gRPC - Remote Procedure Call

Jolan Philippe

21 août 2026

Web services, M1 MIAGE, 2026 - 2027

Table des matières

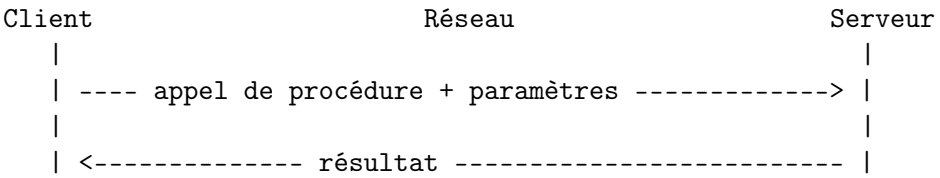
Rappel - API API (Application Programming Interface) : ensemble de règles, fonctions et points d'accès permettant à deux logiciels de communiquer. Une API définit essentiellement ce qu'on peut demander à un service et sous quelle forme.

Rappel - REST Les APIs REST (*Representational State Transfer*) donnent des points d'entrée à une façade distante, (historiquement) par dessus le protocole HTTP 1.1. Les points d'entrée : (i) un "verbe", ou méthode, HTTP (par ex. GET, PUT, POST, DELETE) ; (ii) une uri (ou chemin) ; (iii) des paramètres (optionnels) sous différentes formes (body, path, etc.). Un utilisateur peut alors écrire et envoyer une requête HTTP sur ce point d'entrée et récupère une réponse HTTP, contenant un code (voir https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP), et un corps.

1. Les principes généraux de RPC

1.1. Remote Procedure Call

On définit une API à l'aide d'un language dédié, appelé un **"Interface Definition Language"** (IDL). Dans cette API, on définit des fonctions (ou procédure) qui peuvent être appelée de manière distante. Comme dans un language de programmation, les fonctions ont un nom, des paramètres et retourne un résultat. Ici, les **paramètres sont typés**, ainsi que la valeur retournée. Le principe général peut être résumé ainsi.



En pratique, lors de l'utilisation d'une fonction de l'API, un **appel à distance** est fait par un client, sur une fonction / méthode / procédure fournie par un serveur. Le but est de **cacher la distance** à l'utilisateur L'API s'utilise ainsi comme une bibliothèque.

Pattern Proxy Le fonctionnement global est basé sur le design pattern **Proxy**. Il consiste à placer un objet intermédiaire entre le programme appelant, et l'objet distant que le client souhaite utiliser. Le proxy expose généralement la même interface que l'objet distant et se charge d'intercepter l'appel du programme appelant en un appel vers l'objet distant.

1.2. Proxy et "stub"

Dans RPC, il n'y a pas un, **deux objets faisant office de proxy**. Ils ont pour but de sérialiser / désérialiser (aussi appelé marshalling / unmarshalling) les entrées / sorties de l'API.

— **Coté client**, le **stub local** est en charge d'exposer les méthodes distantes. Lors d'un appel du client sur le stub, le stub sérialise les paramètres, créé et envoie une requête au serveur, puis intercepte la réponse et la désérialise.

— **Coté serveur**, le **stub**, ou **skeleton**, ou **handler**, intercepte la requête du client, la désérialise, et appel le code de traitement. Le retour de ce traitement est ensuite sérialisé, puis renvoyé au **stub client**.

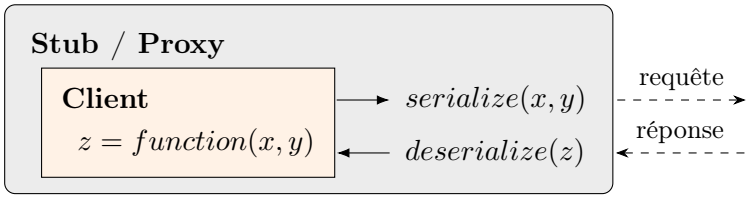


FIGURE 1 – Diagramme de fonctionnement du stub coté client.

