gRPC

mardi 12 novembre 2024 13:12

Qu'est-ce que gRPC ? :

- Framework RPC (Appel Distant de Procédures)
- Moderne, universel et Open source (2015)
- Version open source de Stubby (google)
- Haute performance (faible latence, performant)
- Multiplateforme (mobile, navigateurs, etc.)
- Destiné à l'informatique massivement distribuée
- Permet de connecter des appareils, des applications mobiles et des navigateurs à différents services
- Orienté services (micro-services) vs. Objet
- Orienté messages (vs. références)
- Architecture client/serveur
- Création des systèmes distribués, connectés
- Projet de la CNCF (Cloud Native Computing Fondation, 2017)

Pourquoi gRPC ? :

- Définition de services simples
  - À l'aide de Protocole Buffers (IDL), un ensemble d'outils de langages de sérialisation binaire
- Démarrage rapidement et évolutif
  - Installation facile des environnements d'exécution et de développement avec une seule ligne
  - Adaptation à des millions d'appels distants par seconde avec le Framework
- Fonctionnement avec plusieurs langages et plateformes
  - Génération automatiquement des stubs client et serveur idiomatique pour des services dans une variété de langages et plateformes
- Streaming bidirectionnel et authentification intégrée
  - Authentification pluggable, entièrement intégrée avec transport basé sur le protocole HTTP/2

Caractéristiques :

- Concept de IDL (langage de définition/description d'interfaces de services/opérations)
- Protocole HTTP/2
- Protobuffer3 (Protocol Buffers 3)
- Langages supportés : C++, Objective-C, PHP, Python, Ruby, Node.js, Go, C#, Java
- Options de connexion : unaire, streaming côté serveur, streaming côté client, streaming bidirectionnel
- Communications asynchrones entre le client et le serveur par défaut (paramètres in et out)
- Communications synchrones possible (appels de procédures/fonctions)
- Options d'authentification : SSL/TSL, (Token based authentification)

Architecture gRPC :
Système distribué, 100% hétérogène, avec un architecture client/serveur

Pourquoi HTTP/2 :

- Protocole binaire(plus de texte)
- Support de stream nativement (pas besoin de webSocket)
- Stream multiplexing (connexion simple)
- Compression des entêtes (moins de bandes passantes)

Le langage protocol buffers :

- Indépendant des langages de programmation et des plateformes (OS)
- IDL pour gRPC
- Protobuf (.proto)
- Versions : Proto2 et Proto3 (ces deux versions ne sont pas compatibles : dans ce cours on utilise la version Proto3)
- Extensible
- Binaire (binary & compact)
- Fortement typé
- Versionné (contrôle de version)
- Facilement transformable en JSON

Définition d'un service gRPC :

Définition d'un service gRPC :

```
syntax = "proto3";

service Service {
        // Opérations rpc (méthodes rpc, 4types)
        rpc search(SearchRequest) returns (SearchResponse); // opération unaire
        rpc search(SearchRequest) returns (stream SearchResponse); // on peut avoir 1 ou plusieurs données en sortie
        rpc search(stream SearchRequest) returns (SearchResponse); // on peut avoir 1 ou plusieurs données en entrée
        rpc search(stream SearchRequest) returns (stream SearchResponse); // streaming bidirectionnel (on peut avoir 1
      ou plusieurs données en entrée et/ou en sortie
}
```

Il y a toujours une donnée en entrée et une donnée en sortie
Quand on a rien à envoyer on met void

Définition d'un type de message :

```
syntax ="proto3";

message  MessageRequest {
        // Fields
        string message =1; //champs // chaque champ a un identifiant 1,2,3…
        int32 i=2;
}

message MessageResponse {
        // Fields
        int64 x=1;
}

message MessageResponse {
        // Fields
        repeated int64 Result=1; // série de int64
}
```

1. On doit définir les services
2. Si les services utilisent des messages ont doit définir les messages

Définition d'une énumération :

```
syntax = "proto3";

enum Operation {
        // Values
        ADD = 0;
        MUL = 1;
        SUB = 2;
        DIV = 3;
}
// MessageDefintition.Operation.ADD;
```

Exemple :

```
syntax = "proto3";

message MessageDefinition {
        // Fields
        string message =1;
        enum Operation {
                // Values
```

```
        // Fields
        string message =1;
        enum Operation {
                // Values
                ADD = 0;
                MUL = 1;
                SUB = 2;
                DIV = 3;
        }
        Operation op = 2
}
```

Types de protobuf (.proto) : (types de bases du protocol buffer)

- double
- float
- int32
- int64
- unit32
- unit64
- sint32
- sint64
- fixed32
- fixed64
- sfixed32
- sfixed64
- bool
- string
- bytes

