Systèmes et Applications Distribués

Introduction à gRPC

Bachir Djafri
bachir.djafri@univ-evry.fr
Lab. IBISC/Dépt. d'Informatique
Université d'Évry – Paris-Saclay

Plan du cours

- Introduction
  - définitions
  - problématiques
  - architectures de distribution
- Distribution intra-applications
  - notion de processus
  - programmation multi-thread
- **Distribution inter-applications et inter-machines**
  - sockets
  - middlewares par appel de procédures distantes (RPC)
  - middlewares par objets distribués (Java RMI, CORBA, gRPC)
- Conclusion

Qu'est-ce que gRPC ?

- Framework RPC (Appel Distant de Procedures)
- 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

Qu'est-ce que gRPC ?

- Permet de connecter des appareils, des applications mobiles et des navigateurs à différents services
- Orienté services (micro-services) vs. Objets
- Orienté Messages (vs. Références)
- Architecture Client/Serveur
- Création de systèmes distribués, connectés
- Projet de la CNCF (Cloud Native Computing Fondation, 2017)

gRPC, pourquoi ?

- Définition de services simples
  - à l'aide de Protocol Buffers (IDL), un ensemble d'outils et 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 idiomatiques pour des services dans une variété de langages et de plateformes
- Streaming bidirectionnel et authentification intégrée
  - authentification *pluggable*, entièrement intégrée avec transport basé sur le protocole **HTTP/2**

Caractéristiques de gRPC

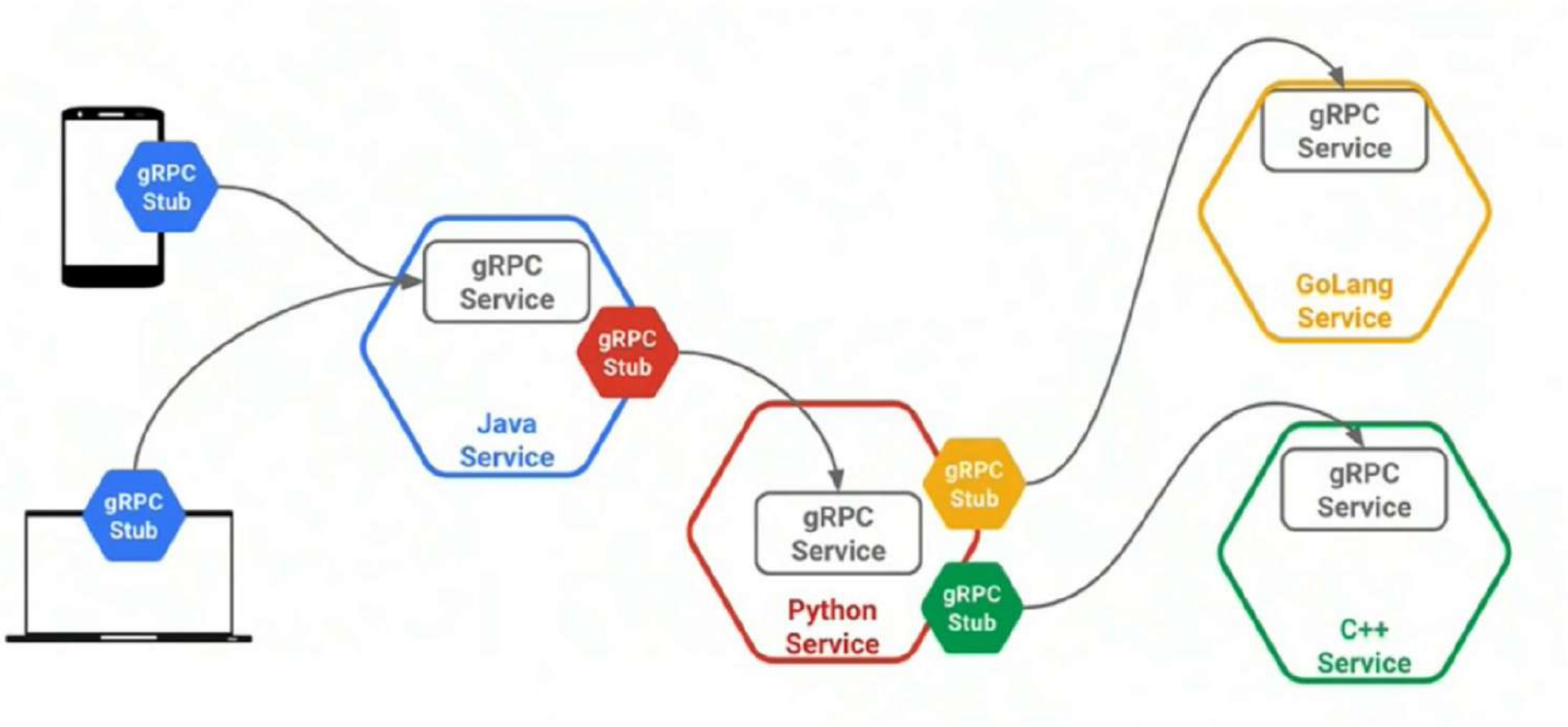
- Concept de IDL (langage de definition/description d'interfaces de services/opérations)
- Protocole HTTP/2
- Protobuffer3 (Protocol Buffers 3)
- Langages supportés : C++, *Objective-C*, PHP, Python, Ruby, Node.js, Go, *C#, Java*

