# Protocol Buffers et gRPC

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA



SRCS – Master 1 SAR 2019/2020

sources :

https ://developers.google.com/protocol-buffers

https ://grpc.io/

# Introduction

## Protocol Buffers : qu'est ce que c'est ?

Un utilitaire de sérialisation binaire développé par Google et qui offre :

- une API implantée dans la plupart des langages
- un DSL
- un compilateur vers un langage cible

**Permet de définir des messages**

## gRPC : qu'est ce que c'est ?

Un utilitaire développé par Google pour définir des services distants :

- Extension de protobuf pour le rpc
- Basé sur HTTP/2 pour le transport de données

**Permet de définir des services (interface + implantation)**

# Protobuf vs. autres systèmes de sérialisation

## Protobuf vs. XML

Protobuf est

- plus compact (sérialisation binaire + optimisation d'encodage)
- plus performant
- moins verbeux

## Protobuf vs. Sérialisation Java

Java n'est pas interopérable avec C++ ou python.

## Protobuf vs. Sérialisation "*maison*"

Plus flexible mais nécessite de programmer l'encodage et le décodage.

# Les utilisateurs

## Utilisateurs de Protobuf

- Outils Google
- Twitter
- Apache Mesos (gestionnaire de conteneurs)

## Utilisateurs de gRPC

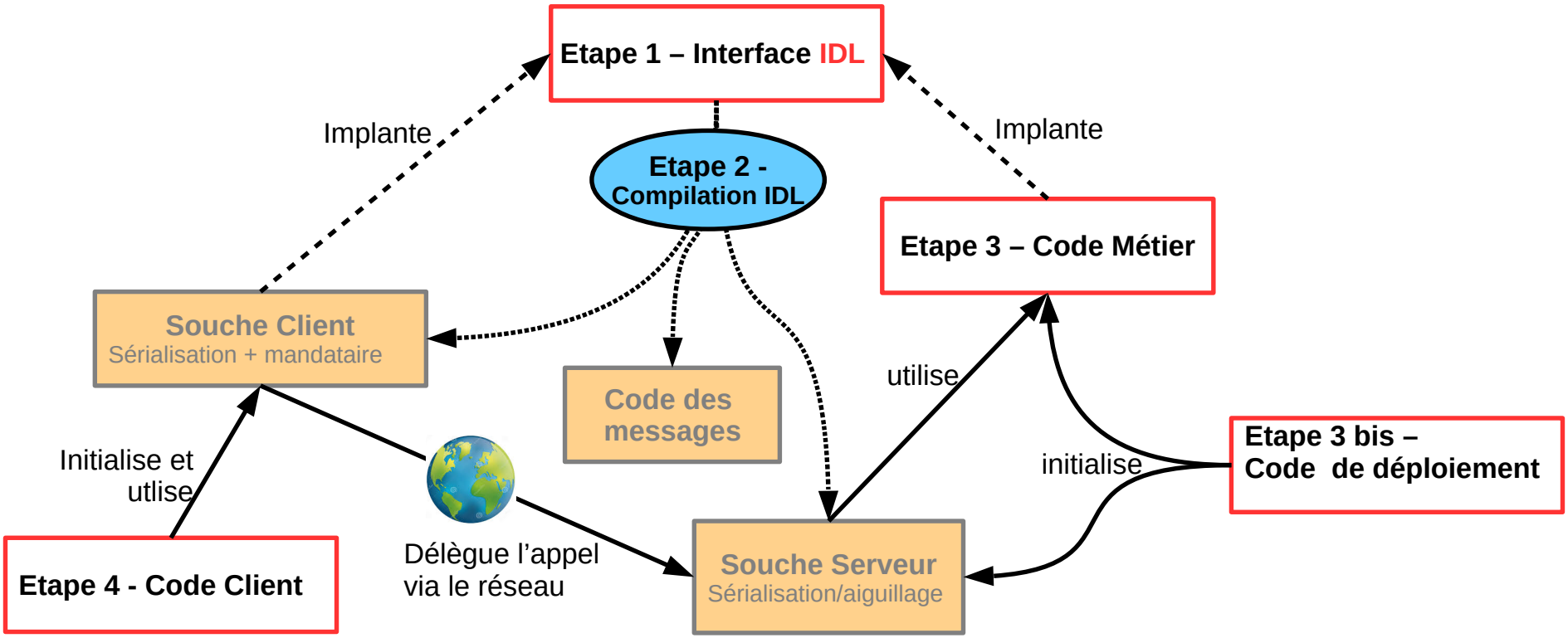
- NetFLix
- Cisco
- University of Wisconsin-Madison

