Author: ***Phạm Nhân***

gRPC - Google Remote Procedure Call
Trước bài này cần tìm hiểu tới RPC và REST

I. Why gRPC ?

1. Sự ra đời và lý do phát triển gRPC.

gRPC được phát triển bởi Google, vốn đã sử dụng một hạ tầng RPC nội bộ có tên là Stubby để kết nối hàng trăm microservices vận hành trong và giữa các trung tâm dữ liệu suốt hơn một thập kỷ. Tuy nhiên, sự phát triển không ngừng của hệ thống microservices, cùng với nhu cầu ngày càng cao về hiệu suất và khả năng mở rộng, đã khiến Stubby không còn đáp ứng đủ.
Vào tháng 3 năm 2015, Google quyết định xây dựng một phiên bản mới thay thế Stubby và chuyển nó thành mã nguồn mở, với tên gọi **gRPC**. Framework này được thiết kế nhằm giải quyết các thách thức trong việc giao tiếp giữa các microservices trên quy mô lớn. Hai lý do chính dẫn đến sự ra đời của gRPC là:

1. Cải thiện hiệu suất

- Hệ thống microservices của Google đòi hỏi một giải pháp RPC hiệu suất cao để xử lý khối lượng lớn yêu cầu giữa các dịch vụ, đặc biệt trong các trung tâm dữ liệu phân tán.
- Stubby không còn đáp ứng tốt các yêu cầu về tốc độ và hiệu quả, nhất là khi phải xử lý các tác vụ encode/decode dữ liệu trung gian liên tục, vốn tiêu tốn tài nguyên CPU.

2. Tương thích đa ngôn ngữ

- Môi trường phát triển tại Google sử dụng nhiều ngôn ngữ lập trình khác nhau như C++, Java, Go, Python, v.v.
- Một giải pháp RPC mới cần có khả năng tương thích với nhiều nền tảng và ngôn ngữ, để hỗ trợ sự đa dạng trong các nhóm phát triển.

Với gRPC, Google không chỉ giải quyết bài toán giao tiếp nội bộ mà còn mang đến một giải pháp hiệu quả, linh hoạt cho cộng đồng. Ngày nay, gRPC được sử dụng rộng rãi bên

ngoài Google để hỗ trợ các kiến trúc microservices, cũng như các ứng dụng trong "last mile" của điện toán, bao gồm di động, web và Internet vạn vật (IoT).

2. Tổng quan.

REST API và thách thức của kỷ nguyên Microservices

Trong thời kỳ REST API trở nên phổ biến và được sử dụng rộng rãi, việc giao tiếp giữa client và server đã được giải quyết khá tốt. Tuy nhiên, trong bối cảnh hệ thống Microservices phát triển mạnh mẽ, nhu cầu tăng hiệu suất và thông lượng giữa các service đã đặt ra thách thức mới.
Khi hệ thống chỉ có ít service hoặc tải không lớn, các giải pháp hiện tại như REST API có thể đáp ứng tốt. Nhưng khi phải xử lý hàng trăm service với tải lớn, như vượt mốc 100.000 người dùng đồng thời (CCU), các vấn đề bắt đầu xuất hiện.

Ví dụ:
Một request cần tổng hợp dữ liệu từ nhiều service khác nhau. Mỗi lần dữ liệu được truyền qua lại giữa các service, chúng phải được encode và decode liên tục (chẳng hạn qua JSON), gây quá tải cho CPU. Thay vì dành tài nguyên cho các tác vụ quan trọng hơn, CPU bị lãng phí vào việc xử lý encode/decode trung gian.
=> Đây chính là lý do gRPC ra đời: để tối ưu hóa giao tiếp giữa các service, giảm tải encode/decode dữ liệu và tăng tốc độ xử lý tổng thể.

3. RPC không phải REST API

Câu hỏi đặt ra là tại sao không phải "gAPI" hay "gREST", mà lại là "gRPC". Đơn giản vì framework này tập trung vào RPC (Remote Procedure Call – Thủ tục gọi từ xa). RPC là một khái niệm đã xuất hiện từ rất lâu, trước cả REST API, nhằm cho phép một hệ thống gọi các hàm được triển khai ở hệ thống khác.
Với REST API, client và server trao đổi dữ liệu thông qua các resource, trong đó các response trả về thường chứa một object đầy đủ. Trong khi đó, với RPC, client gửi yêu cầu thực hiện một tác vụ cụ thể (như gọi hàm), và server chỉ trả về kết quả của tác vụ đó, không kèm dữ liệu thừa.

Ví dụ:

- Với REST API: Khi muốn lấy thông tin của user có ID = 1, server sẽ trả về toàn bộ object user chứa ID = 1.
- Với RPC: Nếu muốn tính tổng thu nhập của user trong tháng này, server chỉ trả về một con số, không phải một object phức tạp.

Sự khác biệt lớn nhất là REST API tập trung vào việc quản lý resource, trong khi RPC tập trung vào việc thực hiện các tác vụ hoặc tính toán cụ thể.

Câu hỏi: Thế thì tại sao không phải sửa rest api lại chỉ trả về mỗi thông tin người dùng thôi?

4. Tại sao gRPC vượt trội?

1. gRPC sử dụng **Protocol Buffers (Protobuf)** làm định dạng tuần tự hóa dữ liệu thay vì các định dạng text-based như JSON hoặc XML.

- Kích thước dữ liệu nhỏ hơn (binary format), giảm băng thông truyền tải.
- Tốc độ xử lý encode/decode nhanh hơn rất nhiều.
- Dễ dàng định nghĩa và duy trì cấu trúc dữ liệu với .proto files.