Caractéristiques de gRPC

- *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/TLS, (Token based authentification)*

Architecture gRPC

Système distribué, 100% hétérogène, avec une architecture Client/Serveur



Pourquoi HTTP/2

- Protocole binaire (plus de texte)
- Support du Stream nativement (pas besoin de WebSocket)
- Stream Multiplexing (connexion simple)
- Compression des entêtes (moins de bande passante)
- http://www.http2demo.io

Le langage Protocol Buffers

- Indépendant des langages de programmation et des plateformes (OS)
- IDL pour gRPC
- Protobuf (.proto)
- Versions : Proto2 et Proto3

Le langage Protocol Buffers

- Extensible
- Binaire (Binary & compact)
- Fortement typé
- Versionné (contrôle de version)
- Facilement transformable en JSON

Définition d'un service gRPC

```
syntax = "proto3";

service Service {

        // Opérations rpc (méthodes rpc, 4 types)

        rpc search(SearchRequest) returns (SearchResponse);

        rpc search(SearchRequest) returns (stream SearchResponse);

        rpc search(stream SearchRequest) returns (SearchResponse);

        rpc search(stream SearchRequest) returns (stream SearchResponse);
}
```

Définition d'un type de message

```
syntax = "proto3";

message MessageRequest {
        // Fields
        string   message = 1;
        int32    i = 2;
}

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

// ceci est un commentaire
```

Définition d'un type de message

```
syntax = "proto3";

message SearchRequest {
        // Fields
        repeated int32 Result = 1;
}

message Result {
        // Fields
        repeated string          url = 1;
        string   message = 2;
}
```

Définition d'une énumération

```
syntax = "proto3";

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

/* ceci est un autre commentaire */
```

Définition d'une énumération

```
syntax = "proto3";

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

Types de protobuf (.proto)

.proto Type	Notes	Java Type	C# Type
double		double	double
float		float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	long	long
uint32	Uses variable-length encoding.	int[2]	uint
uint64	Uses variable-length encoding.	long[2]	ulong
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	long	long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	int[2]	uint
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	long[2]	ulong
sfixed32	Always four bytes.	int	int
sfixed64	Always eight bytes.	long	long
bool		boolean	bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	String	string
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	ByteString	ByteString

Importation de définitions

```
syntax = "proto3";

import "monProjet/unAutreProto.proto" ;

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

Les options de définitions

```
syntax = "proto3";

import "monProjet/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**
- **Une classe par message ou énumération est créée**

  MessageRequest.java
  MessageDefinition.java
  Operation.java
  SearchRequest.java
  etc.

- **+ des interfaces par message**
  - MessageRequestOrBuilder.java
  - SearchRequestOrBuilder.java
  - MessageDefinitionOrBuilder.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** ;
    - **ServiceGrpc.ServiceBlockingStub** ;
    - **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 bibliothèque 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 écrivent simultanément sur un même StreamObserver, les appels doivent être synchronisés
- 3 méthodes
  - public void onNext(V value) ;
  - public void onCompleted();
  - 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 une instance de notre classe d'implémentation de service et transmettez-la à la méthode du ServerBuilder.addService(servant)

3. Appelez le constructeur pour créer et démarrer un serveur RPC pour notre service : ServerBuilder.build().start()

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.forAddress(host, port).usePlaintext());
  - channel = channelBuilder.build();

- 2. Création du stub (synchrone ou asynchrone)

```
blockingStub = ServiceGrpc.newBlockingStub(channel);
asyncStub    = ServiceGrpc.newStub(channel);
```

- 3. Utilisation (invocation/appel) du service
  - Response = blockingStub.operation(request);
  - asyncStub.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();

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

Environnement de travail

Outils :

- **IDE + Maven ou Gradle (officiel)**

Configuration sur : https://github.com/grpc/grpc-java

- **compilateur pour les messages : protoc**
- **plugin java pour les services : protoc-gen-grpc-java**

Ligne de commande :

```
$protoc demo.proto --java_out=proto --plugin=proto-
gen-grpc-java
$protoc --plugin=protoc-gen-grpc-java --grpc-
java_out="$OUTPUT_FILE" --proto_path=
"$DIR_OF_PROTO_FILE" "$PROTO_FILE"
```

Exemple (Interfaces)

Définition des Interfaces (IDL/ProtoBuf)

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "Service";

service Service{
        rpc add(MessageRequest) returns (MessageResponse);
        rpc sub(MessageRequest) returns (MessageResponse);
        rpc mul(MessageRequest) returns (stream MessageResponse);
        rpc div(MessageRequest) returns (MessageResponse);
}

message MessageRequest{
        int32 x = 1;
        int32 y = 2;
}

message MessageResponse{
        int32 resultat = 1;
}
```

