

Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики
Кафедра обчислювальної математики

Дипломна робота

Дослідження віддаленого виклику процедур за допомогою протоколу gRPC

Студент групи ПМП-42:

Яцик Андрій Остапович,

спеціальність 113-прикладна математика

Науковий керівник:

асистент

Марчук Юрій Богданович

Рецензент:

Львів-2023

Зміст

1	Вступ	4
1.1	Загальна інтродукція до проблеми віддаленого виклику процедур	4
1.2	Визначення мети дослідження. Пояснення актуальності теми.	5
2	Теоритичне дослідження протоколу	6
2.1	Історія виникнення	6
2.2	Принцип роботи gRPC	7
2.3	Protobuf	8
2.4	HTTP/2	9
2.5	Виокремлення основних переваг та обмежень gRPC в порівнянні з іншими протоколами віддаленого виклику процедур.	9
3	Практичне дослідження Protobuf	11
3.1	Формулювання задачі дослідження.	11
3.2	Опис використовуваних інструментів та технологій	11
3.3	Хід дослідження	12
3.4	Результати дослідження	18
3.5	Висновок дослідження	20
4	Практичне дослідження gRPC протоколу і порівняння його з REST API	21
4.1	Формулювання конкретної задачі дослідження.	21
4.2	Опис використовуваних інструментів та технологій.	21
4.3	Пояснення процедури проведення експерименту.	22
4.4	Опис процесу реалізації прототипу системи, використовуючи gRPC.	22
4.5	Опис процесу реалізації прототипу системи, використовуючи REST API.	23
4.6	Реалізація файлу для генерації бази даних	23
4.7	Опис проведених експериментів та їх налаштувань.	24
4.8	Представлення отриманих результатів	24
4.9	Обговорення переваг та недоліків gRPC, які впливають з проведених експериментів.	25

4.10	Можливості для подальшого вдосконалення	25
5	Висновок	27
6	Додаток	29

Розділ 1

Вступ

1.1 Загальна інтродукція до проблеми віддаленого виклику процедур

Розподілена система — це набір незалежних комп'ютерів, що представляється їх користувачам єдиною об'єднаною системою. Основною частиною імплементації усіх розподілених систем є обмін інформацією між процесами. Для можливості передачі даних між різними комп'ютерами був запропонований метод при якому процес на машині А, викликає процес на машині Б, у цей момент процес на машині А призупиняється до отримання відповіді від машини Б, після отримання потрібних даних від машини Б, машина А продовжує свій процес, уже з потрібними даними. Цей метод отримав назву RPC (Remote Procedure Call). Наразі існує кілька протоколів, які використовуються для віддаленого виклику процедур (RPC). Ось декілька з них:

- XML-RPC (XML Remote Procedure Call): Цей протокол використовує XML для серіалізації даних і HTTP для передачі запитів та відповідей. Він є простим і легким у використанні, але має обмежену підтримку типів даних і безпеки.
- SOAP (Simple Object Access Protocol): SOAP використовує XML для серіалізації даних і може використовувати різні протоколи передачі, такі як HTTP, SMTP або TCP. Він надає розширену підтримку безпеки, аутентифікації та засоби для опису веб-служб. Однак, SOAP може бути складним у реалізації та має великий розмір повідомлень.
- JSON-RPC (JSON Remote Procedure Call): Цей протокол використовує JSON для серіалізації даних і може використовувати різні протоколи передачі, такі як HTTP або WebSocket. JSON-RPC є простим і компактним, і підтримує передачу даних у вигляді об'єктів JSON. Він широко використовується в веб-розробці.
- gRPC (Google Remote Procedure Call): gRPC використовує Protocol Buffers для серіалізації даних і може використовувати різні протоколи передачі, такі як HTTP/2. Він надає високу продуктивність, підтримку потокової передачі даних та автоматичне створення коду для багатьох мов програмування.
- CORBA (Common Object Request Broker Architecture): CORBA є більш складним протоколом, який надає засоби для взаємодії розподілених об'єктів. Він використовує спеціальний об'єктний брокер для знаходження та виклику віддалених процедур.

CORBA надає розширену функціональність, включаючи підтримку розподілених транзакцій, безпеку і розподілене управління об'єктами. Однак, CORBA є складним у використанні і потребує генерації специфічного коду для взаємодії з об'єктами.

Ці протоколи віддаленого виклику процедур мають свої переваги і обмеження, і вибір протоколу залежить від конкретних потреб проекту. Важливо враховувати продуктивність, безпеку, легкість використання та підтримку типів даних при виборі протоколу RPC.

Антиподом RPC є REST (Representational State Transfer, “передача репрезентативного стану”) і до порівняння цих двох підходів ми повернемося далі.

1.2 Визначення мети дослідження. Пояснення актуальності теми.

У цій роботі я б хотів виділити саме gRPC, та провести ряд досліджень для того аби висвітлити позитивні та негативні сторони цього протоколу. Та в загальному порівняти методика віддаленого виклику процедур за допомогою протоколу gRPC із загально уживаним REST API підходом.

У наш час тема веб-девелопменту розвивається надзвичайно стрімко. Так як фактично кожна людина користувалась, будь яким з продуктів цієї сфери, таких як: відео та аудіо дзвінки за допомогою різноманітних додатків, месенджери та соціальні мережі, відео хостинги, купівля в онлайн магазинах, пошук роботи на різних ресурсах, онлайн банкінг, дистанційне навчання в школі чи університеті, різні онлайн курси та багато іншого. Багато з цих ресурсів приносять дуже великі кошти для їх засновників, для прикладу дохідність платформи YouTube (<https://en.wikipedia.org/wiki/YouTube>) у 2021 році склала 28.8 мільярда доларів. Велика кількість коштів у цій сфері на мою думку є великим рушієм прогресу. Так як такі платформи потребують постійного масштабування через приріст користувачів, та імплементації нових можливостей, це змушує покращувати технології, за допомогою яких, ці платформи будуть розроблятися чи підтримуватись. Важливою складовою розробки будь яких веб сервісів є побудова архітектури на якій буде відбуватись спілкування між клієнтом і сервером, чи між серверами загалом, якщо ми будемо платформу на базі мікросервісної архітектури. У випадку вибору архітектурного підходу віддаленого виклику процедур, нам також потрібно оприділитись із потрібним протоколом, який підійде під задачі платформи найкраще. Саме з цих причин я вважаю що дослідження одного із найсучасніших підходів до розробки веб платформ, наразі є актуальною та важливою темою.