Importation de définitions :

```
syntax = "proto3";

import "nomProjet/unAutreProto.proto";

service Service {
        // Opérations
        rpc operation(Request) returns (Response);
}
```

Les options de définitions :

```
syntax = "proto3";

import "nomProjet/unAutreProto.proto";

option java_package = "sar.exemple.toto";

option java_multiple_files = true;

service Service {
        //Opérations
        rpc search(SearchRequest) returns (SearchResponse);
}
```

Génération des classes des types :

- Compilateur Protobuf : protoc (ne traine que les données => les messages pas les opérations)
- Un classe par message ou énumération est créée
  Ex: MessageRequest.java
  MessageDefinition.java

Operation.java
SearchRequest.java
etc.

- + des interfaces par messages :
  MessageRequest.OrBuilder.java
  Messagedefinition.OrBuilder.java

Génération des stubs et servant :

- ServiceGrpc.java
  - Classe de base pour l'implémentation des services : ServiceGrpc.ServiceImplBase (opérations définies dans service)
  - Classes des stubs client que le client peut utiliser pour échanger avec le serveur :
    - ServiceGrpc.ServiceStub; (ServiceStub fait des appels asynchrones)
    - ServiceGrpc.ServiceBlockingStub; (ServiceBlockingStub fait des appels synchrones)
    - ServiceGrpc.ServiceFutureStub;

Implémentation du servant :

- Servant = Objet de service
- Héritage de la classe interne :
  - NomServiceImplBase.java
- Implémentation des méthodes déclarées dans le service NomService

```
public class ServiceImpl extends ServiceGrpc.ServiceImplBase {
 public void operation(MessageRequest request, StreamObserver<MessageResponse> responseObserver ) {
  //traitements => rep;
  responseObserver.onNext(rep);
  responseObserver.onCompleted();
}
```

L'interface StreamObserver :

- Public interface StreamObserver<V>
- Reçoit des notifications à partir d'un flux de messages
- Utilisé par les stubs client et les implémentations de services pour envoyer ou recevoir des messages de flux
- Utilisé aussi pour les appels UNARY (Unaires)
- Fourni par la bibliothèqye GRPC pour les messages sortants
- Implémenté par l'application qui le transmet à la bibliothèque GRPC pour la réception (pour les messages entrants)
- Si plusieurs threads écrievnt simultanément sur un m^me StreamObserver, les apples doient être synchrones
- 3 méthodes :
  - Public void onNext(v value) // envoie de la réponse
  - Public void onCompleted() // pour dire qu'on a fini
  - Public void onError()

Réalisation du serveur :

1. Spécifiez l'adresse et le port que nous souhaitons utiliser pour écouter les demandes des clients à l'aide de la méthode du ServerBuilder.forPort(int) (Builder de Server)
2. Créez un instance de notre classe d'implémentation de service et transmettez-là à la méthode du ServerBuilder.addService(servant)
3. Appelez le constructeur pour créer et démarrer un serveur RPC pour notre service : ServerBuilder.build().start()

Tous les objets construit avec des builders ne peuvent pas être modifiés
Si on veut modifier, on doit créer un autre builder

Réalisation du client :

1. Création d'un canal gRPC pour le stub client, en spécifiant l'adresse du serveur et le port sur lequel nous souhaitons nous connecter :
   - ManagedChannelBuilder.forAdress(host, port).usePlaintext();
   - channel = channelBuilder.build();
2. Création du stub (synchrone ou asynchrone)
   - blockingStub = ServiceGrpc.newBlockingStub(channel);
   - asynchStub = ServiceGrpc.newStub(channel);
3. Utilisation (invocation/appel) du service

- response = blockingStub.operation(request);
- asynchStub.operation(request, StreamObserver<response>);

Le Design Pattern Builder dans java :

- Une autre façon de construire des objets complexes
- Construction d'objets immuables (exemple : String)
- Similaire / proche du pattern Abstract Factory
- Des getters, mais pas de setters (repassage par Builder)
- UserBuilder : classe statique interne
- Méthodes : public UserBuilder setAttribut(Type v);
- + Méthode : public User build();

  Avantages :
  - accès aux seuls attributs initialisés
  - Gestion efficace de la mémoire (gain d'espace)

Environnement de travail :

Exemple (interfaces) :
Définition des interface (IDL/ProtoBuf) : à voir dans le cours

Exemple (côté serveur) : à voir dans le cours
Implémentation du servant
Implémentation du serveur

Tester le serveur avec BloomRPC :

- BloomRPC : client générique avec interface graphique
- Multiplateforme (windows, mac os, linux)
- Lien

Exemple (implémentation du client) : à voir dans le cours
Implémentation du client
Stream côté client