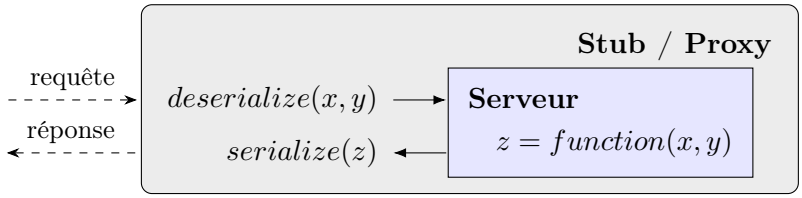


FIGURE 2 – Diagramme de fonctionnement du stub coté serveur.

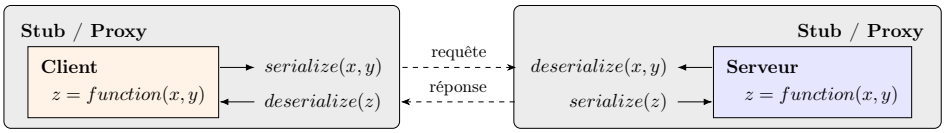


FIGURE 3 – Vue global du fonctionnement des stubs.

1.3. Interface Definition Language (IDL)

L'IDL a un rôle de documentation, et permet de générer automatiquement le code des stubs client et serveur. L'IDL utilisé dans ce cours pour gRPC s'appelle *Protocol Buffers* (Protobuf).

— Langage pour définir une **interface**, comme un **contrat**, entre un client et un serveur ; ici des services.
— Permet de déclarer des méthodes / procédures distantes en définissant leurs **signatures**.
— Permet de spécifier le **type des messages** d'entrée et de sortie des méthodes.

On peut voir la spécification écrite avec un IDL comme une documentation d'API (comme avec OpenAPI et Swagger) mais il a en plus le rôle de contrat à l'éxecution. En effet, les stubs sont générés à partir d'une spécification identique coté serveur et client.

1.4. HTTP/2

Un des éléments importants de gRPC est son utilisation de **HTTP/2**. Contrairement à HTTP/1.1, HTTP/2 permet notamment :

— le **multiplexage** de plusieurs échanges sur une même connexion ;
— la compression des headers ;
— une meilleure utilisation des connexions persistantes ;
⇒ la transmission efficace de flux de données.

2. Un peu d'histoire : des RPC à gRPC

gRPC s'inscrit dans une longue histoire de technologies destinées à faire communiquer des programmes distants. L'idée générale est ancienne : permettre à une application d'appeler une fonction située sur une autre machine presque comme s'il s'agissait d'une fonction locale.

2.1. Les débuts : RPC dans les années 1980

Avant l'apparition du Web et bien avant REST, les systèmes distribués utilisent déjà le concept de *Remote Procedure Call* (RPC).

Le concept de RPC est formalisé au début des années 1980. Parmi les implémentations importantes, on trouve notamment **Sun RPC**, développé par Sun Microsystems dans les années 1980 et utilisé, entre autres, par NFS (*Network File System*).

2.2. CORBA : faire communiquer des systèmes hétérogènes

Au début des années 1990 apparaît **CORBA** (*Common Object Request Broker Architecture*), standardisé par l'OMG (*Object Management Group*) à partir de 1991.

CORBA cherche à résoudre un problème particulièrement important à cette époque : faire communiquer des applications qui ne partagent pas nécessairement :

— le même système d'exploitation ;
— le même langage de programmation ;
— la même architecture matérielle ;
— le même environnement réseau.

L'élément central de CORBA est l'**ORB** (*Object Request Broker*), chargé d'assurer la communication entre les objets distribués. Les interfaces sont décrites pour la première fois grâce à **IDL** (*Interface Definition Language*).

CORBA définit également des protocoles permettant aux différents ORB de communiquer, notamment **IIOP** (*Internet Inter-ORB Protocol*), généralement utilisé au-dessus de TCP/IP.

Cependant, CORBA est relativement complexe à mettre en place et à administrer. Avec l'explosion du Web, des solutions utilisant directement HTTP vont progressivement prendre une place beaucoup plus importante.