Розділ 2

Теоритичне дослідження протоколу

2.1 Історія виникнення

gRPC був розроблений компанією Google, як удосконалення їхньої інфраструктури зв'язку та 'спілкування' між серверами під назвою 'Stubby'. Здавалось, для чого покращувати те, що і так працює? Таке рішення було прийняте знову ж таки через масштабування, потребу в імплементації нових функцій та залучення нових працівників. Попередня версія інфраструктури масштабуванню не заважала, так як за словами Варуна Талвара, продукт менеджера компанії Google їх попередня версія інфраструктури могла обраблювати десятки мільярдів запитів на секунду, ця цифра навіть зараз вражаюча а на 2015 рік це було щось неймовірне. Але з'являлась проблема, у залученні нових фахівців та імплементації нових функцій. Проблема із залученням нових фахівців заключалась у тому, що уся інфраструктура працювала за правилами які були написані фахівцями з Google і використовувались виключно у проектах цієї компанії, тобто якщо компанії потрібно було найняти певну кількість нових співробітників, то не залежно від їхнього досвіду та рівня знань, їм всерівно потрібно було витратити певну кількість часу для того, щоб навчитись працювати з цієї інфраструктурою, а це все втрачений час та гроші. Проблема імплементації нових функцій заключалась у тому, що з'явлюлись нові технології такі як:

- SPDY (читається як «speedy», «спіді») — протокол прикладного рівня для передачі вебвмісту, розроблений корпорацією Google. Основним завданням SPDY є зниження часу завантаження вебсторінок та їх елементів.
- HTTP/2 — друга, розширена версія HTTP 2015 року і стандартизована в 2016 році, що базується на попередній версії HTTP 1.1.
- QUIC (англ. Quick UDP Internet Connections) — транспортний мережевий протокол, який розвивається компанією Google з 2013 року як альтернатива зв'язці TCP + TLS для веб. Він вирішує проблеми з великим часом встановлення і узгодження з'єднань в TCP і усуває затримки при втраті пакетів в процесі передачі даних, та багато інших. Так як 'Stubby' не підтримував їх потрібно було покращувати інфраструктуру.

Ось слова Луї Райана, Головного інженера компанії Google з 2007 по 2023 рік, про розробку gRPC.

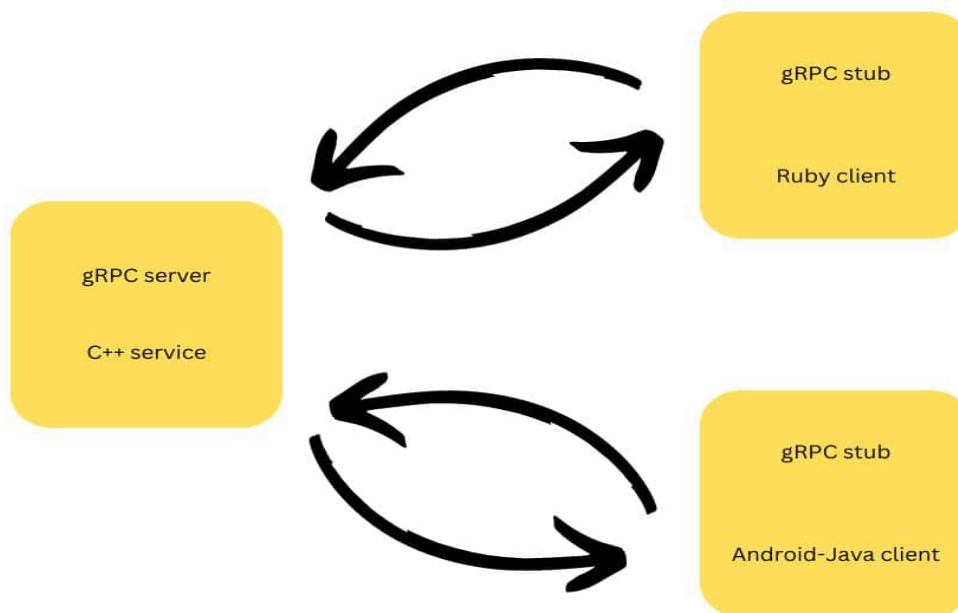
"Google використовує єдину універсальну RPC-інфраструктуру під назвою Stubby для з'єднання великої кількості мікросервісів, що працюють у наших дата-центрах і

між ними, вже більше десяти років. Наші внутрішні системи вже давно використовують архітектуру мікросервісів, яка сьогодні набуває популярності. Наявність єдиної кросплатформної RPC-інфраструктури дозволила впровадити покращення ефективності, безпеки, надійності та поведінкового аналізу, що є критично важливими для підтримки неймовірного зростання, яке ми спостерігали в цей період.

Stubby має багато чудових функцій, але він не базується на жодному стандарті і занадто тісно пов'язаний з нашою внутрішньою інфраструктурою, щоб вважатися придатним для публічного релізу. З появою SPDY, HTTP/2 і QUIC багато з цих можливостей з'явилися у відкритих стандартах, разом з іншими можливостями, яких Stubby не надає. Стало зрозуміло, що настав час переробити Stubby, щоб скористатися перевагами цієї стандартизації і розширити його застосовність до мобільних, IoT і хмарних сценаріїв використання. [5]"

2.2 Принцип роботи gRPC

Ідея gRPC не нова – парадигма, яку використовує даний фреймворк, вже давно відома, вона ґрунтується на RPC (віддаленому виклику процедур). За допомогою RPC відбувається комунікація між клієнтом та сервером, для якої використовується не HTTP-запит, а виклик функції. Клієнт викликає віддалену процедуру, серіалізує параметри та додаткову інформацію у повідомленні, після чого надсилає повідомлення на сервер. Приймаючи дані, сервер здійснює їх десеріалізацію, виконує потрібну операцію і надсилає результат назад клієнту. Такі об'єкти як stub сервера та stub клієнта беруть на себе функції серіалізації та десеріалізації параметрів.[1]