# Construire une application

## Étapes

1) Déclaration des services accessibles à distance
   $\Rightarrow$ Écriture d'une interface IDL
2) Production des souches/squelettes
   $\Rightarrow$ Compilateur IDL vers un langage cible
3) Implantation des services et du déploiement
   $\Rightarrow$ Codage du serveur
4) Interaction avec le serveur
   $\Rightarrow$ Codage du programme client

# Construire une application



# Étape 1 - Le langage IDL Protobuf

## Caractéristiques

- Permet de définir les spécifications d'un service :
  - Messages (partie **Protobuf**)
    $\Rightarrow$ structures des données manipulées
  - Interface du service (partie **gRPC**)
    $\Rightarrow$ méthodes offertes par le service et invocable à distance
- Extension du fichier : `.proto`
- Langage générique compilable vers plusieurs langages
  $\Rightarrow$ le client et le serveur ne sont pas forcément écrits dans le même langage cible

**Interface IDL = Contrat entre client et serveur**

# Étape 1 - IDL : syntaxe générale

## Caractéristiques

- Commentaires : `/*multi-ligne*/` et `//une ligne`
- Identificateurs : `[letter][letter,0-9,_]*`
- Les instructions se terminent par un point-virgule
- Les littéraux de chaînes de caractères se délimitent par des double-quotes ou des simples-quotes

# Étape 1 - IDL Protobuf : les mots clés

## Proto3

| | |
|---|---|
| En-têtes du fichier | `syntax option import package` |
| Modificateurs d'import | `public weak` |
| Entités top-level | `message enum service` |
| Types simple de champs de messages | `double float`<br>`int32 uint32 sint32 sfixed32 fixed32`<br>`int64 uint64 sint64 sfixed64 fixed64`<br>`bool`<br>`string, bytes` |
| Valeurs booléennes | `true false` |
| Types complexes de champs de messages | `oneof map repeated` |
| Réservation d'id de champs | `reserved to max` |
| Définition des services | `rpc returns stream` |

Obsolètes (proto2) : `group extensions required optional default`

# Étape 1 - IDL Protobuf : En-têtes du fichier

## Déclarer la version utilisée

Obligatoire et première instruction du fichier

```
syntax = "proto3";
```

## Déclarer des options

- Permet de paramétrer la compilation
- Peuvent être déclarées aussi dans les messages, enum et services.

Exemples :

```
option java_multiple_files = true;
option java_outer_classname = "Ponycopter";
option optimize_for = CODE_SIZE;
```

# Étape 1 - IDL Protobuf : En-têtes du fichier

## Déclarer un package (optionnel)

- Évite les conflits de nom sur les entités top-level.

```
package foo.bar;
```

## Déclarer des imports

- Utiliser des types qui sont définis dans un autre fichier .proto

```
import "other.proto"
```

- Import transitif avec le modificateur `public`.

```
//f1.proto
```

```
//f2.proto
import public "f1.proto"
import "other.proto"
```

```
//f3.proto
import "f2.proto"
//f1.proto importé
    implicitement
//mais pas other.
    proto
```

# Étape 1 - IDL Protobuf : Messages

## Déclaration d'un message

- Définition d'un nouveau type structuré **sérialisable**

```
message identificateur {
  //liste des champs du message
};
```