2. Hỗ trợ Streaming hai chiều
   gRPC hỗ trợ **full-duplex streaming**
   - Client và server có thể gửi/nhận dữ liệu đồng thời mà không cần chờ bên kia hoàn thành.
   - Điều này rất hữu ích cho các ứng dụng cần giao tiếp thời gian thực như video call, chat, hoặc xử lý dữ liệu luồng lớn (big data).
3. Đa ngôn ngữ và đa nền tảng – Phần này sẽ được giải thích ở các phần sau
4. Tích hợp sẵn các tính năng cần thiết như: Load Balancing, Health Checking, Authentication
5. Hỗ trợ giao tiếp trong môi trường Microservices
6. Backward Compatibility (Khả năng tương thích ngược)
   Khi thay đổi cấu trúc dữ liệu hoặc giao diện, Protocol Buffers của gRPC đảm bảo tương thích ngược, giúp duy trì kết nối giữa các phiên bản service cũ và mới mà không gây gián đoạn.
7. **Sử dụng HTTP/2**
   gRPC dựa trên **HTTP/2**, mang lại các lợi thế
   Multiplexing: Cho phép gửi nhiều yêu cầu qua cùng một kết nối mà không cần tạo lại kết nối mới (giảm overhead).
   Header Compression: Giảm kích thước gói tin HTTP, tiết kiệm băng thông.
   Persistent Connections: Duy trì kết nối lâu dài giữa client và server, giảm độ trễ khi thực hiện nhiều request liên tục.
8. **v…v…**

Tóm lại, hai thứ khiến cho gRPC trở nên phổ biến :
→ Protocol Buffers là định dạng mã hóa nhị phân rất hiệu quả. Nó nhanh hơn nhiều so với JSON.
→ gRPC được xây dựng dựa trên HTTP/2 để cung cấp nền tảng hiệu suất cao ở quy mô lớn.
Với những ưu điểm vượt trội như trên làm cho **gRPC** trở thành giải pháp lý tưởng trong môi trường Microservices phức tạp.

II. How gRPC ?

1. Kiến trúc gRPC.

Các thành phần chính của kiến trúc **gRPC** :

1. **Service Definition**
2. **Protocol Buffer (Protobuf)**
3. **HTTP/2**
4. **Client Stub và Server Implementation**
5. **Channel**

1.1 Service Definition

Service Definition trong gRPC là phần định nghĩa các phương thức dịch vụ mà client có thể gọi từ xa và các message sẽ được sử dụng làm tham số cho những phương thức đó.

1.1.1. Service.

- Một dịch vụ (service) trong gRPC là tập hợp các phương thức mà client có thể gọi từ xa trên server.
- Mỗi phương thức được định nghĩa trong dịch vụ với tên phương thức, các tham số (input), và kiểu dữ liệu trả về (output).

1.1.2. RPC Methods.

- Các phương thức trong dịch vụ sẽ được định nghĩa theo kiểu RPC (Remote Procedure Call).
- Một phương thức có thể có các kiểu giao tiếp khác nhau:
  - **Unary RPC**: Một yêu cầu và một phản hồi.
  - **Server Streaming RPC**: Một yêu cầu và một luồng phản hồi.
  - **Client Streaming RPC**: Một luồng yêu cầu và một phản hồi.
  - **Bidirectional Streaming RPC**: Cả hai bên đều có thể gửi luồng yêu cầu và phản hồi.

1.1.3. Messages (Payload).

- Các tham số đầu vào và đầu ra của phương thức thường được định nghĩa dưới dạng các message trong Protobuf. Một message là một cấu trúc dữ liệu với các trường (fields), mỗi trường có tên và kiểu dữ liệu (string, int32, etc).
- Ví dụ về Message:

```
message MyRequest {
        string name = 1;
        int32 age = 2;
}
message MyResponse {
         string greeting = 1;
}
```

Ví dụ về Service Definition trong gRPC:

```
syntax = "proto3";
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloResponse);
    rpc SayGoodbye (GoodbyeRequest) returns (GoodbyeResponse);
}
message HelloRequest {
    string name = 1;
}
message HelloResponse {
    string message = 1;
}
message GoodbyeRequest {
    string name = 1;
}
message GoodbyeResponse {
    string message = 1;
}
```

Quá trình xây dựng:

B1. Viết file .proto: Định nghĩa dịch vụ và các message cần thiết.

B2. Biên dịch file .proto: Dùng công cụ protoc (Protocol Compiler) để biên dịch file .proto thành các mã nguồn trong ngôn ngữ bạn đang sử dụng (Java, Python, Go, C#, v.v.).

B3. Triển khai dịch vụ: Sau khi biên dịch, bạn triển khai các phương thức của dịch vụ trên server và tạo client stub để gọi các phương thức từ client.

=> Service Definition là bước quan trọng vì nó định hình cách thức giao tiếp giữa client và server trong gRPC.

1.2 Protocol Buffer (Protobuf)

1.2.1. Khái niệm

Protocol Buffer là công cụ để **định nghĩa cấu trúc dữ liệu** (messages) mà sẽ được sử dụng trong quá trình giao tiếp giữa client và server.

1.2.2. Các tính năng của Protobuf.

+ Định dạng dữ liệu nhẹ và nhanh:
    - Protobuf rất nhẹ và hiệu quả về kích thước so với các định dạng khác như XML hoặc JSON. Điều này giúp giảm bớt băng thông và tăng tốc độ truyền tải.
    - Dữ liệu trong Protobuf được lưu trữ dưới dạng nhị phân, giúp giảm dung lượng bộ nhớ và tiết kiệm băng thông.
+ Dễ dàng mở rộng và tương thích ngược:

- Có thể dễ dàng thêm các trường mới vào các message mà không làm hỏng khả năng tương thích với các phiên bản trước của hệ thống. Điều này giúp bảo trì các API trong thời gian dài mà không làm gián đoạn các dịch vụ hiện có.
- Protobuf cho phép thay đổi cấu trúc dữ liệu mà không cần làm thay đổi phần lớn các hệ thống đang sử dụng, miễn là bạn tuân thủ một số nguyên tắc nhất định.

+ Protobuf hỗ trợ nhiều ngôn ngữ lập trình khác nhau như C++, Java, Python, Go, Ruby, JavaScript, C#, PHP và nhiều ngôn ngữ khác => không phải lo lắng về sự khác biệt giữa các ngôn ngữ.

1.2.3 Các thành phần trong Protobuf.

1. **Message**

   Message là đơn vị dữ liệu cơ bản trong Protobuf. Định nghĩa các message trong file .proto, mỗi message sẽ có các trường (fields) với tên và kiểu dữ liệu.
   Mỗi trường được đánh số, và các số này dùng để xác định trường khi dữ liệu được mã hóa (serialization).
   VD:

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

- Person là một message, với ba trường: name (kiểu string), id (kiểu int32), và email (kiểu string).
- Mỗi trường được đánh số (1, 2, 3) để giúp Protobuf xác định chúng khi mã hóa và giải mã dữ liệu.