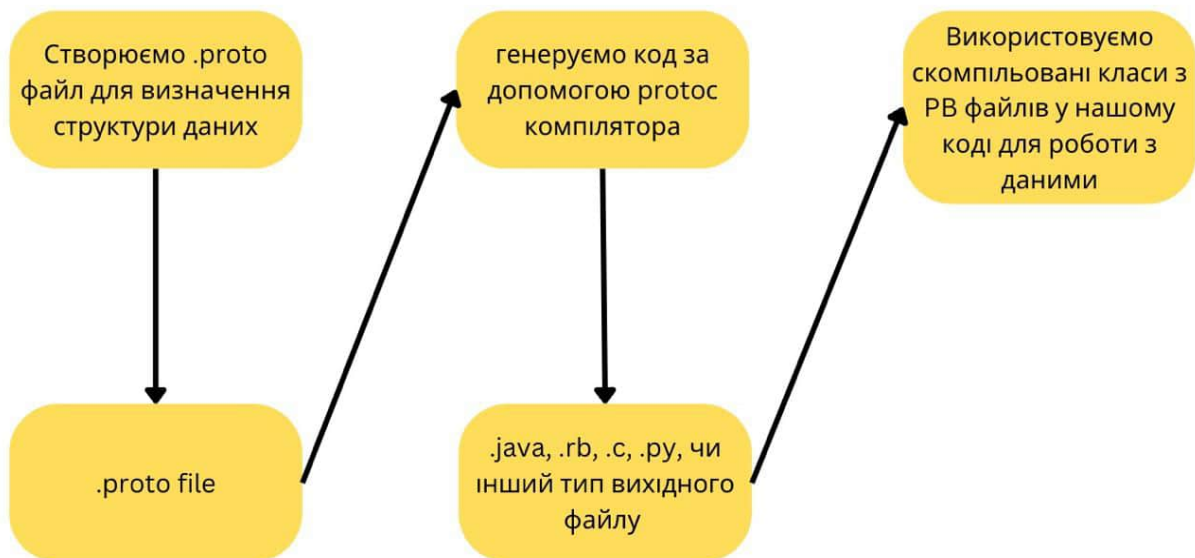


2.3 Protobuf

Protocol Buffers - це безкоштовний крос-платформний формат даних з відкритим вихідним кодом, який використовується для серіалізації структурованих даних. Він корисний при розробці програм для спілкування між собою через мережу або для зберігання даних. Метод включає мову опису інтерфейсу, яка описує структуру деяких даних, і програму, яка генерує вихідний код з цього опису для генерації або розбору потоку байт, що представляє структуровані дані. У gRPC Protobuf виконує роль єдиного варіанту для серіалізації та десеріалізації даних. Для прикладу найбільш поширеним серіалізатором даних є JSON. Основна відмінність між цими двома варіантами серіалізації даних є те що Protobuf - бінарний, а JSON - текстовий формат передачі даних. Розробники заявляють, що Protobuf є швидший за JSON формат. Це твердження потрібно перевірити тому, для цього в наступних частинах будуть проведені експерименти, за допомогою яких ми зможемо зрозуміти, чи є це твердження вірне. Поки будемо базуватись на інформації від розробників. Алгоритм використання Protobuf. Для початку ми створюємо файл з типом даних .proto, у якому описуємо наші класи для роботи з даними. Приклад коду, який створює нам клас Person, з полями name, id і email.

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

Далі за допомогою компілятора ми генеруємо файл з типом мови програмування яку ми використовуємо у проекті, перелік мов програмування які підтримують використання Protobuf: C++, C#, Java, Kotlin, Objective-C, PHP, Python, Ruby.



Після цих кроків ми можемо застосовувати зкомпільовані класи для роботи з даними у нашому проекті.

Великим плюсом є підтримка багатьох мов програмування, що дозволяє нам при побудові мікросервісів на різних мовах програмування, просто згенерувати для кожного

мікросервісу файли з описом класів для роботи з даними, при цьому генерація файлів для різних мікросервісів може відбуватись за допомогою одного і того ж файлу `.proto`. Для прикладу візьмемо кілька мікросервісів які працюють на мовах програмування Ruby, Python і C#. То для них можна створити файл `example.proto`, у якому описати потрібні для роботи класи. Та згенерувати за допомогою нього, файли: `example_pb.rb`, `example_pb.py`, `example_pb.cs`. Це дозволяє нам мати однакову класифікацію даних на усіх мікросервісах і економить час написання класів на кожній з цих мов. Також перевагою Protobuf є те, що він 'легший' за конкурентів, тобто однакова кількість даних закодована за допомогою Protobuf, буде потребувати менше місця на жорсткому диску для зберігання, ніж вона була б закодована за допомогою інших засобів. Це твердження підтверджує перехід компанії Twitter на даний протокол. В 2010 році бекенд Twitter перейшов на Protobuf. Як заявили розробники: база в триліон твітів(постів) на XML займала б 10 петабайт замість одного на Protobuf.

2.4 HTTP/2

HTTP/2 був опублікований у травні 2015 року. З технічної точки зору, однією з найважливіших особливостей, що відрізняють HTTP/1.1 і HTTP/2, є двійковий рівень кадрування. На відміну від HTTP/1.1, який зберігає всі запити і відповіді у форматі звичайного тексту, HTTP/2 використовує двійковий рівень кадрування для інкапсуляції всіх повідомлень у двійковому форматі, зберігаючи при цьому семантику HTTP, таку як дієслова, методи і заголовки. API на рівні додатку все одно буде створювати повідомлення у звичайних форматах HTTP, але базовий рівень перетворить ці повідомлення у двійковий формат. Це гарантує, що веб-додаток, створений до HTTP/2, може продовжувати функціонувати як зазвичай при взаємодії з новим протоколом. [2]

Таким чином, HTTP/1.1 і HTTP/2 мають спільну семантику, гарантуючи, що запити і відповіді, які передаються між сервером і клієнтом в обох протоколах, досягають місця призначення у вигляді традиційно відформатованих повідомлень з заголовками і тілами, використовуючи знайомі методи, такі як GET і POST.

2.5 Виокремлення основних переваг та обмежень gRPC в порівнянні з іншими протоколами віддаленого виклику процедур.

Плюси gRPC:

- **Продуктивність:** gRPC використовує HTTP/2 протокол, який надає багатоканальну передачу даних та компресію, що забезпечує високу швидкість передачі та ефективне використання мережевого пропускового здатності.
- **Підтримка різних мов програмування:** gRPC підтримує багатомовний підхід, що означає, що ви можете використовувати його з різними мовами програмування.
- **Передача даних в форматі Protocol Buffers:** gRPC використовує Protocol Buffers для серіалізації даних, що дозволяє компактно представляти структури даних та забезпе-

чує переваги, такі як ефективність у використанні мережевого пропускового здатності та підтримка версіонування даних.

- Підтримка потокової передачі даних: gRPC надає можливість встановлювати біди-рекціональні потоки для передачі поточкових даних, що дозволяє ефективно використовувати мережеві ресурси та реалізовувати різноманітні сценарії взаємодії.

Мінуси gRPC:

- Складність реалізації: Порівняно з деякими іншими протоколами RPC, реалізація gRPC може бути складнішою через використання Protocol Buffers та HTTP/2. Потрібно мати певні знання та досвід для успішного впровадження.
- Вимоги до мережевих ресурсів: Використання HTTP/2 та додаткових мережевих функцій може вимагати більших ресурсів у порівнянні з простішими протоколами RPC. Це може вплинути на продуктивність та вимоги до мережі.
- Обмежена підтримка старіших браузерів: HTTP/2, який використовується gRPC, може мати обмежену підтримку в старіших версіях браузерів. Це може створювати проблеми, якщо ваші додатки спрямовані на веб-браузери зі старішою версією.

Це лише загальні плюси і мінуси gRPC. Вибір протоколу RPC залежить від конкретних потреб вашого проекту та його контексту.

Розділ 3

Практичне дослідження Protobuf

3.1 Формулювання задачі дослідження.

За твердженням розробників Protobuf є швидшим ніж один з найпоширеніших серіалізаторів даних JSON. Це дослідження проводиться для того, щоб перевірити це твердження на практиці, та зрозуміти яка різниця між швидкістю роботи цих рішень на практиці.