2.3. XML-RPC : les RPC rencontrent le Web

À la fin des années 1990 apparaît **XML-RPC**.

L'idée consiste à reprendre le principe traditionnel des RPC, mais en utilisant les technologies du Web :

— HTTP pour le transport ;
— XML pour représenter les appels et leurs paramètres.

Un appel de fonction peut ainsi être représenté sous forme de document

XML et transmis à travers une infrastructure HTTP classique. XML-RPC est volontairement simple, mais ses possibilités restent limitées. Il sert notamment de base conceptuelle aux premiers travaux autour de SOAP.

2.4. SOAP : les Web Services structurés

En 1998, Microsoft et d'autres acteurs proposent les premières versions de **SOAP** (*Simple Object Access Protocol*). SOAP standardise l'échange de messages structurés en utilisant principalement **XML**. Même si SOAP peut théoriquement fonctionner avec différents protocoles de transport, il est très fréquemment utilisé avec **HTTP**.

Autour de SOAP se développe tout un ensemble de standards pour les **Web Services**. L'un des plus importants est **WSDL** (*Web Services Description Language*), apparu autour de 2000. WSDL joue un rôle proche d'un IDL : il décrit notamment

— les opérations disponibles ;
— les paramètres attendus ;
— les types utilisés ;
— les réponses possibles ;
— la manière d'accéder au service.

À partir d'un fichier WSDL, des outils peuvent générer automatiquement une partie du code client et serveur.

SOAP apporte donc un haut niveau de standardisation, mais au prix d'une certaine lourdeur : les messages XML sont relativement verbeux et leur traitement est par conséquent coûteux.

2.5. REST : une approche différente

Au début des années 2000, une autre approche devient progressivement populaire : REST (Roy Fielding).

REST (Roy Fielding) dans sa thèse publiée en 2000. Avec la généralisation de JSON, les API HTTP/REST deviennent extrêmement populaires. Cependant, pour les communications internes entre services, certaines limites peuvent devenir importantes :

— JSON est un format textuel relativement volumineux ;
— le typage des messages n'est pas toujours strict ;
— la génération automatique de clients dépend des outils utilisés ;
— le streaming bidirectionnel n'est pas naturel avec une API REST classique ;
— de nombreux petits appels entre microservices peuvent générer un coût important.

2.6. Protocol Buffers

Avant gRPC, Google développe un mécanisme de sérialisation de données : **Protocol Buffers**, souvent appelé **Protobuf**. L'approche de Protobuf reprend une idée présente dans CORBA et SOAP : **décrire formellement l'interface afin de générer automatiquement du code**.

2.7. gRPC

gRPC est un framework RPC moderne développé initialement par Google, puis publié en Open Source en 2015.

Il reprend le principe historique des RPC, mais l'adapte aux architectures modernes, notamment aux systèmes distribués et aux architectures de microservices.

gRPC repose principalement sur :

— **HTTP/2** pour le transport ;
— **Protocol Buffers** comme format de sérialisation par défaut ;
— une définition explicite des services et des messages ;
— la génération automatique de code client et serveur.

2.8. Une évolution plutôt qu'une révolution

Période	Technologie	Idée principale
Années 1980	RPC / Sun RPC	Appeler une procédure distante
1991	CORBA	RPC distribué, multi-langage et multi-plateforme
Fin 1990	XML-RPC	RPC utilisant HTTP et XML
1998–2000	SOAP / WSDL	Web Services fortement standardisés
2000+	REST	Manipulation de ressources via HTTP
2000+	Protocol Buffers	Sérialisation binaire et fortement typée
2015	gRPC	RPC moderne basé sur HTTP/2 et Protobuf

3. gRPC

3.1. Protocol Buffers

L'API est spécifiée dans un fichier par un IDL, ici **protobuf**, avec l'extension `.proto`.

— Spécification des signatures de procédures par service
— Structure des messages d'entrée et de sortie.

La documentation officielle : https://protobuf.dev/.