2. **Field Types**

   Protobuf hỗ trợ nhiều kiểu dữ liệu như string, int32, bool, float, double, enum, repeated (cho mảng), v.v.
   Cũng có thể định nghĩa các message phức tạp (message bên trong message).

3. **Enums**

   Có thể định nghĩa các giá trị liệt kê (enum) trong Protobuf để làm rõ ý nghĩa của các giá trị trong hệ thống.
   VD:

```
enum Status {
    PENDING = 0;
    ACTIVE = 1;
    INACTIVE = 2;
}
```

Enum giúp làm cho mã của dễ đọc hơn và tránh dùng các giá trị số không rõ ràng.

4. **Services**

Dịch vụ (service) trong Protobuf mô tả các phương thức sẽ gọi từ client và server. Các phương thức này nhận một hoặc nhiều tham số và trả về một hoặc nhiều kết quả.

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

Trong ví dụ này, **Greeter** là một dịch vụ với một phương thức **SayHello**, nhận **HelloRequest** và trả về **HelloResponse**.

Quá trình sử dụng Protobuf:

B1. Định nghĩa các message và dịch vụ trong file - bắt đầu bằng cách tạo một file .proto chứa các định nghĩa message và service.

B2. Biên dịch file .proto -- Dùng công cụ **protoc (Protocol Compiler)** để biên dịch file .proto thành mã nguồn trong ngôn ngữ đang sử dụng (ví dụ: Java, Python, C++, v.v.). Sau khi biên dịch, sẽ có các lớp hoặc cấu trúc dữ liệu để sử dụng trong mã nguồn.

B3. Mã hóa và giải mã (Serialization/Deserialization)

- Khi muốn gửi một đối tượng từ client đến server hoặc ngược lại, ta sẽ "mã hóa" (serialize) đối tượng thành dữ liệu nhị phân và gửi qua mạng.
- Ở phía nhận, ta sẽ "giải mã" (deserialize) dữ liệu đó thành đối tượng gốc mà bạn có thể làm việc với nó.

VD:

```
syntax = "proto3";

service Greeter {
   rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
   string name = 1;
}

message HelloResponse {
   string message = 1;
}

# Server: Server sẽ triển khai phương thức SayHello, xử lý yêu cầu và trả về phản hồi
# Client: Client sẽ sử dụng stub được tạo ra từ file .proto để gọi phương thức SayHello từ xa.
```

1.3 HTTP/2

Phần này sẽ được đề cập chi tiết hơn ở mục riêng về HTTP/2, ở đây sẽ khái quát 1 số đặc điểm chính của HTTP/2 trong gRPC.

1.3.1 Đặc điểm chính của HTTP/2 trong gRPC

1. **Long-lived connections**
   HTTP/2 cho phép duy trì kết nối lâu dài giữa client và server
   => Giảm latency (độ trễ) và cải thiện hiệu suất tổng thể.
2. **Multiplexing**
   HTTP/2 cho phép gửi nhận nhiều yêu cầu/giao dịch trên cùng một kết nối
   => Tối ưu hóa sử dụng kết nối và tăng hiệu suất.
3. **Header compression**
   HTTP/2 áp dụng kỹ thuật nén tiêu đề (HPACK) để giảm overhead truyền tải dữ liệu => Đặc biệt hữu ích cho giao tiếp trong môi trường mạng chậm
4. **Server push**
   Server có thể chủ động gửi tài nguyên đến client mà không cần yêu cầu
   => Cải thiện thời gian tải trang và trải nghiệm người dùng
5. **Bi-directional data transfer**
   HTTP/2 cho phép gửi nhận dữ liệu hai chiều đồng thời
   => Hữu ích cho các ứng dụng yêu cầu giao tiếp thực-time.

1.3.2 Cách thức hoạt động của HTTP/2 trong gRPC.

B1: Thiết lập kết nối

- Client và server thiết lập một kết nối HTTP/2, Kết nối này sẽ tồn tại trong suốt quá trình giao tiếp.

B2: Gửi yêu cầu

- Khi client muốn gọi một phương thức dịch vụ, nó gửi một yêu cầu HTTP/2 (Yêu cầu này chứa thông tin về phương thức dịch vụ và dữ liệu cần truyền)

B3: Xử lý phản hồi

- Server xử lý yêu cầu và trả về phản hồi qua HTTP/2
- Phản hồi có thể là một hoặc nhiều chuỗi dữ liệu tùy thuộc vào kiểu phương thức RPC

B4: Dùng chung kết nối

- HTTP/2 cho phép sử dụng chung kết nối cho nhiều giao dịch.

Điều này giúp tối ưu hóa sử dụng kết nối và giảm latency.

1.3.3 Lợi ích của việc sử dụng HTTP/2 trong gRPC

- Tăng hiệu suất giao tiếp đáng kể
- Cải thiện khả năng tương tác thực-time
- Giảm latency và cải thiện trải nghiệm người dùng
- Tối ưu hóa sử dụng băng thông mạng

1.4 Client Stub và Server Implementation

Client Stub và Server Implementation là hai thành phần quan trọng trong kiến trúc gRPC, đóng vai trò chủ yếu trong việc xử lý các cuộc gọi Remote Procedure Call (RPC) giữa client và server.

1.4.1. Client Stub (Mã giả client)

1.4.1.1 Khái niệm.

Client Stub là mã được tạo ra từ **Service Definition** (định nghĩa dịch vụ) mà client sử dụng để gọi các phương thức từ server. Nó giống như một "proxy" phía client, giúp client thực hiện cuộc gọi RPC mà không cần phải biết chi tiết về cách thức giao tiếp với server.