3.2 Опис використовуваних інструментів та технологій

Дослідження проводиться за допомогою мови програмування Ruby, версії 3.1.2. Додаткові бібліотеки у цій мові програмування називаються гемами і найкращий офіційний ресурс для пошуку і оцінки потрібних гемів - це <https://rubygems.org>. [3]

Перелік гемів які я обрав з вище вказаного ресурсу, їх функція у цьому дослідженні і відомість по популярності:

benchmark-ips - Benchmark надає функціонал для створення тестів на оцінку продуктивності різних рішень, це можуть бути частини коду, функції, класи та навіть інші геми. Гем який використано у дослідженні є надбудовою до гему benchmark, ips у назві цього гему означає iterations per second, що дозволяє нам не просто оцінювати час за який виконається код, а я кількість ітерацій виконання цього коду за певний період часу. Версія - 2.12.0. Загальна кількість завантажень цього гему - 42,531,940, що у купі з постійними оновленнями версій стало поштовхом до використання цього гему.

google-protobuf - офіційний гем від розробників Protobuf. У цьому дослідженні це єдиний представник гемів для роботи з Protobuf, так як це офіційний гем від розробників протоколу і поки що ніяких аналогів, які могли б з ним конкурувати немає. Версія - 3.23.2. Загальна кількість завантажень - 204,859,557.

Представниками JSON формату будуть 3 геми так як всі вони поширені у різних проектах, тому для чистоти даних було прийняте рішення використовувати всі 3. Всі 3 геми написані на розширені RubyC, із застосуванням мови програмування C, що вказує на їх швидкодію у порівнянні з іншими JSON бібліотеками які написані на Ruby.

json - найпоширеніший із всієї трійки. Версія - 2.6.3. Кількість завантажень - 682,628,285.

oj - Версія - 3.14.3. Кількість завантажень - 157,727,921.

yajl-ruby - Версія - 1.4.3. Кількість завантажень - 60,912,237.

Характеристики системи на якій проводилось дослідження: Процесор - Apple Silicon M1 Pro. 8-ядерний процесор з 6 ядрами продуктивності і 2 ядрами ефективності.

3.3 Хід дослідження

Спершу створимо Gemfile, він потрібний для опису гемів, які ми будемо використовувати у нашому проекті.

```
source :rubygems

gem 'benchmark-ips'
gem 'google-protobuf'
gem 'oj'
gem 'yajl-ruby'
```

source у цьому файлі це посилання на ресурс з якого ми будемо завантажувати геми. У нашому випадку ми завантажуюмо геми з офіційного ресурсу rubygems, тому нам не потрібно вставляти увесь шлях до ресурсу, а достатньо використати заготовлене ключовий символ. Після чого ми просто перелічуємо геми, які нам потрібні. Далі для встановлення цих гемів використовуємо команду:

```
GEM
  remote: http://rubygems.org/
  specs:
    benchmark-ips (2.12.0)
    google-protobuf (3.22.2-arm64-darwin)
    google-protobuf (3.22.2-x86_64-linux)
    oj (3.14.2)
    yajl-ruby (1.4.3)
```

```
PLATFORMS
  arm64-darwin-22
  x86_64-linux
```

```
DEPENDENCIES
  benchmark-ips
  google-protobuf
  oj
  yajl-ruby
```

```
BUNDLED WITH
  2.4.6
```

bundle install, вона встановить всі необхідні геми та їх залежності і створить файл Gemfile.lock у якому ми зможемо побачити їх перелік.

Потрібно визначитись із структурою даних які будемо тестувати. Для дослідження швидкості роботи протоколів найкраще підійдуть комплексні структури даних, для того щоб охопити якомога більше різних типів. Тому використовуємо наступну структуру даних:

```
syntax = "proto3";

message User {
  int32 user_id = 1;
  string username = 2;
  string email = 3;
  string date_joined = 4;
  bool is_active = 5;
  Profile profile = 6;
  repeated Post posts = 7;
}

message Profile {
  string full_name = 1;
  int32 age = 2;
  string address = 3;
  string phone_number = 4;
}

message Post {
  int32 post_id = 1;
  string title = 2;
  string content = 3;
  string date_created = 4;
  int32 likes = 5;
  repeated string tags = 6;
  repeated Comment comments = 7;
}

message Comment {
  int32 comment_id = 1;
  string author = 2;
  string content = 3;
  string date_created = 4;
  int32 likes = 5;
}

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

1. користувач(User)

Ідентифікатор (user_id) - тип: integer,
Псевдонім (username) - тип: string,
Адрес електронної пошти (email) - тип: string,
Дата реєстрації (date_joined) - тип: string,

Онлайн (is_active) - тип: boolean,
Профіль користувача (profile) - клас під номером 2, який ми створили власноруч.
Пости (posts) - масив постів, клас для цих постів знаходиться під номером 3,

2. профіль(Profile)

Прізвище і ім'я (full_name) - тип: string,
Вік (age) - тип: integer,
Адрес (address) - тип: string,
Номер телефону (phone_number) - тип: string,

3. пост(Post)

Ідентифікатор поста (post_id) - тип: integer,
Заголовок (title) - тип: string,
Вміст (content) - тип: string,
Дата створення (date_created) - тип: string,
Кількість лайків (likes) - тип: integer,
Теги (tags) - масив типу - string,
Коментарі (comments) - масив класу під номером 4.

4. коментар(Comment)

Ідентифікатор коментаря (comment_id) - тип: integer,
Автор (author) - тип: string,
Вміст (content) - тип: string,
Дата створення (date_created) - тип: string,
Кількість лайків - (likes) тип: integer,

Опираючись на цю структуру даних створюємо файл user.proto, у якому описуємо потрібні нам класи за допомогою команди 'message'. За допомогою пакету protobuf, який встановлений на нашу систему, генеруємо із файлу user.proto файл user_pb.rb, який будемо використовувати під час тестування. Для цього в терміналі, який відкритий у корні нашого проекту, виконуємо команду: `protoc -ruby_out=. user.proto` Після чого отримуємо файл user_pb.rb із наступним вмістом

```
# Generated by the protocol buffer compiler. DO NOT EDIT!
# source: user.proto

require 'google/protobuf'

Google::Protobuf::DescriptorPool.generated_pool.build do
  add_file("user.proto", :syntax => :proto3) do
    add_message "User" do
      optional :user_id, :int32, 1
      optional :username, :string, 2
      optional :email, :string, 3
      optional :date_joined, :string, 4
      optional :is_active, :bool, 5
      optional :profile, :message, 6, "Profile"
      repeated :posts, :message, 7, "Post"
    end
    add_message "Profile" do
      optional :full_name, :string, 1
      optional :age, :int32, 2
      optional :address, :string, 3
    end
  end
end
```