Exemple (côté Serveur)

Implémentation du servant

```
public class ServiceImpl extends ServiceImplBase{

   @Override
   public void add(MessageRequest request,
                   StreamObserver<MessageResponse> responseObserver) {

      int somme = request.getX() + request.getY();

      MessageResponse rep = MessageResponse.newBuilder()
                                           .setResultat(somme)
                                           .build();

      responseObserver.onNext(rep);

      responseObserver.onCompleted();

      // + implémentation des autres méthodes
   }
}
```

Exemple (stream côté Serveur)

```
public class ServiceImpl extends ServiceImplBase{
    @Override
    public void add(MessageRequest request, StreamObserver<MessageResponse> responseObserver) {

        int somme = request.getX() + request.getY();
        MessageResponse rep = MessageResponse.newBuilder().setResultat(somme).build();
        responseObserver.onNext(rep);
        responseObserver.onCompleted();
    }
    @Override
    public void div(MessageRequest request, StreamObserver<MessageResponse> responseObserver) {
    @Override
    public void sub(MessageRequest request, StreamObserver<MessageResponse> responseObserver) {
    @Override
    public void mul(MessageRequest request, StreamObserver<MessageResponse> responseObserver) {
        int s=1;
        for(int i=1; i<=request.getY(); i++){
            s = request.getX() * i;
            System.out.print("mul: s = "+s);
            MessageResponse rep = MessageResponse.newBuilder().setResultat(s).build();
            responseObserver.onNext(rep);
        }
        responseObserver.onCompleted();
    }
}
```

Exemple (côté Serveur)

Implémentation du serveur

```
public class DemoServer {

   public static void main(String[] args) throws IOException, InterruptedException{

      Server server = ServerBuilder.forPort(4444).addService( new ServiceImpl() ).build();

      server.start();

      System.out.println("Demo Server started on "+ server.getPort());

      server.awaitTermination();

   }
}
```

Tester le serveur avec BloomRPC

- **BloomRPC : client générique avec interface graphique**
- **Multiplateforme (Windows, Mac OS, Linux)**
- **Lien :** https://github.com/uw-labs/bloomrpc/releases

Exemple (implémentation du Client)

Implémentation du client

```
public class DemoClient {
   public static void main(String[] args) throws InterruptedException{

      ManagedChannel channel = ManagedChannelBuilder
                                          .forAddress("localhost", 4444).usePlaintext().build();

      // stubs generated from proto
      ServiceBlockingStub syncStub = ServiceGrpc.newBlockingStub(channel);

      MessageRequest request = MessageRequest.newBuilder().setX(15).setY(13).build();

      MessageResponse reponse = syncStub.add(request);

      System.out.print("add result = " + reponse);

      Iterator<MessageResponse> reponses = syncStub.mul(request);

      while(reponses.hasNext()){ System.out.print("mul stream : "+reponses.next());    }
}
```

Exemple (implémentation du Client)

```
public class DemoClient {
    public static void main(String[] args) throws InterruptedException{
        ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 4444)
                                        .usePlaintext().build();

        MessageRequest request = MessageRequest.newBuilder().setX(15).setY(13).build();

        ServiceStub asyncStub = ServiceGrpc.newStub(channel);

        StreamObserver<MessageResponse> responseObserver = new StreamObserver<>(){

            public void onNext(MessageResponse v) {
                System.out.println("onNext : v.resultat = "+v.getResultat());
            }

            public void onError(Throwable thrwbl) {
                Status status = Status.fromThrowable(thrwbl);
                System.err.println("error from throwable GRPC Server.");
            }

            public void onCompleted() {
                System.out.println("onCompleted : Finished mult");
            }
        };

        // appels asynchrones, sans et avec stream (côté serveur)
        asyncStub.add(request, responseObserver);
        asyncStub.mul(request, responseObserver);
 }
```

Exemple 2 (stream côté Client)

Définition des Interfaces (IDL/ProtoBuf)

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "Service";

service Service{

        rpc moyenne(stream MessageRequest) returns (MessageResponse);
}

message MessageRequest{
        int32 x = 1;
}

message MessageResponse{
        int32 resultat = 1;
}
```

Exemple 2 (stream côté Client)

```java
@Override
public StreamObserver<MessageUnaireRequest> moyenne(StreamObserver<MessageResponse> responseObserver){
    return new StreamObserver<MessageUnaireRequest>(){
        private int somme = 0;
        private int nbInt = 0;

        @Override
        public void onNext(MessageUnaireRequest value) {
            somme+=value.getX();
            nbInt++;
        }

        @Override
        public void onError(Throwable t) {
            Status status = Status.fromThrowable(t);
            System.err.println("onError from throwable GRPC Server. somme = "+somme+", nbInt = "+nbInt);
        }

        @Override
        public void onCompleted() {
            responseObserver.onNext( v:MessageResponse.newBuilder().setResultat(somme/nbInt).build());
            responseObserver.onCompleted();
            System.out.println("moyenne envoyée : " + somme/nbInt);
        }
    };
}
```