1.4.1.2 Chức năng chính của Client Stub.

- **Đóng vai trò là giao diện cho client**: Client Stub cung cấp các phương thức tương ứng với các phương thức mà server cung cấp trong Service Definition. Client chỉ cần gọi các phương thức trong stub mà không cần biết về chi tiết giao thức truyền tải hay cách thức xử lý trên server.
- **Quản lý kết nối và giao thức**: Client Stub sẽ tự động thiết lập kết nối đến server (thường qua HTTP/2 trong gRPC), gửi yêu cầu và nhận phản hồi, tất cả các bước này được ẩn đi đối với client.
- **Mã hóa và giải mã dữ liệu**: Client Stub thực hiện mã hóa dữ liệu trước khi gửi đến server và giải mã dữ liệu khi nhận về từ server. Điều này có thể bao gồm việc sử dụng **Protocol Buffers (Protobuf)** để mã hóa và giải mã các message.
- **Xử lý lỗi và phản hồi**: Client Stub cũng xử lý các lỗi (nếu có) và trả lại kết quả cho client sau khi nhận được từ server.

VD :

*//Giả sử bạn có một dịch vụ **Greeter** với phương thức **SayHello**. **Client Stub** có thể tạo một phương thức **SayHello()** mà client có thể gọi như một phương thức bình thường trong mã của mình. Sau khi **client** gọi **SayHello()**, client stub sẽ đảm nhận tất cả việc truyền tải yêu cầu đến server và nhận phản hồi từ server.*

```
// Client-side code example

GreeterGrpc.GreeterBlockingStub stub = GreeterGrpc.newBlockingStub(channel);
HelloResponse res =stub.sayHello(HelloRequest.newBuilder().setName("John").build());
```

Ở đây, ***GreeterGrpc.newBlockingStub(channel)*** tạo ra một **client stub** cho dịch vụ **Greeter**. Client chỉ cần gọi stub.sayHello() với các tham số yêu cầu, và stub sẽ lo tất cả các chi tiết giao tiếp với server.

1.4.2. Server Implementation (Cài đặt server)

1.4.2.1 Khái niệm

Server Implementation là mã được phát triển phía server để thực thi các phương thức RPC mà client yêu cầu. Server Implementation phải triển khai các phương thức mà bạn đã định nghĩa trong Service Definition và sau đó đăng ký với gRPC server để xử lý các cuộc gọi từ client.

1.4.2.2 Chức năng chính của Server Implementation.

- Cung cấp các phương thức thực thi: Server Implementation thực thi logic của các phương thức đã được định nghĩa trong Service Definition. Mỗi phương thức trong Service Definition cần phải có một triển khai cụ thể trong Server Implementation.

- Lắng nghe và xử lý yêu cầu: Server Implementation sẽ lắng nghe các yêu cầu đến từ client qua kết nối gRPC và thực thi các phương thức tương ứng. Sau khi thực thi xong, server sẽ trả về kết quả cho client.

- Quản lý kết nối: Server Implementation quản lý các kết nối đến từ client, bao gồm xử lý các yêu cầu đơn (Unary RPC), streaming server (Server Streaming), hoặc các loại RPC bidirectional streaming.

- Xử lý các lỗi và thông báo: Trong quá trình thực thi các phương thức, server sẽ xử lý các lỗi, thực thi logic ứng dụng và gửi phản hồi lại client. Server có thể trả về các mã lỗi nếu có sự cố xảy ra trong quá trình thực thi phương thức.

VD:

Nếu bạn có một dịch vụ **Greeter** với phương thức **SayHello**, **Server Implementation** sẽ cần triển khai phương thức này và đăng ký với **gRPC** server.

```
// Server-side code example
public class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
     public void sayHello(HelloRequest req, StreamObserver<HelloResponse> responseObserver){
            String message = "Hello, " + req.getName();
            HelloResponse response =HelloResponse.newBuilder().setMessage(message).build();
            responseObserver.onNext(response);
            responseObserver.onCompleted();
        }
     }

// Server setup
Server server = ServerBuilder.forPort(50051)
   .addService(new GreeterImpl())
   .build()
   .start();
```

Ở đây, **GreeterImpl** là lớp triển khai server cho dịch vụ **Greeter**, trong đó phương thức **sayHello**() thực hiện công việc tạo ra phản hồi cho client. Sau khi xử lý xong, server trả về kết quả cho client thông qua **StreamObserver**.

1.4.2.3 Cài đặt server trong gRPC bao gồm các bước chính

B1. Triển khai các phương thức RPC trong một lớp con của GreeterGrpc.GreeterImplBase (hoặc lớp tương tự).
B2. **Khởi tạo server** và **đăng ký dịch vụ** với gRPC server
B3. **Lắng nghe và xử lý yêu cầu** đến từ client.

1.4.3. Kết luận

Client Stub: Là mã phía client giúp client gọi các phương thức từ xa một cách dễ dàng. Client không cần phải lo lắng về cách thức truyền tải dữ liệu hoặc xử lý kết nối mạng. Client chỉ cần sử dụng stub như một đối tượng bình thường để gọi các phương thức RPC.
Server Implementation: Là mã phía server giúp server thực thi các phương thức mà client yêu cầu. Server Implementation xử lý tất cả các yêu cầu từ client, thực thi logic và trả về phản hồi cho client.

1.5 Channel

Channel là một thành phần rất quan trọng để thiết lập và duy trì kết nối giữa client và server. Nó cung cấp giao diện mà client sử dụng để gửi các yêu cầu đến server và nhận các phản hồi. Channel giúp client quản lý việc kết nối, chuyển dữ liệu và giao tiếp với server một cách hiệu quả.

Channel cung cấp kết nối đến máy chủ gRPC trên máy chủ và cổng được chỉ định. Nó được sử dụng khi tạo stub máy khách. Máy khách có thể chỉ định đối số kênh để sửa đổi hành vi mặc định của gRPC, chẳng hạn như bật hoặc tắt nén tin nhắn. Kênh có trạng thái, bao gồm connected và idle.
Cách gRPC xử lý việc đóng kênh phụ thuộc vào ngôn ngữ. Một số ngôn ngữ cũng cho phép truy vấn trạng thái kênh.