```

    optional :phone_number, :string, 4
  end
  add_message "Post" do
    optional :post_id, :int32, 1
    optional :title, :string, 2
    optional :content, :string, 3
    optional :date_created, :string, 4
    optional :likes, :int32, 5
    repeated :tags, :string, 6
    repeated :comments, :message, 7, "Comment"
  end
  add_message "Comment" do
    optional :comment_id, :int32, 1
    optional :author, :string, 2
    optional :content, :string, 3
    optional :date_created, :string, 4
    optional :likes, :int32, 5
  end
end
end

User = ::Google::Protobuf::DescriptorPool.generated_pool.lookup("User").msgclass
Profile = ::Google::Protobuf::DescriptorPool.generated_pool.lookup("Profile").msgclass
Post = ::Google::Protobuf::DescriptorPool.generated_pool.lookup("Post").msgclass
Comment = ::Google::Protobuf::DescriptorPool.generated_pool.lookup("Comment").msgclass

```

Усі підготовчі кроки виконано, тепер починаємо тестування:

```

require 'benchmark/ips'
require 'yajl'
require 'oj'
require 'json'
require_relative 'user_pb

```

Створюємо файл `benchmark.rb`, у якому одразу імпортуємо необхідні геми.

Далі описуємо хеш із даними, які будемо серіалізувати, змінна `data`, хеш описується за вище вказаною структурою даних. Підготовлюємо змінні для тестування.

```

proto_model = User.new(data)
proto_encoded_data = User.encode(proto_model)
json_encoded_data = JSON.dump(data)

```

`proto_model` - це вище створена модель Protobuf `User`, з ініціалізованими у неї даними. `proto_encoded_data` - це серіалізовані дані за допомогою Protobuf. `json_encoded_data` - це серіалізовані дані за допомогою JSON. Виводимо розмір у байтах серіалізованих даних для кожного протоколу. Розмір даних це важливий аспект нашого дослідження, тому що від розміру даних залежить також навантаження на мережу під час передачі цих даних. Розмір серіалізованих даних за допомогою гемів `Oj` та `YAJL`, можна не виводити, так як він буде однаковий, у цих гемах є різниця тільки у швидкості роботи, а не у розмірі даних.

```
puts "JSON payload bytesize #{json_encoded_data.bytesize}"
puts "Protobuf payload bytesize #{proto_encoded_data.bytesize}"
```

Пишемо тести кількості операцій на секунду для серіалізації даних.

```
Benchmark.ips do |x|
  x.config(time: 10)

  x.report('Yajl encoding') do
    Yajl::Encoder.encode(data)
  end

  x.report('Oj encoding') do
    Oj.dump(data)
  end

  x.report('standard JSON encoding') do
    JSON.dump(data)
  end

  x.report('protobuf encoding') do
    User.encode(proto_model)
  end

  x.report('protobuf with model init') do
    User.new(data).to_proto
  end

  x.compare!
end
```

У Benchmark.ips передаємо блок використовуючи допоміжну змінну 'x' для конфігурації тестів. x.config - потрібен для конфігурації часу на кожен тест, time - це час проходження одного тесту. x.report - функція у яку передається блок який буде тестуватись. Тобто саме код який ми передаємо у цю функцію буде повторюватись на протязі n часу. x.compare! - формує звіт по результатах тестування та виводить його в консоль. Наповнення тестів: перших 3 тести це тести JSON протоколу, для різних гемів. Далі тест серіалізації Protobuf із створеним інстансом класу з файлу user_pb.proto, тобто це тест саме серіалізації даних. Проте на практиці для передачі даних за допомогою Protobuf перед серіалізацією даних у більшості випадків у нас не буде створеного інстансу цього класу, тому останній тест протоколу Protobuf, буде найбільш наближений до практичного його застосування.

```
Benchmark.ips do |x|
  x.config(time: 10)

  x.report('Yajl parsing') do
    Yajl::Parser.parse(json_encoded_data)
  end

  x.report('Oj parsing') do
    Oj.load(json_encoded_data)
  end

  x.report('standard JSON parsing') do
    JSON.parse(json_encoded_data)
  end

  x.report('protobuf parsing') do
    User.decode(proto_encoded_data)
  end

  x.compare!
end
```

Для створення тестів на десеріалізацію даних застосовуємо те саме, тільки тут нам не потрібно створювати тест з наповненням інстансу, так як для десеріалізації даних нам він не потрібний.

3.4 Результати дослідження

Запустивши файл з тестами ми отримаємо наступні результати:

```
JSON payload bytesize 1684
Protobuf payload bytesize 873

Encoding...

Warming up
Yajl encoding      14.922k 1/100ms
  Oj encoding      26.951k 1/100ms
standard JSON encoding  15.789k 1/100ms
  protobuf encoding 186.027k 1/100ms
  protobuf with model init 14.119k 1/100ms

Calculating
Yajl encoding      145.941k (± 5.3%) 1/s - 1.462M in 10.054306s
  Oj encoding      269.023k (± 2.2%) 1/s - 2.695M in 10.023315s
standard JSON encoding  155.039k (± 2.5%) 1/s - 1.563M in 10.036955s
  protobuf encoding 1.086M (± 1.8%) 1/s - 10.921M in 10.056049s
  protobuf with model init 140.443k (± 1.3%) 1/s - 1.412M in 10.054822s

Comparison:
  protobuf encoding: 1086269.1 1/s - 4.04x slower
  Oj encoding:      269022.5 1/s - 6.97x slower
standard JSON encoding  155838.9 1/s - 7.44x slower
  Yajl encoding:    145941.1 1/s - 7.73x slower
  protobuf with model init: 140443.2 1/s - 7.73x slower

Decoding...

Warming up
Yajl parsing       5.245k 1/100ms
  Oj parsing       10.002k 1/100ms
standard JSON parsing  9.528k 1/100ms
  protobuf parsing 58.391k 1/100ms

Calculating
Yajl parsing       53.284k (± 0.6%) 1/s - 534.500k in 10.031445s
  Oj parsing       188.452k (± 1.4%) 1/s - 1.091M in 10.061815s
standard JSON parsing  94.930k (± 0.8%) 1/s - 952.800k in 10.037547s
  protobuf parsing 499.413k (± 1.5%) 1/s - 5.039M in 10.092364s

