O que é RPC?
Remote Procedure Call (RPC) é um protocolo de comunicação de software que aplicações podem utilizar para requisitar um serviço de outra aplicação localizada em outro computador em uma rede sem ter que compreender os detalhes da rede. Uma *procedure call* também é chamada de *function call*.
Utiliza o modelo cliente-servidor. A aplicação requisitante é o cliente, e a aplicação fornecedora de serviço é o servidor. A RPC é uma operação síncrona, ou seja, a aplicação solicitante deve ser suspensa até que os resultados gerados pela RPC sejam retornados. Com a utilização de *threads* que compartilham do mesmo espaço de endereço permitem que multiplas RPCs possam ser executadas simultaneamente.
A Linguagem de Definição de Interface (IDL) – é uma linguagem de especificação utilizada para descrever uma *Application Programming Interface (API)* de um componente de software - é comumente utilizada em softwares RPC e fornece uma ponte entre o cliente e o servidor, que podem estar utilizando diferentes sistemas operacionais e linguagens de programação.

O que ele faz?
A proposta do gRPC é que o cliente interaja com o servidor por meio de chamadas de funções simples, ou seja, de interfaces de códigos geradas automaticamente pela própria aplicação do gRPC. Isso significa que você precisa apenas implementar sua lógica de programação, o que facilita muito a adoção desse recurso.

Com isso, você tem algumas vantagens na sua arquitetura de microsserviços, como, por exemplo:

- Fácil o contrato entre cliente e servidor;
- Melhor o desempenho dos serviços;
- Features nativas do HTTP/2, como streaming de dados, *load balance*, monitoramento etc.

Em resumo, é como se você declarasse funções e classes em um back-end e pudesse acessá-los no front-end graças ao arquivo de contrato que contém suas interfaces (serviços e DTOs).

Arquitetura
Arquiteturas RPC são muito parecidas. A ideia base é que temos sempre um servidor e um cliente, do lado do servidor temos uma camada que é chamada de **skeleton**, que é essencialmente um decriptador de uma chamada de rede para uma chamada de função, este é o responsável por chamar a função do lado do servidor.

Enquanto isso, do lado do cliente, temos uma chamada de rede feita por um **stub**, que é como um "falso" objeto representando o objeto do lado do servidor. Este objeto tem todos os métodos com suas assinaturas.

HTTP/2
Multiplexação de requisições e respostas
Tradicionalmente, o HTTP não pode enviar mais de uma requisição por vez para um servidor, ou então receber mais de uma resposta na mesma conexão, isso torna o HTTP/1.1 mais lento, já que ele precisa criar uma nova conexão para cada requisição. No HTTP/2 temos o que é chamado de multiplexação, que consiste em poder justamente receber várias respostas e enviar várias chamadas em uma mesma conexão. Isto só é possível por conta da criação de um novo frame no pacote HTTP chamado de **Binary Framing**. Este frame essencialmente separa as duas partes (headers e payload) da mensagem em dois frames separados, porém contidos na mesma mensagem dentro de um encoding específico.

Compressão de headers
Em alguns casos os headers de uma chamada HTTP podem ser maiores do que o seu payload, por isso o HTTP/2 tem uma técnica chamada HPack que faz um trabalho bastante interessante.

Inicialmente tudo na chamada é comprimido, inclusive os headers, isso ajuda na performance porque podemos trafegar os dados binários ao invés de texto. Além disso, o HTTP/2 mapeia os headers que vão e vem de cada lado da chamada, dessa forma é possível saber se os headers foram alterados ou se eles estão iguais aos da última chamada.

Se os headers foram alterados, somente os headers alterados são enviados, e os que não foram alterados recebem um índice para o valor anterior do header, evitando que headers sejam enviados repetidamente.

Protocol Buffers
Os protocol buffers (ou só **protobuf** para os intimos), são um método de serialização e desserialização de dados que funciona através de uma linguagem de definição de interfaces (IDL).
A grande vantagem do protobuf é que ele é agnóstico de plataforma, então você poderia escrever a especificação em uma linguagem neutra (o próprio `proto`) e compilar esse contrato para vários outros serviços, dessa forma a Google conseguiu

unificar o desenvolvimento de diversos microsserviços utilizando uma linguagem única de contratos entre seus serviços.
O protobuf em si não contém nenhuma funcionalidade, ele é apenas um descritivo de um serviço. O serviço no gRPC é um conjunto de métodos, pense nele como se fosse uma classe. Então podemos descrever cada serviços com seus parâmetros, entradas e saídas.

Cada método (ou RPC) de um serviço só pode receber um único parâmetro de entrada e um de saída, por isso é importante podermos compor as mensagens de forma que elas formem um único componente.

Além disso, toda mensagem serializada com o protobuf é enviada em formato binário, de forma que a sua velocidade de transmissão para seu receptor é muito mais alta do que o texto puro, já que o binário ocupa menos banda e, como o dado é comprimido pelo HTTP/2, o uso de CPU também é muito menor.

Vantagens

Bash

O compilador vai criar um arquivo `pessoa_pb.js` na pasta `dist` usando o modelo de importação CommonJS (isso é obrigatório se você for executar com Node.js), e ai podemos escrever um arquivo `index.js`:

```
const {Pessoa} = require('./pessoa_pb')

const p = new Pessoa()
p.setId(1)
p.setEmail('hello@lsantos.dev')

const serialized = p.serializeBinary()
console.log(serialized)

const deserialized = Pessoa.deserializeBinary(serialized)
console.table(deserialized.toObject())
console.log(deserialized)
```

JavaScript

Então vamos precisar instalar o protobuf com `npm install google-protobuf` e executamos o código:

```
Uint8Array(21) [
    8,   1,  18,  17, 104, 101,
```

```
  108, 108, 111,  64, 108, 115,
   97, 110, 116, 111, 115,  46,
  100, 101, 118
]
```

```
┌─────────┬──────────────────────┐
│ (index) │        Values        │
├─────────┼──────────────────────┤
│   id    │          1           │
│  email  │ 'hello@lsantos.dev'  │
└─────────┴──────────────────────┘
```

```
{
  wrappers_: null,
  messageId_: undefined,
  arrayIndexOffset_: -1,
  array: [ 1, 'hello@lsantos.dev' ],
  pivot_: 1.7976931348623157e+308,
  convertedPrimitiveFields_: {}
}
```

Output

Veja que temos um encoding igual ao que analizamos antes, uma tabela dos valores em objetos e a classe inteira.

Utilizar o protobuf como camada de contratos é muito útil, por exemplo, para padronizar as mensagens enviadas entre serviços de mensageria e entre microsserviços. Como estes serviços podem receber qualquer tipo de entrada, o protobuf acaba criando uma forma de garantir que todas as entradas sejam válidas.

Vantagens

1. Mais leve e mais rápido por utilizar codificação binária e HTTP/2
2. Multi plataforma com a mesma interface de contratos
3. Funciona em muitas plataformas com pouco ou nenhum overhead
4. O código é auto documentado
5. Implementação relativamente fácil depois do desenvolvimento inicial
6. Excelente para trabalhos entre times que não vão se encontrar, principalmente para definir contratos de projetos open source.

Desvantagens

1. O protobuf não possui um package manager para poder gerenciar as dependências entre arquivos de interface
2. Exige uma pequena mudança de paradigma em relação ao modelo ReST
3. Curva de aprendizado inicial é mais complexa
4. Não é uma especificação conhecida por muitos
5. Por conta de não ser muito conhecido, a documentação é esparsa
6. A arquitetura de um sistema usando gRPC pode se tornar um pouco mais complexa