1.5.1. Khái niệm về Channel

- **Channel** là một đại diện của kết nối **client-to-server** trong gRPC. Đây là thành phần mà client sử dụng để gửi và nhận các cuộc gọi RPC.
- Channel giữ các thông tin về kết nối, chẳng hạn như địa chỉ server, chế độ bảo mật (TLS/SSL), và các cấu hình khác.
- Thực chất, **Channel** là một ống (pipe) truyền tải dữ liệu giữa client và server. Client sử dụng channel để gọi các phương thức được định nghĩa trong dịch vụ (Service Definition) mà server cung cấp.

1.5.2. Chức năng của Channel

Kết nối tới server: Channel sẽ kết nối client đến server, thường qua một giao thức như HTTP/2.

Quản lý kết nối: Channel tự động quản lý các kết nối HTTP/2. Nếu có sự cố xảy ra (ví dụ: server không khả dụng), gRPC có thể tự động thử lại kết nối hoặc tìm kiếm các server thay thế nếu được cấu hình (trong trường hợp có nhiều server).
Chuyển dữ liệu giữa client và server: Channel chuyển tải các yêu cầu từ client và trả lại các phản hồi từ server.
Hỗ trợ bảo mật: Channel có thể sử dụng TLS (Transport Layer Security) để mã hóa kết nối giữa client và server, đảm bảo tính bảo mật cho dữ liệu trong quá trình truyền tải.

1.5.3. Các loại Channel trong gRPC

- Blocking Channel: Đây là loại channel đồng bộ (blocking), trong đó các cuộc gọi RPC từ client sẽ chờ đợi và hoàn thành trước khi tiếp tục thực thi các lệnh khác. Client sẽ bị chặn cho đến khi nhận được phản hồi từ server.

- Non-blocking (Asynchronous) Channel: Đây là loại channel bất đồng bộ, trong đó các cuộc gọi RPC có thể trả lại ngay lập tức và không chặn client. Client có thể tiếp tục thực hiện các công việc khác trong khi chờ đợi kết quả trả về từ server.

1.5.4. Cách tạo và sử dụng Channel

B1: Tạo channel: Client sẽ tạo một channel tới server bằng cách cung cấp địa chỉ (IP hoặc tên miền) của server và cổng mà server đang lắng nghe. Khi channel được tạo, client có thể sử dụng channel để gọi các phương thức RPC đã được định nghĩa trong dịch vụ gRPC.

Địa chỉ và cổng: Channel yêu cầu địa chỉ IP hoặc tên miền của server, ví dụ ***localhost:50051***, nơi gRPC server của bạn đang lắng nghe các yêu cầu.

Ví dụ về tạo channel trong gRPC (Java):

```
// Tạo channel kết nối đến server tại địa chỉ localhost:50051
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 50051)
    .usePlaintext() // Chỉ sử dụng plaintext (không sử dụng TLS)
    .build();
```

B2: Kết nối tới server: Sau khi tạo channel, client có thể sử dụng channel để thực hiện các cuộc gọi RPC.

Ví dụ sử dụng channel với client stub:

```
GreeterGrpc.GreeterBlockingStub stub = GreeterGrpc.newBlockingStub(channel);
HelloResponse res = stub.sayHello(HelloRequest.newBuilder().setName("John").build());
```

1.5.5. Quản lý Channel trong gRPC

- Channel Reuse: Trong gRPC, channel là đối tượng nặng, và việc tạo nhiều channel có thể gây lãng phí tài nguyên. Vì vậy, tốt nhất là nên tái sử dụng channel cho nhiều cuộc gọi RPC thay vì tạo mới mỗi lần. Đây là lý do tại sao gRPC khuyến

nghị sử dụng một channel duy nhất cho nhiều cuộc gọi trong suốt vòng đời của client.

- **Shutdown Channel**: Sau khi client không còn sử dụng channel nữa, bạn cần phải **shutdown channel** để giải phóng tài nguyên và kết nối.

Ví dụ tắt channel (Java):

```
// Khi kết thúc, shutdown channel để giải phóng tài nguyên
channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
```

1.5.6. Các tính năng nâng cao của Channel

- **Load Balancing (Cân bằng tải)**: GRPC hỗ trợ cân bằng tải, cho phép client kết nối đến nhiều server và tự động phân phối các yêu cầu giữa chúng. Điều này giúp cải thiện khả năng mở rộng và độ tin cậy của hệ thống.
- **Retry and Backoff (Thử lại và lùi lại)**: Nếu server không phản hồi hoặc gặp sự cố, channel có thể tự động thử lại yêu cầu hoặc giảm tốc độ gửi yêu cầu (backoff) để tránh tải quá mức lên server.
- **TLS/SSL**: GRPC cho phép bảo mật dữ liệu bằng TLS (Transport Layer Security), bảo vệ các cuộc gọi RPC khỏi bị nghe lén hoặc giả mạo.

1.5.7. Thực tiễn sử dụng Channel

- **Kết nối được duy trì lâu dài:** Channel nên được tái sử dụng xuyên suốt vòng đời của ứng dụng, đặc biệt trong môi trường có lưu lượng yêu cầu lớn.
- **Quản lý tài nguyên hiệu quả**: Vì channel là đối tượng nặng, tốt nhất là không nên tạo một channel cho mỗi yêu cầu. Thay vào đó, ta nên tái sử dụng một channel cho tất cả các yêu cầu đến server.

1.5.8. Kết luận

- **Channel** trong gRPC là thành phần quan trọng giúp thiết lập và duy trì kết nối giữa client và server. Nó cung cấp giao diện cho client để gửi yêu cầu và nhận phản hồi, và có thể được cấu hình để sử dụng các tính năng nâng cao như bảo mật, cân bằng tải, và tự động thử lại.
- Việc hiểu cách sử dụng và quản lý channel giúp tối ưu hóa hiệu suất và tài nguyên trong các ứng dụng gRPC.

2. Phân loại gRPC.

Trong gRPC, các loại dịch vụ (services) được xác định bởi cách thức các phương thức RPC (Remote Procedure Call) được triển khai. gRPC hỗ trợ ba loại dịch vụ chính, bao gồm:

2.1. Unary RPC – Gọi đơn phương

Đặc điểm:

- Client gửi một yêu cầu duy nhất và nhận một phản hồi duy nhất từ server.
- Hoạt động tương tự như một cuộc gọi hàm bình thường.

