткість розробки ПП;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

$K_М$ – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: ТР = 30 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: КП = 1.8. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: КСК = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта КСТ = 0.9. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T1 = 30 \cdot 1.8 \cdot 0.9 = 48{,}6 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто ТР =20 людино-днів, КП =0.9, КСК = 1, КСТ =0.8:

$$T2 = 20 \cdot 0.9 \cdot 0.8 = 14{,}4 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$TI = (48{,}6 + 14{,}4 + 4.8 + 14{,}4) \cdot 8 = 657{,}6 \text{ людино-годин.}$$

ТII = ( 48,6 + 14,4 + 6.91 + 14,4) · 8 = 674,48 людино-годин.

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 17000 грн., один аналітик в області даних з окладом 19000. Визначимо середню зарплату за годину за формулою:

$$СЧ = \frac{М}{T_m \cdot t} грн., \quad (4.14)$$

де М – місячний оклад працівників;

$T_m$ – кількість робочих днів тиждень;

$t$ – кількість робочих годин в день.

$$СЧ = \frac{17000 + 17000 + 19000}{3 \cdot 21 \cdot 8} = 105,16 грн. \quad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$СЗП = C_ч \cdot T_i \cdot КД, \quad (4.16)$$

де $С_Ч$– величина погодинної оплати праці програміста;

$T_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП}$ = 105.16 · 657,6 · 1.2 = 82 983,85 грн.

II. $С_{ЗП}$ = 105.16 · 674,48 · 1.2 = 85 113,98 грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. СВІД = СЗП · 0.22 = 82 983,85 · 0.22 = 18256,44 грн.

II. СВІД = СЗП · 0.22 = 85 113,98 · 0.22 = 18725,07 грн.

Тепер визначимо витрати на оплату однієї машино-години. (СМ)

Так як одна ЕОМ обслуговує одного програміста з окладом 17000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

СГ = 12·М·КЗ = 12 · 17000 · 0,2 = 40800 грн.

З урахуванням додаткової заробітної плати:

СЗП =СГ· (1+ КЗ) = 40800 · (1 + 0.2) = 48960 грн.

Відрахування на соціальний внесок:

СВІД= СЗП · 0.22 = 48960 · 0,22 = 10771,2 грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 10000 грн.

СА = КТМ· КА·ЦПР = 1.4 · 0.12 · 10000 = 1680 грн.,

де КТМ– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

КА– річна норма амортизації;

ЦПР– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

СР = КТМ·ЦПР · КР = 1.4 · 10000 · 0.08 = 1120 грн.,

де КР– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

ТЕФ =(ДК – ДВ – ДС – ДР) · tЗ· КВ = (365 – 104 – 12 – 16) · 8 · 0.9 =

= 1612,8 години,

де ДК – календарна кількість днів у році;

ДВ, ДС – відповідно кількість вихідних та святкових днів;

ДР – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

КВ– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

СЕЛ = ТЕФ· NC· КЗ· ЦЕН = 627,2 · 0,2 · 0,3 · 3,52 = 374,65 грн.,

де NC – середньо-споживча потужність приладу;

КЗ– коефіцієнтом зайнятості приладу;

ЦЕН – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

СН = ЦПР·0.67 = 10000· 0,67 = 6700 грн.

Тоді, річні експлуатаційні витрати будуть:

СЕКС = 48960 + 10771,2 + 1680 + 1120 + 374,65 + 6700 = 69605,85 грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

СМ-Г = СЕКС/ ТЕФ = 69376,90 / 627,2 = 110,97 грн/год.

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

I. СМ = 110,97 · 852 = 94553,8 грн.

II. СМ = 110,97 · 868,88 = 96419,61 грн.

Накладні витрати складають 67% від заробітної плати:

I. СН = 82 983,85 · 0,67 = 55599,17 грн.

II. СН = 85 113,98 · 0,67 = 57026,36 грн.

Отже, вартість розробки ПП за варіантами становить:

I. СПП = 82 983,85 + 18256,44 + 94553,8 + 55599,17 = 251 393,26 грн.

II. СПП = 85 113,98 + 18725,07 + 96419,61 + 57026,36 = 257 285,02 грн.

**5.7 Вибір кращого варіанту ПП техніко-економічного рівня**

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$K_{TEP\,j} = K_{Kj} / C_{\Phi j}, \quad (4.21)$$

$К_{ТЕР1}$ = 20,4 / 251 393,26 = $8,114 \cdot 10^{-5}$,

$К_{ТЕР2}$ = 16,08 / 257 285,02 = $6,249 \cdot 10^{-5}$.

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня КТЕР1= 6,851 ·10-5.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості

КТЕР = 8,114 · 10-5.