## Champs

- Définit par
  - un type
  - un identificateur
  - un numéro d'identification (compris entre 1 et $2^{29} - 1$)
- Possède une cardinalité :
  - zéro ou une fois (par défaut)
  - zéro, une ou plusieurs fois (modificateur `repeated`)

*<type> <identificateur> = <numero>* ;

`repeated` *<type> <identificateur> = <numero>* ;

# Étape 1 - IDL Protobuf : les types simples des champs

| Type | Mots-clés | Caractéristiques |
|---|---|---|
| Booléen | `bool` | 2 valeurs : `true`, `false` |
| Flottant | `float double` | |
| Entier signé | `int32 int64` | encodage dynamique<br>inefficace pour les négatifs |
| | `sint32 sint64` | encodage dynamique |
| | `sfixed32 sfixed64` | encodage fixe, 4 ou 8 octets |
| Entier non signé | `uint32 uint64` | encodage dynamique |
| | `fixed32 fixed64` | encodage fixe, 4 ou 8 octets,<br>efficace en cas de valeur haute |
| Chaîne | `string` | caractère UTF-8<br>limité à $2^{32}$ caractères |
| Séquence d'octets | `bytes` | limité à $2^{32}$ octets |

# Étape 1 - IDL Protobuf : les énumérations

## Caractéristiques

- Définit un type de champs avec des valeurs prédéfinies
- Chaque valeur est associée à un numéro ($\geq 0$)
- Doit avoir un champs avec une valeur égale à zéro (valeur par défaut)

```
enum identificateur {
  <identificateur> = 0 ;
  <identificateur> = <numero> ;
  <identificateur> = <numero> ;
  ...
} ;
```

# Étape 1 - IDL Protobuf : autres types de champs

## Les `OneOf`

Déclare un ensemble de champs où au plus un champs peut être défini à un instant donné ($\simeq$ union en C)

```
message MonMessage {
  sint32 a =1;
  oneof foo{
    sint32 b =2;
    string c =3;
  }
}
```

## Les tables associatives

Associe un type clé à un type valeur

```
map<key_type, value_type> map_field = N;
```

Dans les deux cas, pas de `repeated`

# Étape 1 - IDL Protobuf : Service RPC

## Déclaration

Protobuf peut être associé à intergiciel RPC

⇒ gRPC est la solution la plus commune

```
service identificateur {
  //liste des prototypes de fonction rpc
};
```

## Définition des en-têtes de fonction

Une fonction d'un service se définit par :

- un nom
- un type de message paramètre (message aller)
- un type de message résultat (message retour)

```
rpc <nom> (<typeMessageAller> ) returns (<typeMessageRetour> ) ;
```

# Étape 1 - IDL Protobuf : Les méthodes à flux de messages

## Principes

- Possibilité de définir un flux de messages en paramètre ou en réponse
- Lecture du flux jusqu'à ce qu'il n'y ai plus de message
- L'ordre d'envoi de messages est préservé à la réception

## Les types de méthode à flux de message

- Coté serveur = flux de messages réponse

  `rpc` *<nom>* ( *<typeMessA>* ) `returns` ( `stream` *<typeMessR>* ) ;

- Coté client = flux de messages aller

  `rpc` *<nom>* ( `stream` *<typeMessA>* ) `returns` ( *<typeMessR>* ) ;

- Les deux cotés = flux de messages aller et flux de messages retour

  `rpc` *<nom>* ( `stream` *<typeMessA>* ) `returns` ( `stream` *<typeMessR>* ) ;

# Étape 1 - IDL Protobuf : Types prédéfinis de l'API Google

## Les messages Wrapper (*wrapper.proto*)

`BoolValue` `BytesValue` `DoubleValue` `FloatValue` `Int64Value`
`UInt64Value` `Int32Value` `UInt32Value` `StringValue`

## Les messages relatifs au temps

`Duration` (*duration.proto*) `Timestamp` (*timestamp.proto*)