Cách thức hoạt động:

- B1: Client gọi một phương thức trên Stub.
- B2: Stub của client mã hóa và gửi yêu cầu đến server.
- B3: Server xử lý yêu cầu và trả về phản hồi.
- B4: Stub của client giải mã và trả về kết quả cho client.

Ứng dụng:

Thích hợp cho các yêu cầu đơn giản mà client chỉ cần nhận một kết quả trả về sau khi gửi yêu cầu.

VD:

```
//Giả sử dịch vụ Greeter với phương thức SayHello, đây là một ví dụ về Unary RPC:

#proto
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

// Client gửi yêu cầu HelloRequest và nhận về phản hồi HelloResponse.

#java
GreeterGrpc.GreeterBlockingStub stub = GreeterGrpc.newBlockingStub(channel);
HelloResponse res = stub.sayHello(HelloRequest.newBuilder().setName("Alice").build());
```

2.2. Server Streaming RPC - Truyền từ server đến client

Đặc điểm:

- Client gửi một yêu cầu duy nhất và nhận một stream dữ liệu từ server
- Server có thể gửi nhiều phản hồi trước khi kết thúc cuộc gọi.

Cách thức hoạt động:

- B1: Client gửi yêu cầu đến server.
- B2: Server bắt đầu gửi một stream dữ liệu đến client.
- B3: Client đọc từ stream cho đến khi không còn dữ liệu nào.
- B4: Server hoàn thành sau khi gửi hết tất cả các dữ liệu.

Ứng dụng:

Thích hợp cho các ứng dụng yêu cầu server gửi một lượng lớn dữ liệu hoặc theo dõi một chuỗi sự kiện.

VD:

```
// Dịch vụ StockPrice cung cấp một luồng dữ liệu giá cổ phiếu theo thời gian thực:

#proto
service StockPrice {
  rpc StreamStockPrices (StockRequest) returns (stream StockResponse);
}

// Client gửi yêu cầu StockRequest và nhận về một loạt các StockResponse từ server
theo thời gian.

#java
StockPriceGrpc.StockPriceStub stub = StockPriceGrpc.newStub(channel);
stub.streamStockPrices(request, new StreamObserver<StockResponse>() {
  @Override
  public void onNext(StockResponse response) {
     System.out.println("Received price: " + response.getPrice());
  }
  @Override
  public void onError(Throwable t) {
     t.printStackTrace();
  }
  @Override
  public void onCompleted() {
     System.out.println("Stream completed");
  }
});
```

2.3. Client Streaming RPC - Truyền từ client đến server

Đặc điểm:

- Client gửi một stream dữ liệu và nhận một phản hồi duy nhất từ server.
- Client có thể gửi nhiều yêu cầu trước khi nhận được phản hồi.

Cách thức hoạt động:

- B1: Client bắt đầu gửi một stream dữ liệu đến server.
- B2: Server đọc toàn bộ stream từ client.
- B3: Server xử lý tất cả các dữ liệu và trả về một phản hồi duy nhất.
- B4: Client hoàn thành sau khi nhận được phản hồi.

Ứng dụng:

Thích hợp cho các ứng dụng mà client cần gửi một loạt dữ liệu cho server (ví dụ: upload nhiều tệp tin) và nhận về một phản hồi duy nhất.

VD:

```
// Dịch vụ UploadFiles cho phép client gửi nhiều tệp tin và nhận phản hồi khi hoàn thành:

#proto
service FileService {
  rpc UploadFiles (stream FileRequest) returns (UploadResponse);
}

// Client gửi một chuỗi các FileRequest và server trả về một phản hồi UploadResponse.

#java
FileServiceGrpc.FileServiceStub stub = FileServiceGrpc.newStub(channel);
StreamObserver<FileRequest> requestObserver = stub.uploadFiles(new
StreamObserver<UploadResponse>() {
  @Override
  public void onNext(UploadResponse response) {
     System.out.println("Upload completed: " + response.getStatus());
  }
  @Override
  public void onError(Throwable t) {
     t.printStackTrace();
  }
  @Override
  public void onCompleted() {
     System.out.println("File upload finished");
  }
});

// Gửi các tệp tin dưới dạng stream
requestObserver.onNext(FileRequest.newBuilder().setFileName("file1.txt").build());
requestObserver.onNext(FileRequest.newBuilder().setFileName("file2.txt").build());
requestObserver.onCompleted();
```

2.4. Bidirectional Streaming RPC - Truyền hai chiều

Đặc điểm:

- Cả client và server đều gửi/receive stream dữ liệu trong cùng một thời gian.
- Hai stream hoạt động độc lập, cho phép gửi nhận dữ liệu theo bất kỳ thứ tự nào.

Cách thức hoạt động:

- B1: Client gọi phương thức và server nhận thông tin về cuộc gọi
- B2: Server có thể bắt đầu gửi stream dữ liệu ngay lập tức hoặc chờ đợi cho đến khi client bắt đầu gửi stream.
- B3: Cả client và server đều đọc/gửi dữ liệu từ/to các stream riêng biệt.

- B4: Cuộc gọi kết thúc khi cả hai bên hoàn thành việc gửi nhận dữ liệu.

Ứng dụng:

Thích hợp cho các ứng dụng cần trao đổi dữ liệu liên tục và đồng thời, như ứng dụng chat, video call, hoặc các hệ thống xử lý sự kiện thời gian thực.