Comparison:
  protobuf parsing: 499412.7 1/s - 4.60x slower
  Oj parsing:      188452.1 1/s - 5.26x slower
standard JSON parsing  94929.5 1/s - 9.37x slower
  Yajl parsing:    53284.2 1/s - 9.37x slower
```

Для початку звернемо увагу на перші два рядки результатів. Тут ми бачимо розмір серіалізованих даних у байтах і розмір даних Protobuf у 1.93 рази менший за JSON, що є доволі великою різницею. Далі секція серіалізації, спершу бачимо підсекцію Warmup. Warmup - це так звана 'розминка', потрібна для того, щоб всі компоненти комп'ютера вийшли на максимальний рівень продуктивності. Для прикладу, якщо перед запуском тестів комп'ютер використовувався не активно то процесор, для збереження електроенергії буде працювати не на всю потужність, хоча для отримання повної потужності нам потрібно досить мало часу, але це всерівно буде вносити похибку у результат тестування, особливо для першого виконаного тесту. Наступна підсекція це вже обчислення кількості операцій. У записі для кожного тесту ми бачимо: кількість операції на секунду, похибку, та загальну кількість операцій за 10 секунд. І остання підсекція це порівняння результатів, де ми бачимо, що серіалізація Protobuf є набагато швидшою за будь який JSON, але наюлижений до практичного застосування тест він найповільніший, тому що наповнення моделі даними займає велику частину часу.



Секція десеріалізації має таку саму структуру, як і серіалізації, тому тут звернемо увагу одразу на порівняння результатів. Ми бачимо, що десеріалізація Protobuf у 4.6 разів швидша за найшвидшу десеріалізацію JSON.



3.5 Висновок дослідження

Із цього дослідження можна виділити кілька ключових моментів:

1. Розмір серіалізованих даних Protobuf є практично у 2 рази менший, що на дистанції буде дуже помітно зменшувати навантаження на мережу для передачі цих даних, та зменшувати час запиту/відповіді.
2. Час серіалізації Protobuf є найшвидшим, але наближаючи тест до практичного застосування він поступається усім JSON тестам, хоча у одному випадку практично дорівнює JSON-у.
3. Із десеріалізацією Protobuf справляється найкраще, та ми бачимо велику різницю між часом виконання Protobuf та JSON.

Враховуючи ці пункти можна зробити висновок, що все таки твердження розробників, що Protobuf є швидшим за JSON, можна вважати вірним, навіть якщо брати тест який при серіалізації є найповільнішим, велика різниця у швидкості десеріалізації нівелює розрив між швидкістю серіалізації. Також перевагою є вдвічі менший розмір серіалізованих даних Protobuf, хоча у випадку тестування швидкості роботи протоколів це нам не змінює висновок, але якщо розглядати використання протоколу у будь якому проекті, це серйозна перевага, яка буде відчутно зменшувати навантаження на мережу.

Розділ 4

Практичне дослідження gRPC протоколу і порівняння його з REST API

4.1 Формулювання конкретної задачі дослідження.

Задача дослідження: зрозуміти складність написання платформ на gRPC на мові програмування Ruby та порівняти швидкість роботи і зручність написання gRPC та REST API.

4.2 Опис використовуваних інструментів та технологій.

Мова програмування Ruby - версії 3.1.2. Характеристики системи як у дослідженні Protobuf. Геми для проекту gRPC: grpc - офіційний гем від розробників gRPC для Ruby. Версія - 1.55.0, кількість завантажень - 108,462,593. Залежності - google-protobuf - версії 3.23 і googleapis-common-protos-types - версії 1.0.0. yamml - гем для роботи з файлами розширення *.yaml. Версія - 0.2.1, кількість завантажень - 2,105,897. Геми для REST API проекту: rails - гем для реалізації REST платформ на мові Ruby. Версія - 7.0.4.3, кількість завантажень - 439,361,637. Залежності та їх версії:

```
actioncable=7.0.4.3  
actionmailbox=7.0.4.3  
actionmailer=7.0.4.3  
actionpack=7.0.4.3  
actiontext=7.0.4.3  
actionview=7.0.4.3  
activejob=7.0.4.3  
activemodel=7.0.4.3  
activerecord=7.0.4.3  
activestorage=7.0.4.3  
activesupport=7.0.4.3  
bundler>=1.15.0  
mailties=7.0.4.3
```

yaml - гем для роботи з файлами розширення *.yaml. Версія - 0.2.1, кількість завантажень - 2,105,897. rupa - гем який запускає локальний сервер для проектів на базі rails. Версія - 5.0, кількість завантажень - 288,081,321.

Основу сервера на gRPC, візьмемо з офіційного gitHub-у gRPC, але перепишемо його для проведення дослідження. REST API сервер створимо самостійно. REST API створений за допомогою гему rails базується на архітектурному шаблоні MVC - це шаблон у якому увесь процес обробки запиту проходить через 3 основних кроки:

1. Model - моделі це дані, для прикладу таблиці і записи бази даних
2. View - відображення користувацького інтерфейсу, для моголітних проектів це звичай HTML сторінка, а так як наш проект API, то у нас це просто дані у JSON форматі.
3. Controller - виконує основну функцію обробки бізнес логіки.

Якщо спрощено описати обробку запиту, він буде виглядати таким чином: приходять запит який після обробки url(шляху, посилання) потрапляє в певний контролер, який маніпулює параметрами запиту та зберігає, оновлює чи просто дістає дані, які зберігаються у моделях, після чого формує вигляд (для прикладу JSON чи HTML) і відправляє його на сторону клієнта.

4.3 Пояснення процедури проведення експерименту.

Першочергово створюємо сервери на базі gRPC і REST API. Після чого створюємо простий скрипт, за допомогою якого ми створимо файл db.yaml, який будемо використовувати в якості бази даних. У файл бази даних будемо створювати n кількість записів (0, 1.000, 10.000, 50.000, 100.000 і 200.000) для порівняння швидкості обробки запиту. Для виконання запитів будемо використовувати програму Postman. Це програма яка дає можливість виконувати запити на локальні чи віддалені сервери, а нам вона особливо корисна, бо після виконання запиту, ми можемо побачити час, який був витрачений на обробку цього запиту.

4.4 Опис процесу реалізації прототипу системи, використовуючи gRPC.

За основу gRPC сервера візьмемо сервер із прикладів серверів на gRPC.[4] Його ми отримаємо з офіційного gitHub-у розробників у вкладці examples/ruby, нам підійде greeter_server.rb, також для його роботи нам потрібен прото файл helloworld.proto. Почнемо з прото файлу до нього ми додамо у клас відповіді масив користувачів, з полями ім'я, вік та дайджест (це строка згенерована випадковим чином, яка виступатиме заміною id), кінцевий код прото файлу можна знайти у додатках під номером 1.1. Далі генеруємо helloworld_pb.rb файл, як ми це робили у дослідженні Protobuf. І отримуємо файл який будемо використовувати для відповіді клієнту, приклад коду цього файлу ми бачимо під номером 1.2. Наступним кроком нам потрібно оновити код у файлі сервера. У файлі greeter_server.rb ми бачимо клас GreeterServer та метод say_hello, цей метод відповідає за обробку запиту на виклик процедури SayHello, яку ми будемо виконувати через Postman.

Наразі робота цього методу полягає у тому що він отримує змінну name від клієнта та відповідає повідомленням 'Hello name', де name - це значення отриманої змінної. Для того, щоб у відповідь додати масив користувачів, у цей файл нам потрібно імпортувати гем 'yaml' та з його допомогою у методі say_hello зчитати дані користувачів з файлу db.yml у змінну users та передати її до класу відповіді HelloWorld::HelloReply. Також потрібно перевірити порт на якому буде запущено сервер. Його можна побачити у функції main у параметрах методу 'add_http2_port' класу 'GRPC::RpcServer'. Основна вимога до порту, щоб він не був зайнятий, тобто якщо ми розуміємо, що якась інша програма запущена на нашому комп'ютері використовує порт, який ми знайшли вище, то нам потрібно його замінити на вільний порт, потрібно пам'ятати, що гем rupa, який ми будемо використовувати для запуску REST API сервера, заумовчуванням використовує порт 3000, тому якщо ми хочемо запускати 2 сервера одночасно, для того щоб вони не конфліктували потрібно використовувати різні порти. Кінцевий код сервера gRPC можна знайти у додатку 1.3.

4.5 Опис процесу реалізації прототипу системи, використовуючи REST API.

Створюємо проект за допомогою команди: 'rails new rest --skip-active-record --skip-javascript --api --skip-test', ця команда створить нам порожній проект rails із назвою 'rest' та додаткові параметри: skip-active-record - дозволить нам пропустити створення бази даних, так як ми вокористовуватимемо yaml файл у якості бази даних, skip-javascript - пропускаємо генерацію javascript коду, він нам також не знадобиться, api - створить нам проект який буде налаштований для роботи у якості API додатку, заумовчування створюється монолітний проект, skip-test - дозволить пропустити створення тестів, нам вони не потрібні, тому що тестування серверів, які представлені у цьому дослідженні ми проводити не будемо. Після створення проекту відкриваємо файл config/routes.rb. У ньому видаляємо увесь код всередині Rails.application.routes.draw і вписуємо root "users#index". Це означає, що корнем нашого додатку буде контроллер users із методом index(GET).