Цей варіант реалізації програмного продукту має такі параметри:

- Вибір програмного продукту – Eviews;
- Реалізація важливої постановки з допомогою вбудованих функцій;
- Використання стандартного інтерфейсу для побудови значень.

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, швидку реалізацію програми та доступний функціонал для роботи.

**5.8 Висновки до п’ятого розділу**

В даній частині було проведено повний функціонально-вартісний аналіз програмного продукту. Також було знайдено оцінку основних функцій програмного продукту.

В результаті виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, було визначено та проведено оцінку основних функцій програмного продукту, а також знайдено параметри, які його характеризують.

На основі аналізу вибрано варіант реалізації програмного продукту.

# ВИСНОВОК

Робота присвячена використанню платформи gRPC для реалізації взаємодії веб-сервісів. В ній досліджуються різні аспекти gRPC, включаючи віддалені виклики процедур (RPC), технологію буферів протоколів, протокол HTTP/2, концепції, архітектуру та життєвий цикл gRPC. У роботі також наведено приклади реальних проектів, які використовують gRPC, таких як використання його в Kubernetes CRI, уніфікація робочих процесів на різних мовах програмування, реалізація потокового gRPC для отримання інформації про місцезнаходження транспортних засобів, використання gRPC із зовнішніми веб-клієнтами через проксі-сервер, а також розробка API Gateway для платформи управління життєвим циклом API.

В роботі обговорюється міграція на gRPC в Dropbox і проблеми, з якими стикаються розподілені системи, включаючи надійність, мережеву безпеку, транспортні витрати і багато іншого. Висвітлюється, як платформа gRPC може допомогти вирішити ці проблеми, підкреслюється її реалізація за замовчуванням, скасування запитів і deadline, можливості потокової передачі даних, а також типові завдання, такі як обробка помилок, трасування запитів і налагодження.

Крім того, в роботі представлено реалізацію архітектури мікросервісу з використанням gRPC, що охоплює такі аспекти, як бізнес-логіка, дизайн API, діаграми архітектури додатків, розробка сервісів (стан повітря, ультрафіолетовий індекс, звітування) та результат виконання.

Нарешті, робота порівнює gRPC з іншими технологіями, зокрема з Apache Thrift, зосереджуючись на таких аспектах, як реалізація протоколу RPC, передача даних, потокове передавання, інтеграція та оптимізація.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] "Protocol Buffers" https://protobuf.dev/ (accessed May. 1, 2023)

[2] "gRPC Motivation and Design Principles" https://grpc.io/blog/principles (accessed April. 17, 2023)

[3] "Evaluating Performance of REST vs. gRPC" https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da (accessed May. 1, 2023)

[4] "Mobile Benchmarks" https://grpc.io/blog/mobile-benchmarks/ (accessed May. 1, 2023)

[5] "HTTP2 Expression of Interest" https://lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html (accessed May. 1, 2023)

[6] "HTTP/2" https://hpbn.co/http2 (accessed May. 1, 2023)

[7] "RFC 1945" https://www.rfc-editor.org/rfc/rfc1945 (accessed May. 1, 2023)

[8] "RFC 2616" https://www.rfc-editor.org/rfc/rfc2616 (accessed May. 1, 2023)

[9] "RFC 7230" https://www.rfc-editor.org/rfc/rfc7230 (accessed May. 1, 2023)

[10] "Breach" https://www.breachattack.com (accessed May. 1, 2023)

[11] "CRIME Attack Uses Compression Ratio of TLS Requests as Side Channel to Hijack Secure Sessions" https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-091312/77006 (accessed May. 1, 2023)

[12] "SPDY" https://www.chromium.org/spdy/spdy-whitepaper (accessed May. 1, 2023)

[13] "RFC 7541" https://www.rfc-editor.org/rfc/rfc7541 (accessed May. 1, 2023)

[14] "About gRPC" https://grpc.io/about (accessed April. 17, 2023)

[15] "Apache License 2.0" https://github.com/grpc/grpc/blob/master/LICENSE (accessed April. 17, 2023)

[16] "gRPC Motivation and Design Principles" https://grpc.io/blog/principles (accessed April. 17, 2023)