## Autres messages utilitaires

- `Any` (*any.proto*) : contient un message quelconque
- `Empty` (*empty.proto*) : message vide

# Étape 1 - IDL Protobuf : exemple

```
syntax = "proto3";
package srcs.div;

import "google/protobuf/wrappers.proto";
option java_multiple_files = true;

message TwoDouble{ double a=1; double b=2; }

message TwoInt{ sint32 a=1; sint32 b=2; }

service Division{
  rpc divent(TwoInt) returns(TwoInt);
  rpc div(TwoDouble) returns(google.protobuf.DoubleValue);
}

service CalculStream{
  rpc plus(stream google.protobuf.Int32Value)
                    returns(google.protobuf.Int32Value);
}
```

# Étape 2 - Compilation vers un langage cible

## Généralités

- Projection de la description vers un langage cible :
  C++, Java , Python, Go, Dart, Ruby, C#, PhP, JavaScript
- Une commande pour plusieurs compilateurs : `protoc`

Commande pour compilation Java :

```
protoc --proto_path=<chemin répertoire de fichiers .proto>
       --plugin=<chemin vers binaire du plugin gRPC>
       --grpc-java_out=<chemin répertoire cible fichiers service rpc>
       --java_out=<chemin répertoire cible fichiers message>
       <fichier .proto à compiler>
```

# Étape 2 - Compilation vers Java : les champs des messages

| Type | Type IDL | Type Java |
|---|---|---|
| Booléen | `bool` | `boolean` |
| Flottant | `float` | `float` |
| | `double` | `double` |
| Entier signé | `int32 sint32 sfixed32` | `int` |
| | `int64 sint64 sfixed64` | `long` |
| Entier signé | `uint32 fixed32` | `int` |
| | `uint64 fixed64` | `long` |
| Chaîne | `string` | `String` |
| Séquence d'octets | `bytes` | `com.google.protobuf.ByteString` |