```
Rails.application.routes.draw do
  root "users#index"
end
```

Далі створюємо файл users_controller.rb, у папці app/controllers. У ньому створюємо клас UsersController, який учпадкоуємо від ActionController::API, це потрібно для правильної поведінки нашого контролера, саме у API сервері. У цьому класі створюємо метод index, у ньому ми будемо описувати поведінку обробки запиту, який описали у файлі routes.rb. У цей метод вставляємо наступний код:

```
users = YAML.load_file("#Rails.root/app/controllers/db.yml")
render json: JSON.dump(name: "Hello, #params[:name] users:)
```

Цей код зчитає файл з даними користувачів, запише ці дані у змінну users та згенерує відповідь у JSON форматі, такої ж структури як і наш сервер gRPC.

4.6 Реалізація файлу для генерації бази даних

Для генерації вмісту файлу бази даних напишемо простий скрипт на мові Ruby. require 'yaml' require 'securerandom'

```

n = 1000 # Кількість створених користувачів.
names = [...] # Тут масив імен який взятий з https://gist.github.com/ruanbekker/a1506f06aa1
df06c5a9501cb393626ea,
у самій роботі його не наведено тому що там дуже багато значень і їх усіх записувати у
роботу немає сенсу.
n.times do |index|
arr « name: names[rand(0...names.size)], age: rand(15..60), digest: SecureRandom.hex
end
File.open("db.yml "w") |file| file.write(arr.to_yaml)

```

Після запуску цього коду ми отримаємо файл, у якому буде n кількість записів користувачів. У яких ім'я це випадкове значення з масиву імен, вік це випадкове число від 15 до 60 включно і дайджест це випадково згенерована стрічка за допомогою методу `hex`, класу `SecureRandom`.

4.7 Опис проведених експериментів та їх налаштувань.

Для початку запустимо наші сервера, gRPC сервер запускається звичайним викликом файлу `greeter_server.rb`, REST API сервер запускається командою `rails server`. Після запуску серверів заходимо у Postman і створюємо 2 файли запитів, типи запитів обираємо відповідно до протоколів роботи наших серверів gRPC і HTTP. Для HTTP запиту нам потрібно тільки вписати URL, `0.0.0.0:3000`, `0.0.0.0`. вказує на те що запит буде робитись на localhost (локальну мережу), 3000 - це порт. Також для HTTP нам потрібно відправити параметр `name`, для цього заходимо у вкладку `Body` та обираємо пункт `raw` та `JSON`, у полі для вводу пишемо:

```
{"name": "John"}
```

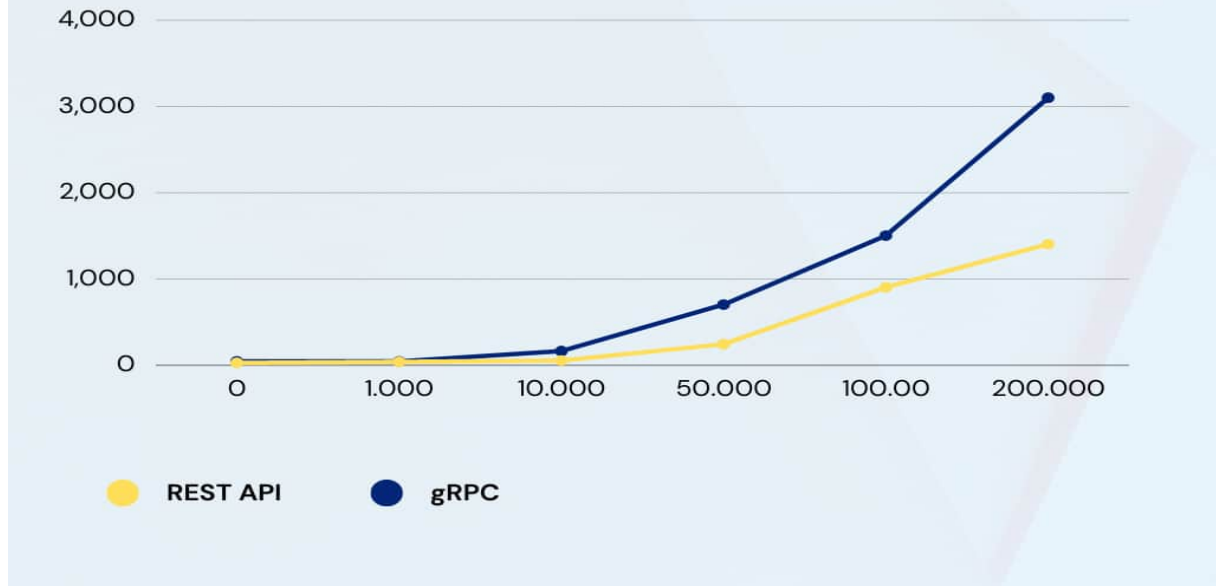
Далі нам потрібно налаштувати gRPC запит. Відкриваємо файл запиту, та вписуємо URL нашого запиту: `0.0.0.0:12345`. Далі заходимо у розділ вибору методу який ми будемо викликати та обираємо імпорт методів із `proto` файлу, завантажуюмо файл `helloworld.proto`, з нього Postman отримає інформацію про методи які він може викликати, та те як отримані дані десеріалізувати. Далі заходимо у вкладку `message` та вписуємо те саме що і в HTTP запит:

```
{"name": "John"}
```

Наразі у нас є все для проведення запитів, у скрипті, який створює записи користувачів, змінній `n` надаємо значення 0 та запускаємо скрипт. Створений файл копіюємо у gRPC та REST API проекти. Далі проводимо запити через Postman, на кожен із серверів проводимо по 5 запитів для того, щоб отримати середній час відповіді, також це потрібно для того, щоб сервер на REST API вийшов на максимальну продуктивність, тому що гем, який запускає наш сервер переходить у режим економії енергії, якщо не здійснювати запити до нього протягом кількох секунд, через це перший запит завжди буде у кілька разів повільніший ніж усі наступні. Далі повторюємо ці кроки для різної кількості записів користувачів (1.000, 10.000, 50.000, 100.000 і 200.000).

4.8 Представлення отриманих результатів

ЗАЛЕЖНІСТЬ ШВИДКОСТІ ВИКОНАННЯ ЗАПИТУ ВІД КІЛЬКОСТІ ЗАПИСІВ



4.9 Обговорення переваг та недоліків gRPC, які випливають з проведених експериментів.

Результати дослідження швидкості виконання запитів доволі не очікувані, тому що здавалось REST API, який використовує повільніший JSON, має відповідати довше, але ні, навіть при збільшенні кількості даних REST API працює швидше. Такий результат може бути наслідком недопрацьованості гему для Ruby, навіть якщо поглянути на gitHub gRPC в прикладах папка `ruby` найменш розвинена та має найменшу кількість файлів для використання, тому можливо сам протокол не оптимізований на роботу з Ruby. У свою чергу REST API створена за допомогою rails давно використовується. Також rails пропонує зручний фреймворк для створення проєктів, тобто після створення проєкту, ми чітко знаємо, де знаходяться файли які відповідають за рутинг, роботу з базою даних, обробкою бізнес логіки і так далі. Це дозволяє розвивати великі проєкти, над якими будуть працювати багато людей і всі будуть розуміти основні правила збереження файлів та їх призначення. gRPC у свою чергу не пропонує нічого для структурування файлів проєкту, тому для створення масштабних проєктів потрібно буде створювати свої правила роботи зі структурою проєкту і відповідно навчати нових співробітників дотримуватись цих правил.

4.10 Можливості для подальшого вдосконалення

Опираючись на попередній розділ можна виділити 2 основних моменти. Перший це оптимізація швидкості роботи гему протоколу gRPC, на мові Ruby. Хоча різниця не критична, але враховуючи, що серіалізатор даних Protobuf в середньому всі операції проводить у 4 рази швидше ніж JSON, має бути можливість оптимізувати швидкість роботи гему хоча

б на рівень REST API. Другий це створити фреймворк для роботи з gRPC, так як в процесі написання просто серверу відчуваються проблеми з розумінням розміщення файлів, але через не великий розмір проекту ця проблема не заважає його написанню, так як можна просто всі файли розмістити в корні проекту. При масштабуванні проекту це стане серйозною проблемою.

Розділ 5

Висновок

З виконаної роботи можна зробити висновок, що протокол gRPC є сучасним рішенням для роботи з віддаленим викликом процедур і саме як протокол RPC він є хорошим вибором для архітектури сервера. Але якщо на стадії вибору архітектури сервера є можливість обрати REST API, тобто у сервера не буде задач, при яких потрібно працювати із поточним з'єднанням або важливою є саме RPC архітектура, то поки що для сервера на мові Ruby краще обирати REST API, так як це перевірений часом варіант побудови архітектури сервера. Для цього варіанту є багато додаткових рішень, які допоможуть вирішити поставлені задачі при розробці сервера. У свою чергу для gRPC основний і практично єдиний ресурс знань та рішень на мові Ruby це офіційна документація від розробників, яка на жаль не є вичерпною.

Список літератури

- [1] *gRPC Motivation and Design Principles* [Електронний ресурс]. – дата візиту 12.05.2023. – Режим доступу до ресурсу: <https://grpc.io/blog/principles/#motivation>
- [2] *Що таке gRPC і як він працює* [Електронний ресурс]. – дата візиту 20.05.2023. – Режим доступу до ресурсу: <https://highload.today/uk/shho-take-grpc-i-yak-vin-pratsyuye/>
- [3] *HTTP/2 and gRPC: The De Facto Standard for Microservices Communication* [Електронний ресурс]. – дата візиту 21.05.2023. – Режим доступу до ресурсу: <https://betterprogramming.pub/http-2-and-grpc-the-de-facto-for-microservices-communication-84a6bb2a6126>
- [4] *Ruby Gems* [Електронний ресурс]. – дата візиту 26.05.2023. – Режим доступу до ресурсу: <https://rubygems.org>
- [5] *GitHub gRPC* [Електронний ресурс]. – дата візиту 27.05.2023. – Режим доступу до ресурсу: <https://github.com/grpc/grpc/tree/master/examples/ruby>

Розділ 6

Додаток

Додаток 1.1

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}

  rpc SayHelloStreamReply (HelloRequest) returns (stream HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
  repeated User users = 2;
}

message User {
  string name = 1;
```

```
int32 age = 2;
string digest = 3;
}
```

Додаток 1.2

```
require 'google/protobuf'

Google::Protobuf::DescriptorPool.generated_pool.build do
  add_file("protos/helloworld.proto", :syntax => :proto3) do
    add_message "helloworld.HelloRequest" do
      optional :name, :string, 1
    end
    add_message "helloworld.HelloReply" do
      optional :message, :string, 1
      repeated :users, :message, 2, "helloworld.User"
    end
    add_message "helloworld.User" do
      optional :name, :string, 1
      optional :age, :int32, 2
      optional :digest, :string, 3
    end
  end
end

module Helloworld
  HelloRequest = ::Google::Protobuf::DescriptorPool.generated_pool.lookup(
    "helloworld.HelloRequest").msgclass
  HelloReply = ::Google::Protobuf::DescriptorPool.generated_pool.lookup(
    "helloworld.HelloReply").msgclass
  User = ::Google::Protobuf::DescriptorPool.generated_pool.lookup(
    "helloworld.User").msgclass
end
```

Додаток 1.3

```
this_dir = File.expand_path(File.dirname(__FILE__))
lib_dir = File.join(this_dir, 'lib')
$LOAD_PATH.unshift(lib_dir) unless $LOAD_PATH.include?(lib_dir)
```

```
require 'grpc'
require 'helloworld_services_pb'
require 'yaml'

class GreeterServer < Helloworld::Greeter::Service
  def say_hello(hello_req, _unused_call)
    users = YAML.load_file('db.yml')
    Helloworld::HelloReply.new(message: "Hello #{hello_req.name}", users:)
  end
end

def main
  s = GRPC::RpcServer.new
  s.add_http2_port('0.0.0.0:12345', :this_port_is_insecure)
  s.handle(GreeterServer)

  s.run_till_terminated_or_interrupted([1, 'int', 'SIGTERM'])
end

main
```