[17] “gRPC Project is now 1.0 and ready for production deployments” https://grpc.io/blog/ga-announcement (accessed April. 17, 2023)
[18] “Supported languages” https://grpc.io/docs/languages (accessed April. 17, 2023)
[19] “HTTP/2: Smarter at scale” https://www.cncf.io/blog/2018/07/03/http-2-smarter-at-scale (accessed April. 17, 2023)
[20] “Introduction to gRPC” https://grpc.io/docs/what-is-grpc/introduction (accessed April. 17, 2023)
[21] “Showcase” https://grpc.io/showcase (accessed April. 17, 2023)
[22] “Who we are” https://www.cncf.io/about/who-we-are (accessed April. 17, 2023)
[23] “Protocol Buffers” https://protobuf.dev (accessed May. 1, 2023)
[24] “Kubernetes: Overview” https://kubernetes.io/docs/concepts/overview (accessed May. 5, 2023)
[25] “Kubernetes: Components” https://kubernetes.io/docs/concepts/overview/components (accessed May. 5, 2023)
[26] “gRPC and temporal.io” https://www.linkedin.com/pulse/grpc-temporalio-workflow-code-mehul-thakkar (accessed May. 5, 2023)
[27] “Announcing Envoy: C++ L7 proxy and communication bus” https://eng.lyft.com/announcing-envoy-c-l7-proxy-and-communication-bus-92520b6c8191 (accessed May. 5, 2023)
[28] “Lyft’s Journey through Mobile Networking” https://eng.lyft.com/lyfts-journey-through-mobile-networking-d8e13c938166 (accessed May. 5, 2023)
[29] “gRPC-web: Using gRPC infront-endapplications” https://torq.io/blog/grpc-web-using-grpc-in-your-front-end-application (accessed May. 5, 2023)
[30] “The Architecture of Uber’s API gateway” https://www.uber.com/en-IN/blog/architecture-api-gateway
(accessed May. 5, 2023)

[31] "Fallacies of distributed systems" https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems (accessed May. 6, 2023)

[32] "The Eight Fallacies of Distributed Computing" https://www.red-gate.com/simple-talk/blogs/the-eight-fallacies-of-distributed-computing (accessed May. 6, 2023)

[33] "Fallacies of Distributed Computing Explained" https://www.se.rit.edu/~se442/doc/fallacies.pdf (accessed May. 6, 2023)

[34] "Javadoc Tool" https://www.oracle.com/ie/technical-resources/articles/java/javadoc-tool.html (accessed May. 9, 2023)

[35] "Language Guide (proto 3)" https://protobuf.dev/programming-guides/proto3 (accessed May. 9, 2023)

[36] "Protobuf overview" https://protobuf.dev/overview (accessed May. 9, 2023)

[37] "Command line tool" https://github.com/grpc/grpc/blob/master/doc/command_line_tool.md (accessed May. 9, 2023)

[38] "Evans" https://github.com/ktr0731/evans (accessed May. 9, 2023)

[39] "gRPCox" https://github.com/gusaul/grpcox (accessed May. 9, 2023)

[40] "Postman" https://www.postman.com (accessed May. 9, 2023)

[41] "Zipkin" https://zipkin.io (accessed May. 9, 2023)

[42] "Status codes and their use in gRPC" https://grpc.github.io/grpc/core/md_doc_statuscodes.html (accessed May. 9, 2023)

[43] "Language Guide (proto 3)" https://protobuf.dev/programming-guides/proto3 (accessed May. 13, 2023)

[44] "Basics tutorial: Python" https://grpc.io/docs/languages/python/basics (accessed May. 13, 2023)

[45] “Basics tutorial: Node” https://grpc.io/docs/languages/node/basics (accessed May. 13, 2023)
[46] “Basics tutorial: Golang” https://grpc.io/docs/languages/go/basics (accessed May. 13, 2023)
[47] “Basics tutorial: Java” https://grpc.io/docs/languages/java/basics (accessed May. 13, 2023)
[48] “gRPC command line tool” https://github.com/grpc/grpc/blob/38b75749ea95f58c364c995e0d7a9760f98306a1/doc/command_line_tool.md (accessed May. 13, 2023)
[49] “Apache Thrift” https://github.com/apache/thrift (accessed May. 14, 2023)
[50] “OpenAPI-Specification” https://github.com/OAI/OpenAPI-Specification (accessed May. 14, 2023)
[51] “Facebook Thrift” https://github.com/facebook/fbthrift (accessed May. 14, 2023)
[52] “Meet Bandaid, the Dropbox service proxy” https://dropbox.tech/infrastructure/meet-bandaid-the-dropbox-service-proxy (accessed May. 14, 2023)
[53] “gRPC Channelz” https://github.com/grpc/proposal/blob/master/A14-channelz.md (accessed May. 14, 2023)
[54] WHITE, J. E. A high-level framework for network-based resource sharing. In Proc. National Computer Conference, (June 1976).
[55] Frank E. Heart, Robert E. Kahn, Severo M. Ornstein, William R. Crowther, and David C. Walden. “The interface message processor for the ARPA computer network.” AFIPS Press, May, 1970.
[56] Bruce Nelson. “Thesis problems in remote procedure call.” Internal memorandum, Xerox Palo Alto Research Center, August, 1979.