## Cas des champs `repeated`

Conversion en `java.util.List<...>` paramétré par le type du champ

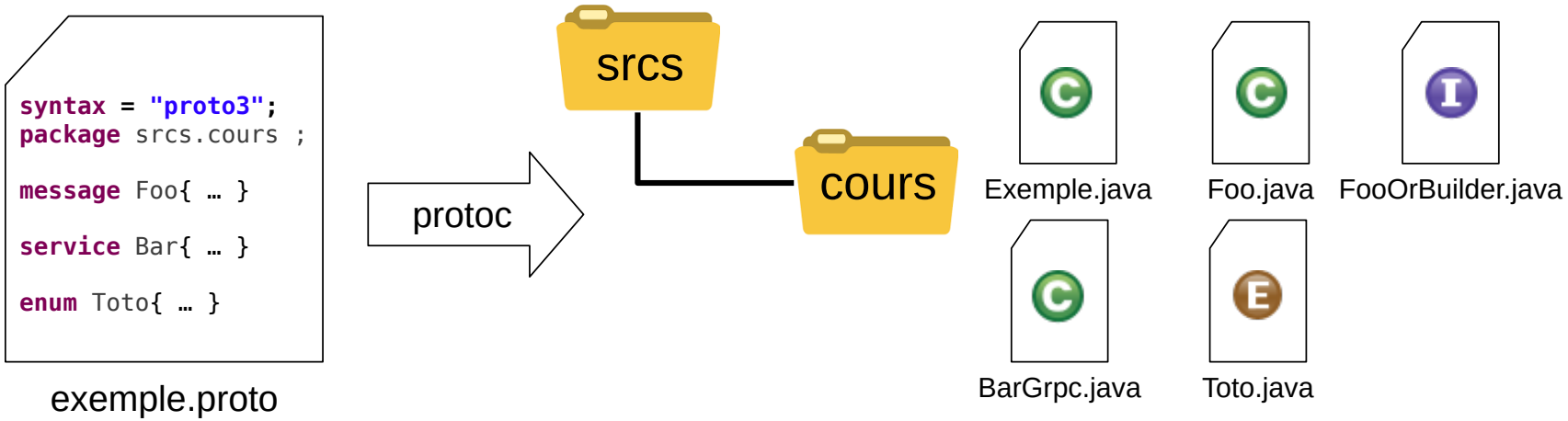
## Cas des champs `oneof`

Production d'une enum java pour sélectionner le champs actif

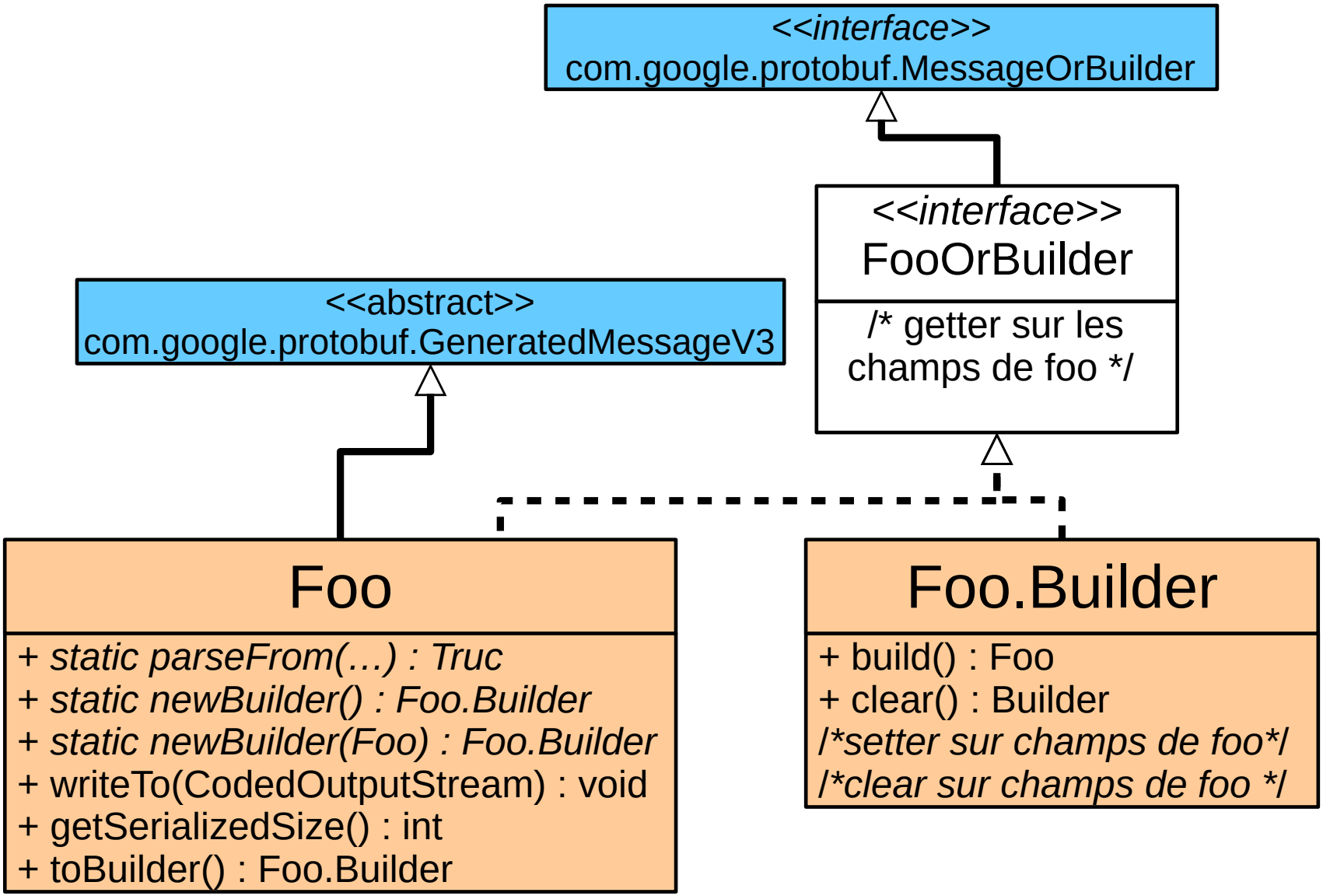
# Étape 2 - Compilation vers Java : fichiers générés