Listing 1 – Example student.proto

```proto
syntax = "proto3";

package example;

message Empty {}

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

message UserSubmission {
  string name = 1;
  string email = 2;
}

message UserRequest {
  int32 id = 1;
}

message UserId {
  int32 id = 1;
}

service UserService {
  rpc GetUser(UserRequest) returns (UserDTO);
  rpc GetUsers(Empty) returns (stream UserDTO);
  rpc AddUser(UserSubmission) returns (UserId);
}
```

Grammaire simplifiée de Proto3 :

```
<proto> ::= <syntax> <declaration>*

<syntax> ::= "syntax" "=" "\"proto3\"" ";"

<declaration> ::= <package>
                | <import>
                | <option>
                | <message>
                | <enum>
                | <service>

<message> ::= "message" ID "{"
                <message-element>*
              "}"

<message-element> ::= <field>
                    | <map>
                    | <oneof>
                    | <message>
                    | <enum>

<field> ::= ["optional" | "repeated"]
            <type> ID "=" INTEGER ";"

<map> ::= "map" "<" <key-type> "," <type> ">"
          ID "=" INTEGER ";"

<oneof> ::= "oneof" ID "{"
              <field>*
            "}"

<enum> ::= "enum" ID "{"
             (ID "=" INTEGER ";")*
           "}"

<service> ::= "service" ID "{"
                <rpc>*
              "}"

<rpc> ::= "rpc" ID
          "(" ["stream"] <message-type> ")"
          "returns"
          "(" ["stream"] <message-type> ")"
          ";"

<type> ::= <scalar-type>
         | <message-type>
         | <enum-type>
```

3.2. gRPC en Java

Une fois le fichier `.proto` écrit, le code Java peut être généré. La génération comprendra des classes correspondantes aux messages ; et des classes correspondantes au service. Si on prend l'exemple du Listing 1, génère

— un fichier `Student.java` contenant les classes `User`, `UserSubmission`, `UserRequest`, `UserId`. Chacune de ses classes possède des accesseurs correspondants aux champs des messages défini dans le fichier `.proto` ;

— un fichier `UserServiceGrpc.java`, contenant le squelettes des méthodes du service.

Attention : Ces fichiers ne doivent surtout pas être modifié manuellement.

3.3. Côté serveur

La génération du fichier `.proto` produit une classe abstraite servant de **squelette pour l'implémentation du serveur**. Pour notre service `UserService`, cette classe est :

```
UserServiceGrpc.UserServiceImplBase
```

Pour implémenter le service, il suffit donc de créer une classe Java héritant de `UserServiceImplBase`, puis de redéfinir les méthodes RPC souhaitées.

Avec Spring gRPC, l'annotation `@GrpcService` permet d'enregistrer automatiquement cette implémentation comme un service gRPC. C'est le même principe que `@RestController`.

```
@GrpcService
public class UserService
        extends UserServiceGrpc.UserServiceImplBase {
    ...
}
```

Chaque méthode RPC reçoit :

— le **message de requête**, généré à partir du fichier `.proto` ;

— un `StreamObserver` permettant d'envoyer la ou les réponses au client.

Par exemple, pour la procédure :

```
rpc GetUser(UserRequest) returns (UserDTO);
```

la méthode Java générée possède la forme suivante :

```
public void getUser(
    Student.UserDTORequest request,
    StreamObserver<Student.UserDTO> responseObserver) {
    ...
}
```

3.3.1 Construction des messages

Les messages Protobuf générés sont **immutables**. Ils ne sont donc pas construits directement avec un constructeur classique.

On utilise le **Builder** généré par Protocol Buffers. Le principe :

`newBuilder()` → `set...()` → `build()`

```
Student.UserDTO user = Student.UserDTO.newBuilder()
    .setId(1)
    .setName("Alice")
    .setEmail("alice@example.com")
    .build();
```

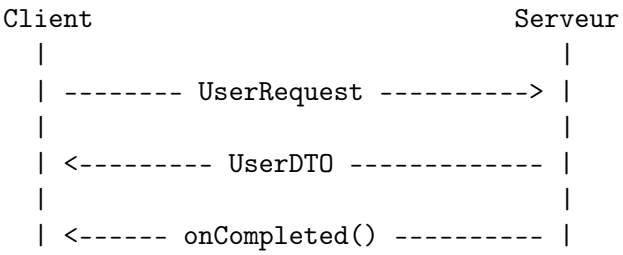
La réponse est envoyée grâce à un objet `StreamObserver<Response>`.