VD:

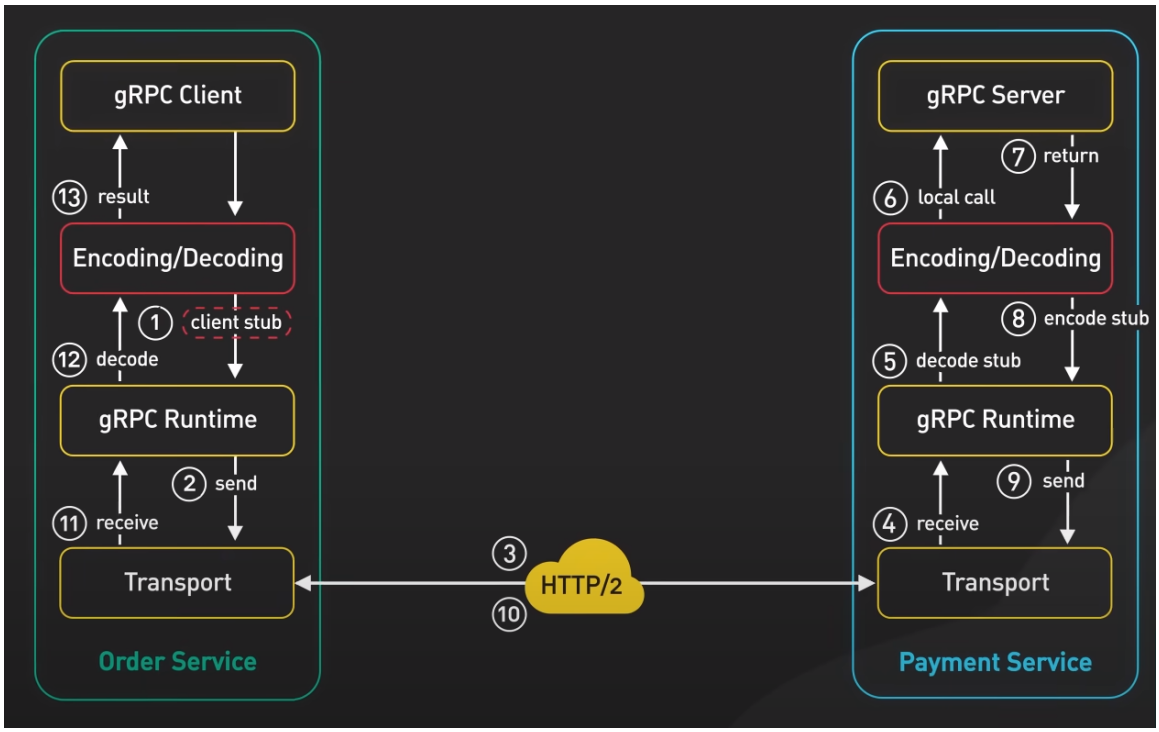
```
// Dịch vụ Chat cho phép client và server gửi và nhận tin nhắn đồng thời:

#proto
service Chat {
  rpc ChatStream (stream ChatMessage) returns (stream ChatMessage);
}

// Client và server gửi và nhận các ChatMessage theo dạng luồng (streaming).

#java
ChatGrpc.ChatStub stub = ChatGrpc.newStub(channel);
StreamObserver<ChatMessage> requestObserver = stub.chatStream(new
StreamObserver<ChatMessage>() {
  @Override
  public void onNext(ChatMessage message) {
     System.out.println("Received: " + message.getContent());
  }
  @Override
  public void onError(Throwable t) {
     t.printStackTrace();
  }
  @Override
  public void onCompleted() {
     System.out.println("Chat completed");
  }
});

// Client gửi tin nhắn tới server
requestObserver.onNext(ChatMessage.newBuilder().setContent("Hello,
server!").build());
requestObserver.onNext(ChatMessage.newBuilder().setContent("How are
you?").build());
requestObserver.onCompleted();
```

2.5. Kết luận

- **Unary RPC**: Client gửi một yêu cầu và nhận một phản hồi duy nhất từ server.

- **Server Streaming RPC:** Client gửi một yêu cầu và nhận một chuỗi phản hồi từ server.
- **Client Streaming RPC**: Client gửi một chuỗi yêu cầu và nhận một phản hồi duy nhất từ server.
- **Bidirectional Streaming RPC:** Cả client và server đều gửi và nhận dữ liệu theo dạng luồng, đồng thời và liên tục.

3. Luồng hoạt động.



B1. Client Stub (gRPC Client).

- Ứng dụng client gọi một phương thức RPC thông qua stub được tạo từ tệp .proto.
- Dữ liệu đầu vào của phương thức được truyền đến tầng Encoding/Decoding.

B2. Encoding/Decoding (Client).

- Dữ liệu yêu cầu (request) được mã hóa (serialize) thành định dạng nhị phân bằng Protocol Buffers (Protobuf).
- Dữ liệu sau đó được gửi xuống tầng gRPC Runtime**.**

B3. gRPC Runtime (Client).

- Tầng này chịu trách nhiệm quản lý phiên làm việc của gRPC, bao gồm xử lý kết nối, thời gian chờ (timeout), và tái kết nối nếu cần.
- Yêu cầu được chuyển đến tầng Transport**.**

B4. Transport (Client).

- Dữ liệu được gửi qua HTTP/2 đến server thông qua kết nối TCP.
- Đây là tầng chịu trách nhiệm truyền dữ liệu.

B5. Transport (Server).

- Server nhận dữ liệu từ client qua HTTP/2 và chuyển dữ liệu đến tầng gRPC Runtime trên server.

B6. gRPC Runtime (Server).

- Dữ liệu nhị phân được chuyển đến tầng Encoding/Decoding để giải mã.

B7. Encoding/Decoding (Server):

- Dữ liệu yêu cầu (request) được giải mã (deserialize) thành đối tượng hoặc cấu trúc dữ liệu để server có thể xử lý.
- Sau đó, yêu cầu được chuyển đến gRPC Server.

B8. gRPC Server.

- Phương thức RPC được gọi với dữ liệu đã giải mã.
- Server thực thi logic nghiệp vụ, xử lý yêu cầu, và trả về dữ liệu phản hồi.

B9. Encoding/Decoding (Server).

- Dữ liệu phản hồi (response) được mã hóa (serialize) thành định dạng nhị phân để chuẩn bị gửi lại client.

B10. gRPC Runtime (Server).

- Dữ liệu phản hồi được chuyển xuống tầng Transport trên server.

B11. Transport (Server).

- Dữ liệu được gửi qua HTTP/2 trở lại client.

B12. Transport (Client).

- Client nhận dữ liệu phản hồi từ server qua HTTP/2 và chuyển nó đến tầng gRPC Runtime.

B13. gRPC Runtime và Encoding/Decoding (Client).

- Dữ liệu phản hồi được giải mã (deserialize) thành đối tượng để client sử dụng.
- Kết quả được trả về ứng dụng client qua stub.

B14. Client Stub.