## Arborescence Java créée pour un fichier `exemple.proto`

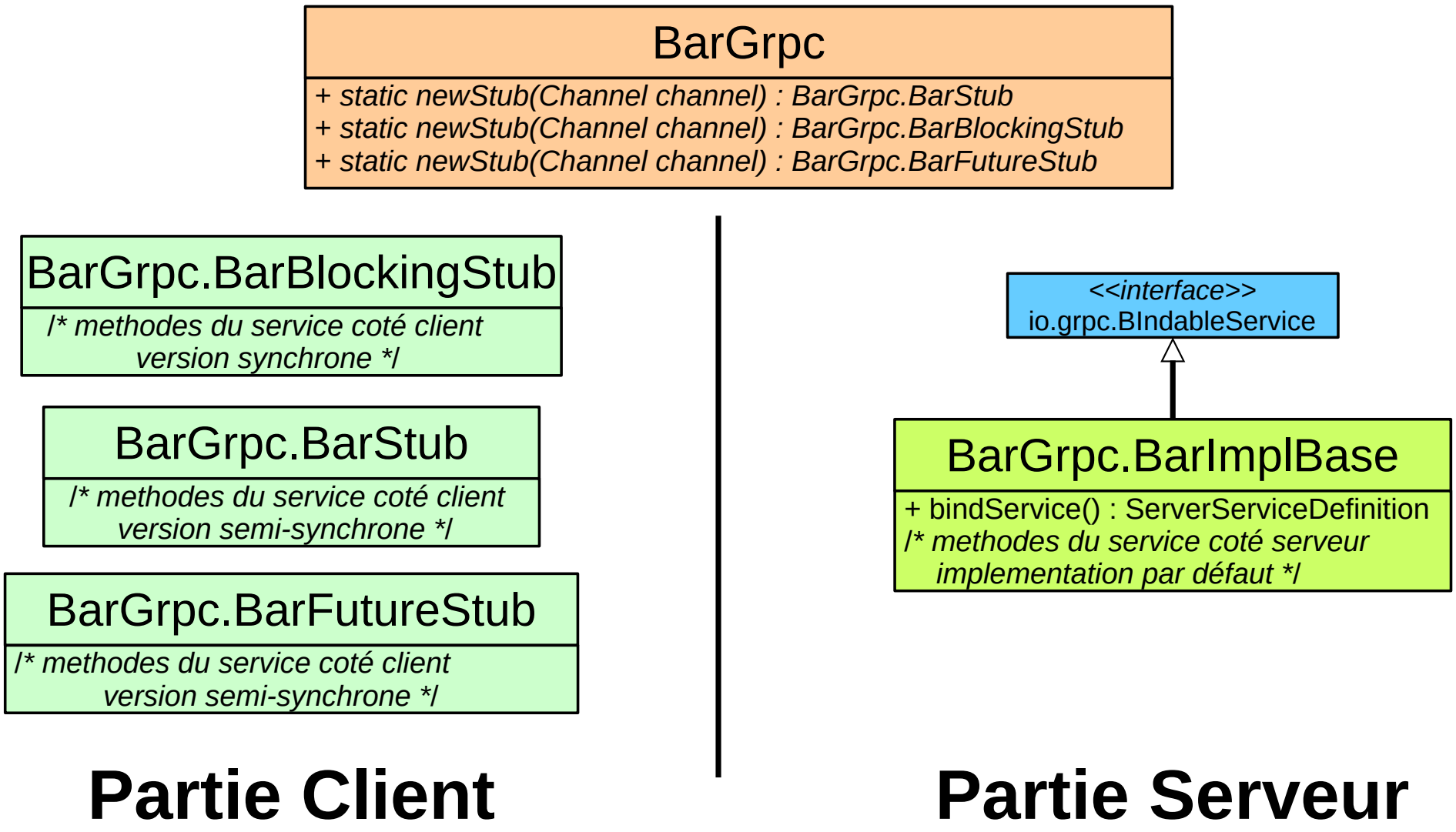
- le `package` protobuf produit un package java du même nom
- si `option java_multiple_files = true;` :
  - un fichier de méta-données *Exemple.java* (classe à champs `static`)
  - pour tout `message Foo {..}` $\Rightarrow$ :
    - *FooOrBuilder.java* (interface Java)
    - *Foo.java* (classe Java)
  - pour tout `service Bar{..}` $\Rightarrow$ *BarGrpc.java* (classe Java)
  - pour tout `enum Toto{..}` $\Rightarrow$ *Toto.java* (enum Java)
- si `option java_multiple_files = false;` : $\Rightarrow$ tout dans *Exemple.java*