Trois méthodes sont particulièrement importantes :

- `onNext(message)` : envoie un message au client ;
- `onCompleted()` : indique que la communication s'est terminée normalement ;
- `onError(error)` : termine la communication avec une erreur.

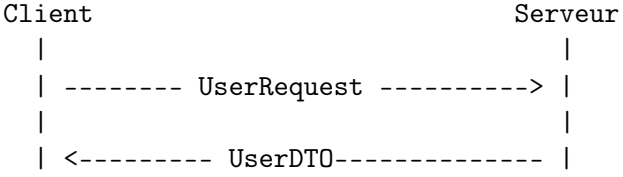
Dans le cas d'un appel RPC simple (*Unary RPC*), le serveur envoie généralement un seul message puis termine la communication :

```
responseObserver.onNext(response);
responseObserver.onCompleted();
```



Dans le cas d'un *streaming*, plusieurs appels à `onNext()` peuvent être effectués avant l'appel à `onCompleted()` :

```
responseObserver.onNext(response1);
responseObserver.onNext(response2);
responseObserver.onNext(response3);
responseObserver.onCompleted();
```



```
| <--------- UserDTO-------------- |
| <--------- UserDTO -------------- |
|                                   |
| <------ onCompleted() ---------- |
```

3.3.2 Exemple complet d'un serveur

```java
@GrpcService
public class UserService
        extends UserServiceGrpc.UserServiceImplBase {

    @Autowired
    UserFacade userFacade; // Model

    @Override
    public void getUser(
            Student.UserDTORequest request,
            StreamObserver<Student.UserDTO> responseObserver) {

        // 1 - Get user in the model
        User user = userFacade.getUser(request.getId());

        // 2 - Build the protobuf message
        Student.UserDTO.Builder builder =
            Student.UserDTO.newBuilder();

        builder.setId(user.getId());
        builder.setEmail(user.getEmail());
        builder.setName(user.getName());

        // 3 - Send the message
        responseObserver.onNext(builder.build());

        // 4 - Close the response
        responseObserver.onCompleted();
    }

    @Override
    public void getUsers(Student.Empty request,
                         StreamObserver<Student.UserDTO> responseObserver) {
        for (User user : userFacade.getUsers()){
            Student.UserDTO.Builder builder = Student.UserDTO.newBuilder();
            builder.setId(user.getId());
            builder.setEmail(user.getEmail());
            builder.setName(user.getName());
            responseObserver.onNext(builder.build());
        }
        responseObserver.onCompleted();
    }
}
```

Listing 2 – Exemple d'implémentation pour UserServiceGrpc

Il est également possible de retourner une erreur gRPC. Par exemple, si l'utilisateur demandé n'existe pas :

```java
responseObserver.onError(
```

```
    Status.NOT_FOUND
        .withDescription("User not found")
        .asRuntimeException()
);
```

gRPC définit plusieurs codes d'erreur standards, par exemple :

- `INVALID_ARGUMENT` ;
- `NOT_FOUND` ;
- `ALREADY_EXISTS` ;
- `PERMISSION_DENIED` ;
- `UNAUTHENTICATED` ;
- `INTERNAL` ;
- `UNAVAILABLE`.

3.3.3 Configuration du serveur

Le port utilisé par le serveur gRPC peut être configuré dans :

`src/main/resources/application.properties`

avec :

```
spring.grpc.server.port=9090
```

Le port par défaut d'un serveur gRPC Spring (utilisant Netty) est `9090`. Le serveur peut alors être contacté à l'adresse :

`localhost:9090`

Il est tout à fait possible de faire cohabiter une API REST et une API gRPC dans la même application Spring. Les deux interfaces peuvent alors interragir avec la même couche métier.