- Ứng dụng client nhận được kết quả và xử lý theo logic của nó.

III. Summary gRPC

1. Khái niệm

gRPC (Google Remote Procedure Call) là một framework mã nguồn mở do Google phát triển, được sử dụng để thực hiện các cuộc gọi thủ tục từ xa (Remote Procedure Call) giữa các ứng dụng, giúp các hệ thống giao tiếp với nhau dễ dàng và hiệu quả. gRPC hoạt động dựa trên HTTP/2 và sử dụng Protocol Buffers (protobuf) để định nghĩa các cấu trúc dữ liệu và giao thức giao tiếp.

2. Phân loại

Unary RPC - Giao tiếp một lần
Server Streaming RPC – Truyền từ server đến client
Client Streaming RPC – Truyền từ client đến server
Bidirectional Streaming RPC – Truyền hai chiều

3. Kiến trúc

Kiến trúc gRPC được xây dựng dựa trên giao thức Remote Procedure Call (RPC) với các công nghệ hiện đại như HTTP/2 và Protocol Buffers (protobuf). gRPC giúp client và server giao tiếp với nhau một cách hiệu quả thông qua các cuộc gọi thủ tục từ xa.

Service Definition
Protocol Buffer (Protobuf)
HTTP/2
Client Stub và Server Implementation
Channel

4. Ưu điểm

gRPC có một số ưu điểm nổi bật, bao gồm:

1. **Hiệu suất cao:** gRPC sử dụng giao thức HTTP/2, giúp tối ưu hóa băng thông, giảm độ trễ và hỗ trợ streaming. Điều này giúp gRPC trở thành lựa chọn lý tưởng cho các ứng dụng cần giao tiếp nhanh và có tần suất cao.
2. **Hỗ trợ đa ngôn ngữ**: gRPC hỗ trợ nhiều ngôn ngữ lập trình như Go, Java, C#, Python, Ruby, C++, và nhiều ngôn ngữ khác, giúp dễ dàng tích hợp với các hệ thống hiện có hoặc hệ thống đa ngôn ngữ.
3. **Kiến trúc API rõ ràng:** gRPC sử dụng Protobuf (Protocol Buffers) để định nghĩa các API. Điều này tạo ra một giao diện rõ ràng và dễ dàng đọc, hiểu, và duy trì.
4. **Streaming**: gRPC hỗ trợ giao tiếp stream (duplex, server-side, và client-side) giúp tối ưu hóa các ứng dụng cần trao đổi dữ liệu liên tục, ví dụ như truyền video hoặc cập nhật trạng thái thời gian thực.
5. **Xử lý lỗi mạnh mẽ:** gRPC cung cấp các cơ chế như retry và backoff để xử lý lỗi một cách hiệu quả, giúp hệ thống duy trì tính sẵn sàng và độ tin cậy cao.
6. **An toàn và bảo mật:** gRPC hỗ trợ mã hóa dữ liệu bằng SSL/TLS, bảo vệ dữ liệu trong quá trình truyền tải. Nó cũng cung cấp các cơ chế xác thực và ủy quyền mạnh mẽ.
7. **Dễ dàng tích hợp với hệ thống microservices:** gRPC rất phù hợp với các kiến trúc microservices, nơi mà các dịch vụ cần giao tiếp với nhau một cách nhanh chóng và đáng tin cậy.

8. **Tính mở rộng:** gRPC được thiết kế để có thể mở rộng linh hoạt, hỗ trợ nhiều máy chủ và client với các yêu cầu khối lượng công việc lớn mà không gặp vấn đề về hiệu suất.

5. Nhược điểm

1. **Cần HTTP/2:** gRPC yêu cầu sử dụng giao thức HTTP/2, điều này có thể gây khó khăn cho việc triển khai trong các hệ thống không hỗ trợ HTTP/2 hoặc yêu cầu hỗ trợ các giao thức HTTP/1.1 cũ hơn.

2. **Phức tạp trong cấu hình:** Việc thiết lập gRPC có thể phức tạp hơn so với các REST API thông thường, đặc biệt là khi triển khai trên các môi trường phức tạp hoặc trong hệ thống không quen thuộc với Protobuf.

3. **Hỗ trợ trình duyệt hạn chế:** gRPC không được hỗ trợ trực tiếp trên các trình duyệt web. Mặc dù có thể sử dụng gRPC-Web để giải quyết vấn đề này, nhưng đây là một giải pháp bổ sung, có thể làm phức tạp hóa việc triển khai và tích hợp.

4. **Dễ bị giới hạn khi cần tương thích với các hệ thống cũ:** gRPC yêu cầu sử dụng Protocol Buffers để định nghĩa API, điều này có thể không dễ dàng tích hợp vào các hệ thống đã sử dụng JSON hoặc các giao thức khác. Việc chuyển đổi các hệ thống cũ sang gRPC có thể tốn thời gian và công sức.

5. **Không phù hợp cho các ứng dụng đơn giản:** Đối với những ứng dụng nhỏ hoặc các API đơn giản, gRPC có thể quá phức tạp và không cần thiết. RESTful API vẫn là lựa chọn dễ dàng và phổ biến cho các ứng dụng không yêu cầu tính năng như streaming hoặc hiệu suất cực kỳ cao.

6. **Lượng tài nguyên tiêu thụ cao**: Protobuf có thể tạo ra các payload nhỏ gọn nhưng vẫn yêu cầu phần mềm xử lý bổ sung để mã hóa và giải mã, điều này có thể tiêu tốn tài nguyên hệ thống, đặc biệt khi có nhiều dịch vụ hoặc yêu cầu xử lý đồng thời.

7. **Khó khăn khi debug:** Việc debug các cuộc gọi gRPC có thể khó khăn hơn so với REST, vì gRPC thường không dễ dàng hiển thị thông tin chi tiết về lỗi, và cần công cụ đặc biệt để theo dõi lưu lượng và sự kiện.

8. **Tính tương thích ngược (Backward Compatibility):** Mặc dù Protocol Buffers cung cấp các cơ chế để duy trì tính tương thích ngược, nhưng khi thay đổi schema, việc quản lý các phiên bản khác nhau của API có thể trở nên phức tạp.