# Étape 2 - Compilation vers Java : les messages



# Étape 2 - Compilation vers Java : les services



# Étape 2 - Compilation vers Java : fonctions et souches

## Coté client

- `f(A) returns(B)`

  `BarBlockingStub` → `B f(A request)`

  `BarStub` → `void f(A request, StreamObserver<B> resObs);`

  `BarFuturetub` → `ListenableFuture<B> f(A request)`

- `f(A) returns(stream B)`

  `BarBlockingStub` → `Iterator<B> f(A request)`

  `BarStub` → `void f(A request, StreamObserver<B> resObs);`

- `f(stream A) returns(B)` et `f(stream A) returns(stream B)`

  `BarStub` → `StreamObserver<A> f(StreamObserver<B> resObs)`

## Coté Serveur

La souche serveur offre les mêmes fonctions que la souche cliente `BarStub` :

⇒ la signature dépend du caractère stream de la requête de l'appel

# Étape 3 - Code serveur : code métier du service

## Comment procéder ?

- Créer une classe java qui étend `BarGrpc.BarImplBase`
- Y implanter le code métier des fonctions en les redéfinissant

### Cas classique

```
public void f(A request,
    StreamObserver<B> out){
  //code implementation
  out.onNext(reponse);
  // autres onNext eventuels
  // si retour = mode stream
  out.onCompleted();
}
```

### Cas stream sur requête

```
public StreamObserver<A> f(StreamObserver<B> out){
 return new StreamObserver<A>() {
   public void onNext(A value) {
     //traitement reception d'un A
   }
   public void onError(Throwable t){
     //traitement si erreur
   }
   public void onCompleted() {
     //traitement fin du flux de A
   }
 };
}
```

# Étape 3 bis - Code serveur : déploiement

```
int port = 1234;//port d'écoute du serveur
Server server = ServerBuilder.forPort(port)
                             .addService(new ServiceAImpl())
                             .addService(new ServiceBImpl())
                             // ...
                             .build();
server.start();//démarrage des services + écoute réseau
               //non bloquant

server.awaitTermination();//bloquant jusqu'à server.shutdown
```

**Remarque**

Le serveur utilise par défaut un pool de threads statique

# Étape 4 - Code Client

```
String host= "serveur.fr";
int port = 1234;
ManagedChannel chan = ManagedChannelBuilder.forAddress(host,port)
                                           .usePlaintext()
                                           .build();

A a = A.newBuilder().setX(..).setY(..).build();

//cas souche cliente bloquante
BarBlockingStub service = BarGrpc.newBlockingStub(c);
B b = service.f(a);

//cas souche cliente non bloquante
service.f(a, new StreamObserver<B>() {
    public void onNext(B value) {  //traitement reception B
    }
    public void onError(Throwable t) {//traitement erreur sur le
    flux
    }
    public void onCompleted() { //traitement fin flux
    }
});
```