3.4. Côté client

Le code généré à partir du fichier `.proto` contient également le code nécessaire à la création d'un **client gRPC**.

```
service UserService {
    rpc GetUser(UserRequest) returns (UserDTO);
    rpc AddUser(UserSubmission) returns (UserId);
}
```

produit notamment un stub possédant des méthodes correspondant à `getUser()` et `addUser()`, utilisable par le client.

3.4.1 Le Channel

Le client doit créer un **Channel**. Un Channel représente la connexion entre le client et le serveur. Il est généralement conservé et réutilisé pour

plusieurs appels RPC. On definit un channel, en lui donnant un nom, pointant vers le serveur gRPC. On associe ce channel au stub généré, et ensuite, ce stub peut utiliser les méthodes distantes du serveur.

Avec GrpcChannelFactory Avec Spring gRPC, un channel peut être créé à l'aide de `GrpcChannelFactory`. Dans ce cas, la configuration suivante peut doit être ajoutée dans `application.properties` :

```
spring.grpc.client.channels.users.address=localhost:9090
```

Note : `GrpcChannelFactory` est un bean Spring, il n'est pas nécessaire de l'instancier.

```java
@Configuration
public class GrpcClientConfig {

    @Bean
    public UserServiceGrpc.UserServiceBlockingStub userStub(
            GrpcChannelFactory channelFactory) {
        ManagedChannel channel =
                channelFactory.createChannel("users");
        return UserServiceGrpc.newBlockingStub(channel);
    }
}
```

Avec ManagedChannelBuilder Autrement, on peut construire et configurer le channel programmatiquement.

```java
ManagedChannel channel = ManagedChannelBuilder
                            .forAddress("localhost", 9090)
                            .usePlaintext()
                            .build();
UserServiceGrpc.UserServiceBlockingStub stub = example.UserServiceGrpc.
     newBlockingStub(channel);
```

3.4.2 Appel d'une procédure distante

Pour appeler la procédure :

```
rpc GetUser(UserRequest) returns (UserDTO);
```

le client commence par construire le message de requête, puis utilise son stub pour emettre la requête avec un simple appel de fonction.

```java
Student.UserDTORequest request =
    Student.UserDTORequest.newBuilder()
        .setId(42)
        .build();
Student.UserDTO user = userStub.getUser(request);
```

Derrière cet appel se cachent plusieurs opérations :

1. sérialisation de `UserRequest` avec Protocol Buffers ;
2. envoi du message au serveur via HTTP/2 ;
3. exécution de `getUser()` sur le serveur ;
4. sérialisation de la réponse `User` ;
5. transmission de la réponse via HTTP/2 ;
6. désérialisation de la réponse côté client.

Le stub masque donc une grande partie de la complexité réseau :

3.4.3 Exemple complet d'un client

```java
@Service
public class SimpleClient {

    private final example.UserServiceGrpc.UserServiceBlockingStub stub;

    public SimpleClient(){
        ManagedChannel channel = ManagedChannelBuilder
                .forAddress("localhost", 9090)
                .usePlaintext()
                .build();
        stub = example.UserServiceGrpc.newBlockingStub(channel);
    }

    public void getUser(int id) {
        Student.UserDTORequest request =
                Student.UserDTORequest.newBuilder()
                        .setId(id).build();
        Student.UserDTO message_user = this.stub.getUser(request);
        User model_user = new User(message_user.getId(), message_user.getName
     (), message_user.getEmail());
    }

    public void getUsers() {
        Student.Empty request =
                Student.Empty.newBuilder().build();
        Iterator<Student.UserDTODTO> message_users = this.stub.getUsers(
     request);
        while (message_users.hasNext()){
            Student.UserDTODTO message_user = message_users.next();
            User model_user = new User(message_user.getId(), message_user.
     getName(), message_user.getEmail());
        }
    }
}
```

Listing 3 – Exemple d'utilisation d'un stub gRPC